DOI:10.7544/issn1000-1239.2018.20170687
55(2):246-264,2018

# 面向大数据处理的基于 Spark 的异质内存编程框架

王晨曦[1,2] 吕 方[1,4] 崔慧敏[1] 曹 婷[1] John Zigman[3] 庄良吉[1,2] 冯晓兵[1,2]

[1](计算机体系结构国家重点实验室(中国科学院计算技术研究所) 北京 100190)

[2](中国科学院大学 北京 100049)

[3](澳大利亚野外机器人中心(悉尼大学) 澳大利亚悉尼 2006)

[4](数学工程与先进计算国家重点实验室 江苏无锡 214125)

(wangchenxi@ict.ac.cn)

# Heterogeneous Memory Programming Framework Based on Spark for Big Data Processing

Wang Chenxi[1,2], Lü Fang[1,4], Cui Huimin[1], Cao Ting[1], John Zigman[3], Zhuang Liangji[1,2], and Feng Xiaobing[1,2]

[1](*State Key Laboratory of Computer Architecture* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)

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

[3](*Australia Centre for Field Robotics* (*University of Sydney*), *Sydney*, *Australia* 2006)

[4](*State Key Laboratory of Mathematical Engineering and Advanced Computing*, *Wuxi*, *Jiangsu* 214125)

**Abstract** Due to the boom of big data applications, the amount of data being processed by servers is increasing rapidly. In order to improve processing and response speed, industry is deploying in-memory big data computing systems, such as Apache Spark. However, traditional DRAM memory cannot satisfy the large memory request of these systems for the following reasons: firstly, the energy consumption of DRAM can be as high as 40% of the total; secondly, the scaling of DRAM manufacturing technology is hitting the limit. As a result, heterogeneous memory integrating DRAM and NVM (non-volatile memory) is a promising candidate for future memory systems. However, because of the longer latency and lower bandwidth of NVM compared with DRAM, it is necessary to place data in appropriate memory module to achieve ideal performance. This paper analyzes the memory access behavior of Spark applications and proposes a heterogeneous memory programming framework based on Spark. It is easy to apply this framework to existing Spark applications without rewriting the code. Experiments show that for Spark benchmarks, by utilizing our framework, only placing 20%~25% data on DRAM and the remaining on NVM can reach 90% of the performance when all the data is placed on DRAM. This leads to an improved performance-dollar ratio compared

**收稿日期**:2017-09-18;**修回日期**:2017-11-07

**基金项目**:国家"八六三"高技术研究发展计划基金项目(2015AA011505,2015AA015306);国家"九七三"重点基础研究计划基金项目(2016YFB1000402);国家自然科学基金项目(61402445,61672492,61432016,61521092);数学工程与先进计算国家重点实验室开放基金项目(2016A03)

This work was supported by the National High Technology Research and Development Program of China (863 Program) (2015AA011505, 2015AA015306), the National Basic Research Program of China (973 Program) (2016YFB1000402), the National Natural Science Foundation of China (61402445, 61672492, 61432016, 61521092), and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (2016A03).

**通信作者**:崔慧敏(cuihm@ict.ac.cn)

with DRAM-only servers and the potential support for larger scale in-memory computing applications.

**Key words** in-memory computing; Spark; heterogeneous memory; non-volatile memory (NVM); programming framework

**摘 要** 随着大数据应用的发展,需要处理的数据量急剧增长,企业为了保证数据的及时处理并快速响应客户,正在广泛部署以 Apache Spark 为代表的内存计算系统.然而 TB 级别的内存不但造成了服务器成本的上升,也促进了功耗的增长.由于 DRAM 的功耗、容量密度受限于工艺瓶颈,无法满足内存计算快速增长的内存需求,因此研发人员将目光逐渐移向了新型的非易失性内存(non-volatile memory, NVM).由 DRAM 和 NVM 共同构成的异质内存,具有低成本、低功耗、高容量密度等特点,但由于 NVM 读写性能较差,如何合理布局数据到异质内存是一个关键的研究问题.系统分析了 Spark 应用的访存特征,并结合 OpenJDK 的内存使用特点,提出了一套管理数据在 DRAM 和 NVM 之间布局的编程框架.应用开发者通过对本文提供接口的简单调用,便可将数据合理布局在异质内存之中.仅需 20%~25%的 DRAM 和大量的 NVM,便可以达到使用等量的 DRAM 时 90%左右的性能.该框架可以通过有效利用异质内存来满足内存计算不断增长的计算规模.同时,"性能/价格"比仅用 DRAM 时提高了数倍.

**关键词** 内存计算;Spark;异质内存;非易失性内存;编程框架

**中图法分类号** TP312

随着数据中心、内存数据库、内存计算的发展,需要处理的数据量急剧增加.企业为了快速响应用户的需求,需要对数据进行实时处理,因此将大量的数据置于随机读写性能良好的内存中,以求降低开销极大的磁盘 I/O 带来的性能损失[1].而目前使用的 DRAM 内存技术受限于工艺瓶颈,其存储密度难以继续增加,功耗持续升高,同时其价格也难以继续降低[2],阻碍了内存计算规模的继续增长.为了解决这些问题,一类新型的内存结构——非易失性内存(non-volatile memory, NVM),逐渐获得了研究人员的关注.非易失内存所具有的存储密度大、静态功耗低、价格便宜等特性[3]可以很好地满足内存计对内存容量持续增加的需求.然而,目前比较实用的非易失性内存,如 PCM(phase change memory)等,其延迟、带宽等性能方面和 DRAM(dynamic random access memory)之间有一定差距.因此,如何利用 NVM 来解决内存计算对内存容量快速增长的需求,是一个重要的研究问题.

Spark[4]是一种基于 JVM 运行、被业界广泛采用的分布式内存计算系统.其将计算的中间结果和一些容错信息存储在内存中,以加速计算、增强容错性.因此,其对内存的需求量随着计算规模扩大而快速增加.但是由于 DRAM 的工艺瓶颈,内存容量无法满足其计算需求,此时便会造成应用的性能快速下降.此时,造成 Spark 性能下降的主要原因在于内存不足时、对磁盘的频繁使用和 Java Heap 无法满足计算规模时造成的频繁 GC(garbage collection)行为.同时,由于 Spark 自身的数据使用特点和当前 GC 策略的不匹配[5],也加重了单次 GC 的开销.此时,GC 开销甚至占据了程序运行时间的 50%[5-6].

为了解决 DRAM 工艺限制导致 Spark 内存容量不足的问题,本文利用容量密度更大的 NVM 来扩展 Spark 程序的可用内存量.首先,本文利用 NVM 和 DRAM 性能接近的特点,直接将其用来扩展 Java Heap 的容量.同时,为了解决 Spark 自身计算行为导致的 GC 开销,我们利用 Unsafe 接口开发了供 Spark 编程人员使用的可以直接将数据移出 Java Heap 并置于 NVM 的 API,以便进一步减轻 GC 开销.

由于 DRAM 和 NVM 的性能具有一定的差距,因此需要将数据在二者之间恰当布局才能避免程序性能的下降.数据随机在 DRAM 和 NVM 之间分布时,和全部使用 DRAM 相比性能下降达 32%.本文提出的异质内存编程框架,可以在 Spark,OpenJDK 两个层次来进行数据布局.在仅适用 20%~25%的 DRAM 时,便可以达到全部使用 DRAM 时性能的 90%以上.

首先,由于 Spark 框架运行于 JVM 之上,因此其内存使用特点受到了 JVM 对 Java Heap 管理的影响,不同程序的访存特点具有一定的共性.本文通过对多个 Spark 程序的分析,发现 Java Heap 中容量较小的新生代、元数据区的内存读写比例远高于

容量庞大的旧生代区.因此,我们根据各个区域的访存热度将区域直接映射到了 DRAM 和 NVM 区.

同时,Spark 程序的访存特点也受到了其自身计算行为的影响.Spark 程序使用的数据(RDD),结构规则,计算行为简单,在构造 Spark 应用时,开发人员很容易感知发生在特定数据上的计算、访存行为.因此,我们基于 Off Heap 机制开发了 API 来允许开发人员直接决定数据在 DRAM 和 NVM 中的位置.

此外,本文通过对 Spark 应用访存行为的分析,发现内存中的大量数据访问频率低下.这些数据无法持续利用 DRAM 的性能,如"低延迟"和"超高带宽(多通道)".同时,这些数据的存活时间很长,每一次 GC 都会因为对这些数据的分析而造成显著的开销.因此,将这些大量的冷数据通过 Off Heap 置于 NVM 时,虽然会带来一定的序列化开销,但显著地减少了 GC 开销.而又比将数据置于磁盘带来十分可观的性能提升(可达 88%).

最后,为了减少 Spark 程序开发人员的负担,数据在 OpenJDK 层次的映射将会自动完成;此外,我们在 Spark 层次提供了简单的 API 接口,供开发人员根据 Spark 程序特点来布局数据.本文在第 3 节中对 API 使用规则进行了描述,使其非常易于使用.

经过实验分析,本文提出的异质内存编程框架在搜索引擎离线分析(Spark PageRank[7])、图计算(Spark GraphX)、社交网络离线分析(Spark K-means[7])等领域中均取得了较好效果.

本文的主要贡献如下:

1) 利用 NVM 高密度、接近 DRAM 性能的特点,将其用于服务器主存,在 Java Heap,Off Heap 两个方面扩展了 Spark 编程框架的内存容量.解决了由于 DRAM 容量无法满足内存计算需求增长所带来的性能瓶颈.相较于 DRAM 不足时使用磁盘,使用 NVM 后应用性能提升了 88%.

2) 通过分析 Spark 程序的行为,实现了数据在异质内存上的合理布局.在仅用少量 DRAM(20%~25%)和大量 NVM 的情况下,便可达到使用等量 DRAM 性能的 90%左右.让 NVM 可以实际应用于内存计算,使其"性能/价格"比全部使用 DRAM 高了 2.9 倍.

3) 编程框架提供了简单的 API 和自动布局策略,有很好的易用性,无需编程人员重新编写已经开发的应用,仅需要简单的 Storage Level 替换和相关功能开启便可以将原有的 Spark 应用运行于异质内存.

## 1 研究背景

### 1.1 内存计算系统

大数据是继云计算、移动互联网、物联网之后的一个新兴的前沿领域.MapReduce 是一种流行的大数据编程框架,开发者只需要将数据用 Map,Reduce 函数串行处理,便可以利用高扩展性、容错性的分布式计算框架来进行高并发的处理.该框架目前广泛应用于主流大数据处理平台,如 Hadoop,Spark 等.

以 Spark 为代表的内存计算框架正在兴起,并被企业广泛采用.通过将大量的中间结果缓存于内存,极大地减少了传统 Map,Reduce 的计算过程中对磁盘的使用,尤其适用于加速迭代计算.然而,随着数据量的快速增加,计算规模也逐渐增大,因此对内存容量提出了迫切的需求.然而由于工艺水平的限制,目前主流的内存介质,DRAM 已经很难满足内存计算需求的快速增长.当内存容量不足时,内存计算性能将受到严重的干扰,限制了计算规模的增长.此外,DRAM 功耗也随着存储密度的增加而逐渐增加,甚至可以占到某些服务器整机功耗的 40%以上[8].

### 1.2 托管式语言

托管式语言(如 Java,C#,Scala 等)因具有跨平台、面向对象、垃圾回收机制(garbage collection,GC)等特性,被广泛应用于大数据系统开发,如 Spark[4],Hadoop[9] 等均基于 JVM(Java virtual machine)运行.托管式语言利用自己的堆(Java Heap)机制来管理从操作系统中获取的内存空间,因此运行于其上的应用访存特点均具有一些类似的特性.比如,其新生代的内存写比例明显大于旧生代等.这种特点给研究人员提供了进行数据快速布局的天然优势.

### 1.3 非易失性内存

非易失性内存(NVM)是一类具有非易失性的新型内存介质.其不但具有和 DRAM 类似的性能,同时具有更高的存储密度、更低的静态功耗.但是,其也具有一些自身的缺点,相比于 DRAM,其写性能低下、写寿命较低.因此,难以直接使用来全面替代 DRAM.目前异质内存中,以 PCM 较为成熟,吸引了众多研究者的探索.但利用 DRAM 和 NVM 构成的异质内存却可以满足一些大数据计算应用的需求.

由 DRAM 和 NVM 构成的异质内存有 2 种主流结构:DRAM 作为 NVM 缓存的竖直结构

(inclusive architecture);DRAM 和 NVM 平级的结构(exclusive architecture),如图 1 所示.相比于前者,在大规模计算中,后者允许我们有更大的物理内存可以使用,也减少了 DRAM 和 NVM 之间巨量的数据移动[8],因此本文工作基于平级结构.

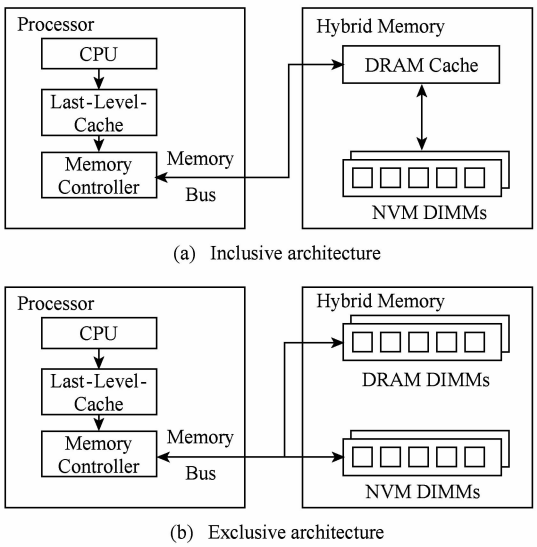


Fig. 1 Architecture of heterogeneous memory[8]

图 1 异质内存结构[8]

## 2 研究现状

本研究涉及了上层分布式编程框架、中层的托管式运行时系统、底层的非易失内存 3 方面,所以本文将从这 3 个方面之间的联系来对国内外的研究工作进行探讨.

### 2.1 非易失性内存的管理优化

由于非易失性内存(NVM)是一类内存技术的统称,其中包含了性能各异的介质类型.目前针对 NVM 的研究,在应用层次上可以分为两大类:将 NVM 作为存储介质;将 NVM 作为内存介质.由于闪存技术已经大规模的商用化,因此通过闪存来研究 NVM 在存储层次的应用较为成熟,清华大学总结了当今 NVM 在存储领域的应用情况[10-11].而对于 NVM 在内存层次的研究多以利用模拟器的方式进行,中国科学院计算技术研究所对 NVM 应用于内存层次时的性能瓶颈进行了透彻的分析[12].此外,还有针对 NVM 安全[13]、功耗[14]等方面的研究.本文将 NVM 应用于内存层次,将其和 DRAM 同时使用,构建成大容量、低价格的异质内存.

对于由 DRAM 和 NVM 构成的异质内存,如果不辅以恰当的数据布局,会因为 NVM 的性能不足导致严重的性能瓶颈.因此,针对不同的应用场景对异质内存进行数据布局的管理工作是一个研究热点.根据数据管理方式分类,常见的有硬件管理策略;硬件、操作系统结合的管理策略;运行时/应用/程序员管理策略.本节按照该种分类形式,对国内外现有的管理策略进行介绍.

#### 2.1.1 硬件管理策略

硬件管理策略多采用缓存结构,此时 DRAM 为 NVM 的缓存,通过类似于 Cache 的管理策略来将 NVM 中频繁访问的数据取到 DRAM 中,对上层操作系统、程序完全透明.其中,IBM 研究院提出了由硬件管理的异质内存缓存结构,并提出了延迟写、细粒度写回等策略,在保证性能的前提下提高 PCM 的使用寿命[15].最终,该研究认为只需使用 NVM 容量 3% 的 DRAM,就可以使异质内存和同容量 DRAM 的性能接近.此外,匹兹堡大学[16]、密歇根大学[17]、宾夕法尼亚州立大学[18]、微软研究院[19]等也提出了采用硬件来对异质内存进行管理,以节省功耗、提升 NVM 寿命的管理策略.由硬件进行冷热数据识别、数据迁移开销较小,但是硬件管理策略不适用于内存容量巨大的大数据应用,因为如果 NVM 容量太大,DRAM 缓存的 tag 等标志位会造成很大的浪费.如文献[15]中所用配置,32 GB NVM,1 GB DRAM,16 way,256 B line 时,便需要 13% DRAM 空间用作 tag 等标志位,而本文的实验规模在 128 GB,面向的应用环境在 TB 级别.此外,如果大数据程序的局部性不好,会导致数据频繁的换入、换出 DRAM,反而会造成更高的功耗,同时引起性能下降.

#### 2.1.2 软硬件结合管理策略

该种方式可以由硬件(内存控制器)来完成冷热数据的识别、数据迁移,然后由操作系统来进行物理页面和虚拟页面的映射维护[20-21];或者由操作系统进行在线的分析,识别出性能关键数据,并在 DRAM 和 NVM 之间进行数据迁移[22-24];亦或者由硬件、操作系统、上层应用联合完成数据识别、迁移等工作[8].该种管理方式可以应用于缓存结构和平级结构,较为灵活.如罗格斯大学提出了软硬结合的 Rapp(rank-based page placement)算法[20],其利用内存控制器维护了一个多级队列,每一级对应访问热度不同的页面,并根据此将热页面换入 DRAM,冷页面替换出到 NVM.但是基于历史使用信息的在线监测,会造成一定比例的错误预测.同时,这种在线监测也会带来一定的开销.中国科学院计算技术研究所通过对程序的具体分析发现,数据可以根据

产生位置被划为为不同的类别,而且同一类别往往具有类似的访存行为[8].并基于此提出了 Rapp 的改进算法,即,首先根据静态分析对数据进行初步的分类,并直接布局到 DRAM 或 NVM;然后在此基础上应用 Rapp 算法,可以有效减少数据迁移数量和监测开销.类似的数据分类研究还有文献[25-26].硬件与操作系统结合的管理方式多是以 page 为粒度,并使用基于历史访问信息的在线监测方式.但是对于不同类型的程序,固定的 page 粒度往往不是最优的选择,如文献[27]通过研究发现在某些应用中(jpeg encoding),以 object 粒度进行数据迁移可以比 page 粒度进行数据迁移获得更好的功耗收益.然而,对内存使用量巨大的大数据应用来说,page 却显得过于细粒度,并且会造成较大的监测开销.而且,由于大数据系统中运行时堆的存在,其自身便会对数据进行移动,往往会破坏操作系统优化好的数据布局.

2.1.3 运行时/应用/程序员管理策略

程序的计算行为、数据的结构特点往往决定了数据被访问的特点.Intel 研究院、佐治亚理工学院通过对大数据程序进行分析,根据数据访问的形式,将其分为顺序访问、指针类型数据访问、随机访问 3 类,而且发现每一类数据的访问开销也各不相同.如对于同样数量的读写,指针类型数据的访问延迟开销远大于被顺序访问的数据.因此该研究提出了结合静态分析来调整数据在异质内存中布局的方式[3].但是该研究提供的数据布局操作只能作用于 C,C++实现的大数据系统,并没有能力控制基于托管式语言实现的大数据平台中的数据.而流行的大数据系统 Hadoop,Spark 等,往往是基于托管式语言实现的.类似地,文献[28]也提出了通过编译静态分析的方式来识别数据上的访问频度,并指导数据在 NUMA 节点上进行迁移.相对于操作系统,硬件往往以 page 为数据的迁移粒度,编译器、程序员往往可以做到更加灵活的数据管理粒度,如文献[27]提出了编译分析和操作系统相结合的方式,在 object 级别进行细粒度冷热数据分析、迁移.不过,应用直接管理 NVM 上的数据往往需要操作系统给予一定的支持,对此佐治亚理工学院提出了一套允许处理器可以直接访问 NVM 的系统 PMFS[29].但是,数据的访问特点往往是随着程序执行而不断变化的[30],因此需要配合运行时的监测来对数据布局进行动态的调整才取得更大的收益.同时,编译分析往往需要对应用进行离线分析(off-line profiling),而当程序的输入集改变时程序行为可能进行相应的变化,导致离线分析的准确性会有所降低.

本文面向的应用环境内存规模单节点便在上百 GB,甚至讨论了 TB 级别.因此,上述的细粒度软件、硬件策略并不能很好地契合本文研究对象的需求.此外,由于本文所面向的内存计算框架 Spark 运行于 OpenJDK 之上,上述的硬件、操作系统、编程接口等数据管理策略并没有考虑到托管式运行时对数据的移动,所以很可能造成数据的无效迁移.

## 2.2 大数据系统的优化工作

本文的应用环境为分布式内存计算框架.因此,我们研究了当今一些针对大数据系统进行优化的主流工作作为参考.这些优化工作给本文的性能调优提供了很好的启发.

随着上层软件系统的发展,越来越复杂的软件层次导致了程序执行特点、语义信息的多样性与复杂化.而上层程序的语义在下传过程中的逐渐丢失,导致运行时和上层软件系统之间出现了多个层次的不匹配行为.如垃圾回收机制中“生代假设”和大数据系统数据使用行为的不匹配[5];运行时处理数据的粒度(object)和大数据系统数据的使用粒度(如 Spark 中的 RDD)不匹配.这些不匹配造成 GC 策略效率低下,成为了内存计算框架的性能瓶颈,甚至可以造成 50%以上的开销.为了解决运行时和和大数据系统的不匹配行为,国内外学者从 2 个方面进行了研究:1)从运行时出发,修改运行时机制匹配大数据系统的数据使用行为;2)从大数据系统出发,修改其数据使用行为,以便符合运行时的管理特点.

2.2.1 面向大数据系统调整运行时的工作

University of California,Irvine 的研究者通过对 Hyracks,Hadoop,GraphChi 大数据系统的研究,发现了适用于大数据系统的数据生命周期假设——“epoch 假设”,并在 HotSpot 基础上提出了新的 GC 机制——Yak[5].其研究发现,大数据计算行为会导致数据生命周期具有 epoch(世代)的特点.大多数的数据在 epoch 开始时产生,其生命周期一直持续到对应的 epoch 结束.而当前运行时的 GC 策略使用的数据生命周期假设大多数数据对象的生命周期都是短暂的,因此可以通过频繁的轻量级 GC(minor GC)来对多数无用数据进行回收,并将少量的存活时间较长的数据迁移到旧生代,等待全局 GC(full GC)处理[31-32].而一些大数据系统的数据生命周期不再遵循这种假设,因此其触发的 minor GC 行为多数是无效的,只能回收极少量数据,同时带来巨大

的分析开销.因此,文献[5]针对这些大数据系统提出了新的数据使用假设——epoch 假设,并利用该假设在运行时堆中划分出一个新的区域进行管理.但是,经过我们的分析,epoch 假设不适用于 Spark 等内存计算系统.Spark PageRank 执行的过程中,每一次 GC 均会回收较多的无用数据.因此该研究的适用范围具有一定的局限性.

2.2.2 调整大数据系统来重新匹配运行时特点

传统的 GC 分析策略无法应对大数据系统产生的大量数据对象[33].GC 需要分析众多的数据引用,而这些分析导致 GC 消耗的时间甚至占据了整个程序运行时间的 50%以上[5-6].而研究者发现,大数据系统中产生的数据对象生命周期较为规律,所以提出了 Off Heap 的思想.即将大量的、生命周期规律的数据移出 Java Heap,置于本地内存(native memory),由程序员或者编译器来进行释放,不再由 GC 进行分析.Databrick 和 University of California,Irvine 的研究者分别提出了自己的 Off Heap 框架,见文献[6,34].其中文献[6]使用编译器对代码进行分析和变换,并使用 region-based 内存管理思想[35-37]来进行数据管理.文献[34]对程序员提供了 Off Heap 接口,可以让开发人员根据需求将对应的数据分配到 Off Heap.经过我们分析,通过编译变换的手段,仅能支持较少的大数据系统,如无法对 scala 语言实现的 Spark 系统进行变化.同时,将数据的创建、销毁任务重新返回给程序员,不但增加了编程难度,同时减弱了复杂系统的稳定性.

华中科技大学和 Databricks 面向 Spark 计算框架,分别提出了将常规 Java 数据对象压缩为字节流,并直接在字节流上进行计算的思路[34,38].通过这种数据类型的转化,不但减少了数据所占用的空间大小,还极大地减少了数据对象的数目,因此减少了 GC 带来的开销.但是,将常规 Java 数据对象转化为字节流,并直接在其上进行计算的操作,该方式对 Java 数据对象的结构有一定限制,限制了大数据系统的适用范围.同时,这种计算虽然节省了 GC 开销,但要比直接在 Java 数据对象上带来更大的计算开销,因此当需要 GC 的数据较少时,这种框架并不一定能取得优势.

通过对该部分研究工作的调研,让我们了解了内存计算系统的性能瓶颈,怎样平衡序列化的 CPU 计算开销和 GC 的开销、Off Heap 机制的利用等问题.

### 2.3 利用托管式语言对异质内存进行管理

托管式语言,如 scala,Java,C#等,因为其易编程性和良好的扩展性被广泛地应用于嵌入式开发、分布式系统开发.托管式语言会自动控制数据在堆中的位置.另外其所具有的垃圾回收机制(GC)会在程序运行过程中动态地回收程序无用的数据,并在 Object Promotion,Compact 等过程中以 object 粒度移动堆中的数据.这给通过运行时对异质内存中的数据进行管理提供了机会.如澳大利亚国立大学提出了一套硬件、操作系统、运行时结合的 NVM 管理方式来延长 NVM 的使用寿命.当硬件监测到 NVM 局部损坏时,相比起直接舍弃掉整个 page(4KB),运行时会避免使用对应的 line(256B).在该种策略下,即使 NVM 的损坏率达到 50%时,仅会引起 12%的性能下降[39].筑波大学[40-43]提出可以将运行时堆中的新生代直接映射到 DRAM,但是并没有给出具体的定量分析来进行解释.文献[44]对 Java Heap 各个区域的访存冷热进行了分析,并提出了在函数粒度对冷热数据进行识别,并在 Object 级别迁移的想法.但其面向的应用为单机应用程序.此外文献[45-46]等工作提出了如何在执行 GC 时避免发成 swap 数据到硬盘现象,当异质内存为缓存结构时,可以借鉴这些研究的成果.

该部分工作均为基于传统的单机 Java 程序进行.对于 TB 级别的分布式内存计算系统,其计算、访存行为将会发生很大的改变.本文重新对 TB 级别的大规模内存计算系统进行了分析,并提出了高效、可行的数据布局策略.

## 3 基于 Spark 的异质内存编程框架

Spark 和 OpenJDK 均有自己的内存管理模型,数据的布局同时受到二者的内存管理特点影响.因此,为了使数据可以在异质内存中合理地布局,我们在 Spark 和 OpenJDK 两个层次进行了异质内存支持.在 Spark 层次提供了开发人员可以主动调用的数据布局 API;在 OpenJDK 层次我们根据 HotSpot 的 Java Heap 管理特点提供了冷、热数据的自动布局方案.二者的对应关系如图 2 所示.

本节以 Spark 层的内存管理模型为切入,对提出的异质内存编程框架的设计与实现进行阐述:

Spark 将内存划分为 Storage Memory,Execution Memory 两大部分.Storage Memory 中为缓存的 RDD 数据;而 Execution Memory 中为 Map,Shuffle 等计算数据.

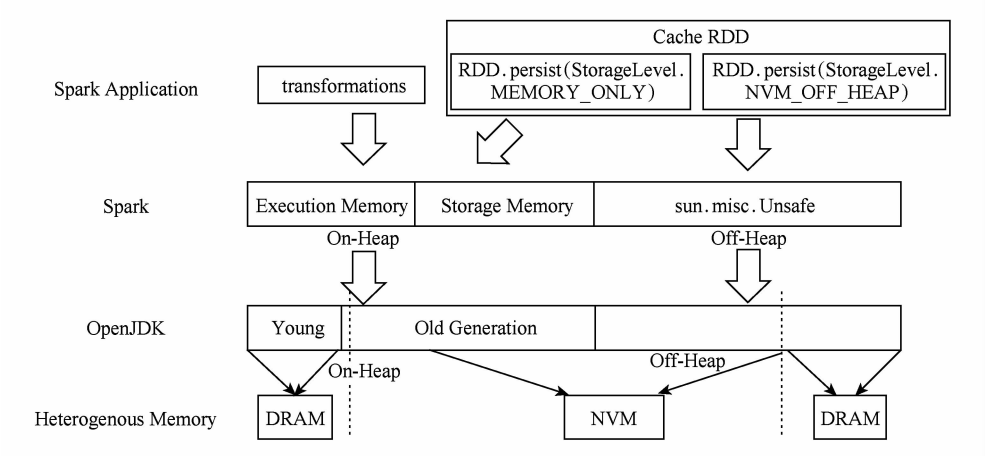


Fig. 2 Architecture of heterogeneous memory programming framework based on Spark

图 2 基于 Spark 的异质内存编程框架结构

对于 Storage Memory 中的数据,其由存活时间较长、数量巨大的 Object 群构成,因此会给 GC 的扫描过程带来显著的开销. 同时,Storage Memory 中的众多缓存数据主要用于容错,其访问并不频繁,因此可以将其中的大量冷数据序列化、压缩后,通过 Off Heap 机制置于 NVM. 由于 RDD 数据的冷热可以较容易地由 Spark 应用开发人员识别,因此我们提供 API 来供其控制数据布局.

对于 Execution Memory 中的数据,其只能存在于 Java Heap 中. 因此,为了解决 DRAM 无法满足内存计算规模增长而造成的 Java Heap 相对于计算数据量较小时频繁 GC 带来的开销,我们利用存储密度远大于 DRAM 的 NVM 来扩展 Java Heap. 但是,由于该部分数据访问频繁,对内存性能(带宽、延迟)有较高要求. 因此,我们进一步利用 OpenJDK 对其中的冷、热数据进行了自动布局.

为了减少 Spark 应用开发人员的复杂性,我们仅仅暴露出了一个简单的 Storage Level 接口. 一旦开发人员使用了我们的接口来指导 Storage Memory 中数据的布局,OpenJDK 便自动对 Java Heap 中数据的布局进行自动管理. 即使用户不使用我们提供的 Storage level 接口,也可以通过命令快速开启 OpenJDK 层次的数据自动布局.

### 3.1 Spark Storage Memory 管理接口

Storage Memory 用于储存被 persist,cache 的 RDD 数据,这些数据为一些中间数据,或者用于容错,其中众多的数据访问并不频繁. 为了降低 GC 开销,Spark Tungsten 利用 OpenJDK 的 unsafe 接口,开发了 Spark Off Heap 机制. Spark 和 OpenJDK 的内存管理对应关系如表 1 所示:

**Table 1 The Relation between Spark and OpenJDK Memory Management**

**表 1 Spark 和 OpenJDK 内存管理模型之间的对应关系**

| Spark Memory Management | OpenJDK Memory Management |
|---|---|
| Execution Memory<br>Storage Memory | Java Heap |
| Off-Heap Storage RDD | Off Heap |

Storage Memory 中的数据可以通过 Off Heap 机制,由 Spark 应用开发人员细粒度地决定其在异质内存中的位置;我们增加了一个 Storage Level 来让 Spark 应用开发人员直接将持久化的 RDD 置于 NVM:NVM_OFF_HEAP;同时,我们将原有的 OFF_HEAP 重命名为 DRAM_OFF_HEAP. 此时 RDD 所具有的 Storage Level 如表 2 所示:

**Table 2 RDD Storage Level**

**表 2 RDD 存储等级**

| Storage Level | Description |
|---|---|
| cache()/persist() | same to MEMORY_ONLY |
| DISK_ONLY | Serialization & Store to disk |
| MEMORY_ONLY | Store to memory |
| MEMORY_ONLY_SER | Serialization & Store to memory |
| MEMORY_AND_DISK | Store to disk if memory runs out |
| MEMORY_AND_DISK_SER | Serialization & Store to disk if memory runs out |
| DRAM_OFF_HEAP | the original OFF_HEAP |
| NVM_OFF_HEAP | Extended Storage Level |

Note: Only list partial RDD storage levels which affect RDD's position and status.

Spark 程序开发人员只需要将原有程序中的 DISK_ONLY, MEMORY_AND_DISK, MEMORY_

AND_DISK_SER 等携带“DISK”,“SER”关键字的 Storage Level 直接替换为本文提供的 NVM_OFF_HEAP,此时便将冷数据置于了 NVM Off Heap 中. 而对于一些只能处于非序列化状态的频繁访问的热数据,仍旧将其保留在 Java Heap 中,交由 OpenJDK 来控制其在异质内存中的布局,如图 3 所示. 该 API 十分易用使用,并未给开发人员增加额外额性能分析负担.

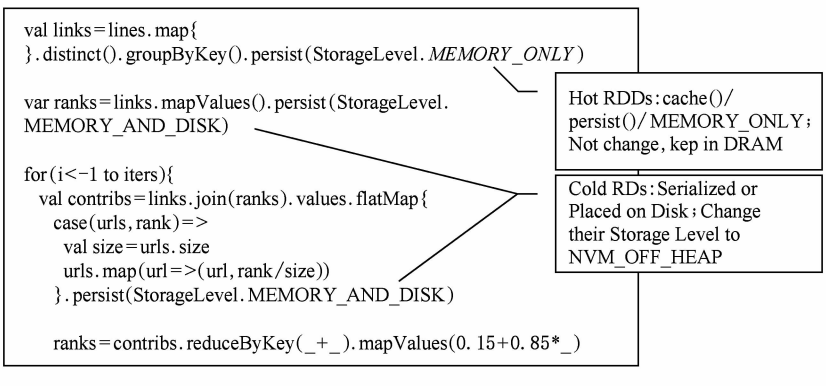


Fig. 3 An example of API: NVM_OFF_HEAP's usage

图 3 NVM_OFF_HEAP 接口用法展示

而对于有经验的开发者,仍旧可以根据自己的经验,将持久化的 RDD 置于 Off Heap 的 DRAM 区域,或者序列化后直接置于 Java Heap 中. 该框架在提供易用性的同时,并未减少 Spark 对于 RDD 控制的灵活度.

### 3.2 Spark Execution Memory 在异质内存中的布局

Execution Memory 用于存储进行 map,shuffle 等操作的计算时的数据,其上会频繁地发生计算、访存行为. Execution Memory 中的数据和 Storage Memory 中非序列化的 RDD 只能置于 Java Heap 中,而数据在 Java Heap 中的布局均由 OpenJDK 控制,应用程序无法直接进行布局. 因此,我们需要在 OpenJDK 层次进行修改,来控制应用数据在异质内存中的布局.

不同于磁盘,NVM 具有良好的随机读写性能,不但可以利用 Off Heap 机制来储存序列化的 RDD,也可用来扩展 Java Heap,减少频繁 GC 带来的开销. Java Heap 中的数据需要频繁的访问,但 NVM 相较于 DRAM 性能较弱,因此需要将 Java Heap 中的数据在异质内存中进行合理布局,否则性能会显著下降. 本文使用了文献[44]的部分研究成果. 该研究针对的是单机常规 Java 应用,但是经过我们分析,Spark 这类分布式内存计算应用同样具有类似的性质.

OpenJDK 使用自己的内存模型来管理从操作系统中获取的空间. 目前其采用的 HotSpot 机制,将内存划分为新生代、旧生代、元数据区三大部分. 新生代又细分为 eden/from/to 三个区域. 按照文献[44]中的分析方法,我们发现新生代仍然占据了大部分的 cache miss,并伴有大量的内存写操作. 不同于文献[44]中的结论,此时的 cache 已经无法处理内存计算应用中较大的元数据区,其会频繁地造成 cache miss. 所以,本文将新生代、元数据区直接定位到 DRAM. 而庞大的旧生代会随机使用余下的其他区域.

一旦 Spark 程序开发人员使用本文提供的 Storage Level 来将部分持久化的 RDD 布局到 NVM 区域,OpenJDK 便会自动布局新生代、元数据区到 DRAM 区域. 而其他部分将会自由使用剩余的 DRAM 和 NVM. 或者不用 NVM_OFF_HEAP 接口时,可以由单一的命令:-XyoungGenDRAM,开启 OpenJDK 的自动映射功能. 为了确保新生代可以完全至于 DRAM 区,框架在尽可能少影响性能的前提下,限制了新生代、旧生代的容量. 此时虽然可能造成更多的 Minor GC,但是却极大地减少了在 NVM 上的访存数量. 同时,该方案可以减少对 DRAM 的需求,缓解了由于 DRAM 容量不足而造成的内存计算规模限制.

本文在 Spark,OpenJDK 两个层次,分别提供了手动调用的 API 和 Java Heap 中数据的自动映射 2 种相互配合的方式,来将数据布局到异质内存. 仅向 Spark 程序开发人员暴露简单的编程接口,便可将 Spark Storage Memory,Spark Execution Memory 中的数据合理地布局到 DRAM 和 NVM. 通过使用 NVM 来减少内存计算系统对 DRAM 容量的需求,解决了由于 DRAM 工艺瓶颈对内存计算规模持续

扩展的限制. Spark 编程框架中的数据布局形式如图 2 所示. 本文对 Big Data Bench 中提供的搜索引擎离线分析(PageRank)、图计算(GraphX)、社交网络分析(K-means)等程序进行了性能分析,并取得了较好的结果.

## 4 Spark 内存使用特点和框架设计原则解析

本节针对 Big Date Bench 中的 PageRank[7] 应用,分别在磁盘 I/O、GC 开销、Spark 访存特点等方面进行了实验分析,剖析了内存不足时造成 Spark 性能瓶颈的具体原因. 结合 Spark 程序的性能瓶颈,本节解释了编程框架的设计原则,并展示了如何通过使用该编程框架在异质内存中取得良好的性能.

经过分析发现,当物理内存容量不足时,Spark 仍旧会产生大量的磁盘操作,造成较高比例的 I/O 开销;同时由于 Java Heap 空间不足,大规模的计算将会造成频繁的 GC,严重地降低了性能;此外,由于 Spark 程序会缓存众多的中间结果到 Java Heap 来加速计算、用于容错,这也给 GC 带来了显著的开销. 同时,在使用 NVM 时,需要合理布局数据,否则会因为 NVM 性能不如 DRAM,造成严重的性能下降. 4.4 节阐述了如何利用本文提出的编程框架解决以上问题. 通过使用本文提出的异质内存编程框架,可以在仅使用 20%～25%DRAM 的情况下,达到使用等量 DRAM 性能的 90%以上,并比 DRAM 不足,使用磁盘时加速 88%.

本节首先讨论了内存容量不足时带来的磁盘 I/O 开销和 GC 开销的问题. 在分析其原因的同时,展示了充足内存如何带来可观的加速比. 随后展示了如何通异质内存编程框架,利用性能较差的 NVM 来达到接近充足 DRAM 时的性能. 并讨论了该 Spark 编程框架在更大计算规模上的扩展性. 最后讨论了由于异质内存编程框架使用了序列化而带来的额外 CPU 计算开销问题.

### 4.1 实验平台以及配置

1) NVM 模拟

本文利用 NUMA 平台来模拟 DRAM 和 NVM 构成的平级异质内存架构[44]. 将计算固定到 CPU0,此时 Node0 上的本地内存为 DRAM,Node1 上的远端内存为 NVM. CPU0 访问 Node1 上内存的延迟为访问本地内存的 2.5 倍,带宽约为本地内存的 1/3,符合 PCM 和 DRAM 的参数,具体详见表 3. 此外,还可以利用文献[44]中使用的干扰程序来进一步降低延迟和带宽. 相较于传统的 Trace 模拟器,该种方式支持运行大规模的分布式程序.

**Table 3 Comparison of DRAM and NVM Technology [3]**

**表 3 DRAM 和 NVM 参数对比[3]**

| Parameter | DRAM | NVM |
|---|---|---|
| Read Latency/ns | 120 | 300 |
| Bandwidth/GBps | 29 | 10 |
| Capacity per CPU | 100 s of GBs | Terabytes |
| Estimated Cost | 5X | 1X |

2) 硬件和 Spark 的配置

Spark Master:

CPU X5650, 2.67 GHz, x2;

Memory 双通道,DDR3, 1 333 MHz.

Spark Slave:

CPU E7-4809 v3, 2.00 GHz, x4;

Memory 双通道,DDR 4, 1 867 MHz.

3) 测试用例

本文使用 Big Data Bench[7] 作为测试用例. 本节选用其中的 Spark PageRank 来进行性能解析. 第 5 节将选用其他类型应用进行性能测试. 为了避免 Spark 配置、RDD 不同状态带来的不公平现象. 本文将 Spark PageRank 调整到性能最优状态,调整前后的性能对比见表 4 所示. 频繁访问的 RDD 以非序列化的形式置于 DRAM 中(MEMORY_ONLY),而访问较少、用于容错的 RDD 将被设置为 MEMORY_AND_DISK_SER,来避免巨大的 GC 开销. 经过测试,在本文的大内存环境中,该配置性能处于性能最佳状态. 在后面的测试中,如无特殊标注,均使用该调优后的测试用例.

**Table 4 Optimize Spark PageRank's Performance**

**表 4 优化 Spark PageRank 性能**

| Configuration & Results | Original | Optimization |
|---|---|---|
| Physical Memory/GB | 128 | 128 |
| Executor Memory/GB | 120 | 120 |
| Elapsed Time/s | 2 084 | 1 151 |

### 4.2 内存不足时磁盘 I/O 对性能的影响

由于磁盘性能和内存、处理器的巨大差距,其常常成为整个系统的性能瓶颈. 因此,从硬件、操作系统、编程框架多个层次探索如何消除磁盘开销是研究热点. 而内存计算系统,即在编程框架层次,通过将中间计算结果缓存到内存中,减少对磁盘的

使用.同时,当服务器内存充足时,操作系统、文件系统也会利用空闲内存缓存磁盘内容,以减少磁盘 I/O 操作.

由于 DRAM 的工艺水平限制,其密度难以持续增加.因此,DRAM 容量便逐渐成为了内存计算规模扩展的瓶颈.

本节用一个较小规模的应用(memory footprint 在 100 GB 左右)运行在 32 GB 物理内存上,来展示物理内存不足时造成的问题.此时为了避免应用程序造成 swap 引起系统性能严重下降,本节将 Spark Worker 使用的内存限制在 30 GB.当内存容量和计算规模同时扩大时,只要服务器的内存容量无法满足内存计算的需求,从而造成对磁盘的访问时,该问题仍旧存在.同时,本文用 128 GB DRAM 来表示 DRAM 无限大,可以满足计算所需内存的完美情况.表 5 中的第 1、第 2 列展示了 Spark 相同配置下,只需给予充足物理内存,便可减少磁盘开销达 49%.物理内存不足时,造成性能严重下降的原因有 2 个方面:

1) 物理内存容量限制了 Java Heap 大小,本应写入内存的持久化 RDD,因内存而不足而写入磁盘;

2) 操作系统无足够可用物理内存来进行磁盘缓存操作,导致大量数据直接写入磁盘,随后从磁盘读取所需数据.

**Table 5 Spark PageRank's Performance Comparison of Before and After Utilizing Heterogeneous Memory Programming Framework**

**表 5 利用异质内存编程框架后 Spark PageRank 的性能对比**

| Configuration & Results | SMALL MEMORY+DISK | INFINITY DRAM | INFINITY DRAM | SMALL DRAM+NVM |
|---|---|---|---|---|
| Physical Memory/GB | 32 | 128 | 128 | 24 (DRAM) 104 (NVM) |
| Executor Memory/GB | 30 | 30 | 120 | 80 (On-Heap) 40 (Off-Heap) |
| Disk Cache/GB | <2 | Enough | Enough | Enough |
| Shuffle Write to Disk/GB | ≈20 | ≈20 | ≈20 | ≈20 |
| RDD to Disk/GB | 25.9 | 25.9 | ≈0 | ≈0 |
| Elapsed Time/s | 2 373 | 1 591 | 1 151 | 1 263 |

如表 5 中第 1、第 2 列所示,当 Spark Executor (Java Heap)容量不足时,有额外的 26GB 数据写入磁盘.这些数据需要在随后被读取来用于计算.此外,由于此时没有充足的空闲物理内存用于缓存磁盘数据,因此,大量 RDD 会直写入磁盘,并在随后计算时立即读取,造成了频繁的 CPU I/O 等待.图 4 中列出了物理内存不足时,磁盘 I/O read request 数目以及 I/O wait 时间随程序运行的变化趋势.从图 4 中可以看到,CPU I/O wait 时间基本和磁盘的 read request 趋势相符,频繁的磁盘 read request 造成了较严重的 CPU I/O wait.

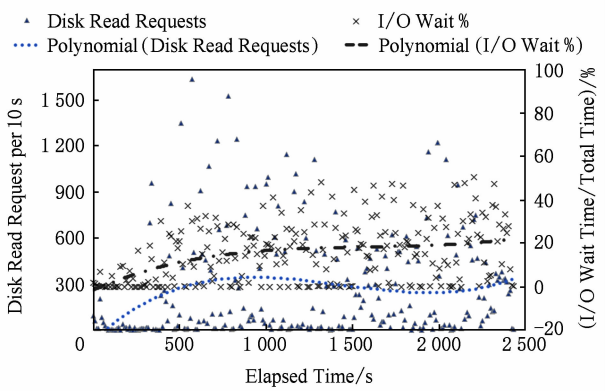


Fig. 4 Trend of disk read request number and CPU I/O wait time

图 4 磁盘读请求数量、CPU I/O 等待时间的变化趋势

在完全同样配置、充足物理内存的情况下再次运行程序,其性能如表 5 第 2 列所示.从运行时间可以看到,通过增大物理内存,虽然没有减少持久化 RDD 写入磁盘的操作,但是通过充足的内存作为磁盘缓存,可以明显地消除磁盘读、写带来的影响.经过测试,此时磁盘读取请求数量很少.

不过,由于目前针对磁盘的优化工作很多,磁盘作为性能瓶颈的情况正在逐渐减少.文献[47]通过分析一些大数据框架得出,磁盘对大数据系统造成的性能瓶颈一般在 19%左右.

本节通过对比相同 Spark 配置下充足物理内存带来的性能提升,分析了 DRAM 不足时造成 Spark 程序性能瓶颈的原因.由于 NVM 的密度为 DRAM 的 10 倍左右(表 3),因此,其在解决内存计算规模扩展性方面具有良好的潜力.本文在 4.3~4.5 节分析了如何利用高密度的 NVM 带来可观的加速比.

### 4.3 内存不足时 GC 对性能的影响

本节分析内存容量不足时造成 GC 开销过大的

原因,以及如何利用大内存恰当的消除 GC 开销.

内存计算系统中,GC 开销主要由 2 部分组成:

1) DRAM 容量无法满足内存计算规模,限制了 Java Heap 大小,造成频繁的 GC.

2)大量长期存活的数据被缓存到 Java Heap,每次 GC 均需要扫描众多的数据.

由于数据的快速增长,内存计算对内存容量的需求也快速增长,而 DRAM 已经难以满足该需求.因此,相对于处理的数据,内存的容量不足会导致 Java Heap 频繁的 GC.

每一次 GC 都需要分析当前 Java Heap 中存活的 Object,并将其移动到旧生代.而由于大数据应用使用的内存容量十分巨大,存活的 object 数目众多,因此每一次 GC 都会造成严重的开销.在我们的实验中,当 Java Heap 达到 120GB 时,单次 Full GC 的开销甚至可以达到 128s.此外,由于 Spark 会将大量的 RDD 数据缓存于内存中,这些数据长期存活,给每一次 GC 都带来了额外的分析、移动开销.所以,在很多大数据应用中,GC 开销甚至可以占到程序运行时间的 50%[5-6].

因此,降低 GC 开销是内存计算的一个研究热点.该部分工作可以划分为 2 类:修改 GC 策略来匹配大数据系统数据特点;修改内存计算框架来适应 GC 策略.例如,文献[5]认为大量的数据存活于一次迭代计算中,在计算结束时同时被回收.在结束前的 GC 行为是无效的,只会造成额外的开销.因此其根据大数据的计算行为,重新调整了 GC 的发生位置来降低 GC 频率.

经过我们分析,Spark Execution Memory 部分基本遵循文献[5]提出的假设.此时,只需直接扩大 Java Heap 便可以有效地减少 GC 的频率,也便因此减少了无效 GC 的发生.然而,Spark Storage Memory 并不遵循生代假设.以 Spark PageRank 为例,大量持久化的 RDD 将会存活很长时间,对该部分内容的 GC,将会造成大量的无用开销.

Spark Tungsten 基于 sun.misc.Unsafe 接口提出了将部分 Storage Memory 中的数据移出 Java Heap 的机制.即将性能不关键的持久化 RDD 置于 Off Heap,避免 GC 对其的扫描开销.本文基于 Off Heap 开发的面向异质内存的编程框架,也是考虑到利用 NVM 和 DRAM 接近的性能,将访问不频繁的并且长期存活的 RDD 直接至于 NVM Off Heap,来降低 GC 开销.

表 5 中第 3 列展示了充足的物理内存,并设置大 Java Heap 时,通过降低 GC 频度带来的性能提升.相比于 Java Heap 较小的第 2 列数据,加速比达到了 38%.图 5 为展示了将第 3 列(大内存、大 Java Heap)消除的时间分散到磁盘 I/O 开销、GC 开销后的比例.将表 5 中第 1 列数据作为基准,第 2 列数据作为通过超大物理内存消除磁盘 I/O 开销的结果,将第 3 列作为同时消除磁盘和 GC 开销的结果.由此可以看到充足物理内存、大 Java Heap 带来的明显加速比.

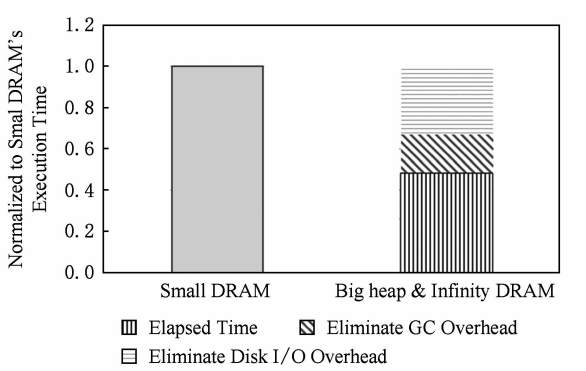


Fig. 5 Decomposition of performance improvement caused by enough physical memory and big Java Heap

图 5 充足内存带来性能提升的分解

### 4.4 异质内存编程框架设计原则解析

通过 4.2 节、4.3 节的分析可以看到内存不足时对 Spark 程序性能造成了严重的影响.本节以 Spark 中的内存管理模型:Storage Memory, Execution Memory 两个方面为切入口来阐述本文提出的异质内存编程框架如何利用 NVM 从 Java Heap, Off Heap 两个方面来扩展容量,降低 GC 开销、磁盘 I/O 开销.

#### 4.4.1 针对 Spark Storage Memory 的管理

Spark Storage Memory 不但会因为内存容量不足导致巨大的磁盘操作,即使内存充足时,其中大量的数据长期存活于 Java Heap,仍旧会造成显著地 GC 开销.除此之外,当 Storage Memory 没有空间来存储新的持久化 RDD 时,Spark 会将旧的 RDD 序列化后写入磁盘,这无疑会增加很大比例的 CPU 额外计算开销.因此,本节讨论如何利用 NVM 来消除 DRAM 容量不足时造成的磁盘 I/O 开销、GC 开销,以及 Spark 数据使用特点造成的 GC 开销.

HotSpot 中的 Parallel GC 需要扫描整个 Java Heap 中的存活数据.而 Storage Memory 中的众多

用于加速计算、容错恢复的数据会长期在内存中存活.因此,GC 对其中的数据扫描将会造成无谓的开销.在大规模的内存计算中,Storage Memory 默认会占据整个 Java Heap 50%的空间.然而和 Execution Memory 受限于 Java Heap 大小不同,持久化的 RDD 将会随着程序的执行而逐渐增加,直至将无法置于 Java Heap 的数据序列化后写入到磁盘.虽然可以通过将中间数据暂存于内存的方式显著地消除了磁盘开销,但是 GC 开销甚至可以达到程序运行时间的 50%.如图 6 所示为程序执行过程中 Java Heap 中数据的变化趋势.可以看到,当将 RDD 以非序列化形式(MEMORY_ONLY)储存于内存时,随着执行,Java Heap 中存活的数据稳步提升.对 120 GB Java Heap,伴有大量存活的、非序列化 RDD 存在时,一次 Full GC 甚至消耗了 128 s 的时间.

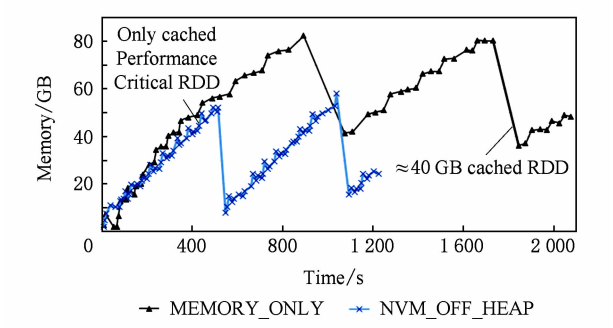


Fig. 6 Alive memory volume after each GC

图 6 每次 GC 后 Java Heap 中存活的数据

为了消除 Storage Memory 造成的 GC 开销,将一些访问不频繁的数据序列化,然后再储存在内存中.此时,一个原为众多独立对象构成的 RDD 变为了一个单一的 byte array,极大地减少了需要 GC 的对象数量,减少了单次 GC 的开销.同时由于压缩了 Storage Memory 所占用的内存空间,也减少了 GC 频率.

因为 Storage Memory 中的很多数据访问并不频繁,可以利用 NVM 来对其进行存储.Spark Tungston 利用 Unsafe 接口,开发了 Spark Off Heap 机制.即将使用不频繁的持久化 RDD 序列化后存储于 Off Heap,此时只有一些访问非常频繁的数据仍在 Java Heap 中.本文便利用该接口开发了管理 Storage Memory 的 NVM API(NVM_OFF_HEAP).图 6 展示了将大量持久化 RDD 置于 Off Heap 后,Java Heap 中数据的变化趋势.此时可看到,由 Execution Memory 数据主导的 Java Heap 会被 GC 频繁的清空,极大地减少了长时间存活、访问不频繁的数据.从而在单次 GC 开销、GC 频率两方面缓解了 GC 开销.此时,虽然对数据的序列化需要造成计算开销,但是对于 100GB+的大规模计算来说,GC 开销占据了主导.如图 7 所示,将访问不频繁的 RDD 序列化后,程序加速了 29%.通过 Off Heap 将序列化的 Storage Memory 数据置于 NVM 时,性能仍旧提升了 23%.

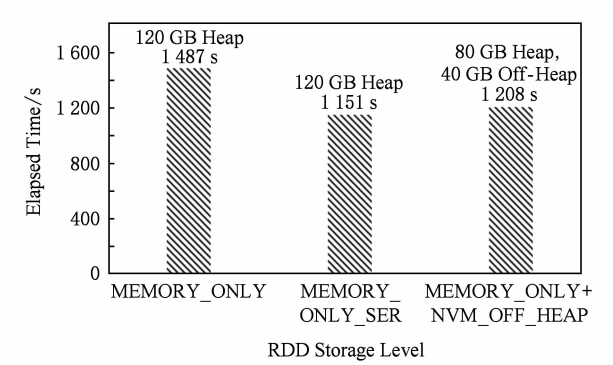


Fig. 7 Performance effect of placing storage memory on NVM

图 7 使用 NVM 储存 Storage Memory 数据后性能

图 7 展示了在内存中以非序列形式缓存 RDD;序列化存储 RDD 并置于 Java Heap 中;序列化 RDD 置于 NVM Off Heap 的性能对比.可以看到,使用 NVM 存储大量 Storage Memory 数据后(~33 GB),性能并没有明显下降(5%以内).因此,Spark 应用开发人员只需要将常规的 DISK_ONLY, MEMORY_AND_DISK, MEMORY_AND_DISK_SER 等带"DISK"、"SER"关键字的 Storage Level 直接替换为本文扩展出的 NVM_OFF_HEAP 便可利用廉价、低功耗的 NVM 达到接近无限 DRAM 的速度.此时 NVM 的使用,不但缓解了 DRAM 密度难以跟随计算规模扩展,必须使用磁盘带来的开销;也减少了价格昂贵、功耗过高的 DRAM 的使用.

将大量序列后的 Storage Memory 数据从 DRAM 移动到 NVM 后性能并没有显著下降的原因在于,这些数据本身并不会被频繁的访问;同时序列化操作需要耗费一定的 CPU 计算资源,进一步缓解了对内存带宽上限的需求.在储存 33 GB 左右的 Storage Memory 数据于 NVM 时,其读带宽峰值仅为 535 MBps 左右,如图 8 所示.但是,通过 4.2 节的分析可知,此时若不使用 NVM 而是将数据直接置于磁盘,则会带来显著的性能下降,因为 NVM 的访存延迟远远高于磁盘.

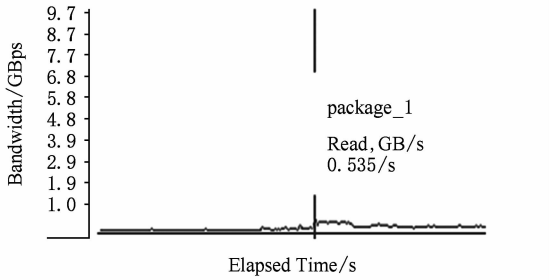


Fig. 8 NVM read bandwidth of placing partial storage memory on NVM

图 8 部分 Storage Memory 置于 NVM 后 NVM 读带宽

4.4.2 针对 Spark Execution Memory 的管理

Spark Execution Memory 仍旧使用了巨量的内存(~80 GB),同时 Execution Memory 只能使用 Java Heap 中的内存. 根据我们在 4.3 节的分析可知,即使在物理内存充足时,过小的 Java Heap 仍旧会因为频繁的 GC 而造成性能显著下降. 因此需要利用 NVM 接近 DRAM 的性能来扩展 Java Heap,以保证 Execution Memory 有充足的内存使用. 但在该过程中,我们发现 Execution Memory 中的数据造成了频繁的内存访问. 如图 9 所示,仅将 Storage Memory 中的大部分数据迁移到 NVM 后,DRAM 端的 Read Bandwidth 带宽峰值可达 15.4 GBps,此时远超了 NVM 的带宽. 因此,如果直接 NVM 替代 DRAM 来存储 Java Heap 中的数据,会导致性能下降达 55%左右;使用 DRAM 和 NVM 组成的异质内存,当程序随机使用二者时,也会有 32%的性能下降,如图 10 所示. 所以,需要合理布局其中的冷热数据,减少对 NVM 的读、写,防止因为 NVM 性能不如 DRAM 而导致新的性能瓶颈.

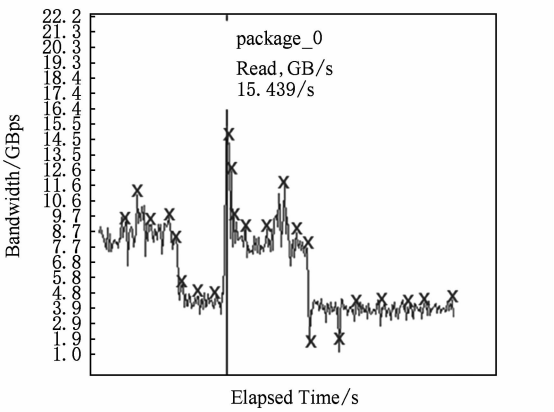


Fig. 9 DRAM read bandwidth of placing partial storage memory on NVM

图 9 部分 Storage Memory 置于 NVM 后 DRAM 读带宽

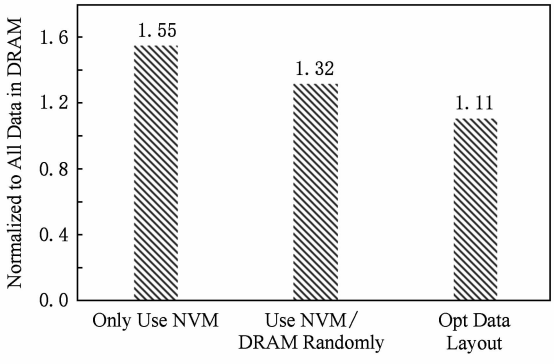


Fig. 10 Performance effect of utilizing heterogeneous memory programming framework

图 10 使用异质内存编程框架后的性能

由于 Execution Memory 中的数据均在 Java Heap 中,因此其访存特点受到 Java Heap 的管理特点影响. 本文根据文献[44]中的研究,进一步对大规模的分布式系统的访问行为进行了分析. 发现在运行 Spark 程序时,其 Heap 中的冷、热区域划分仍旧基本符合文献[44]中的结论. 不同的是,MetaData Space(元数据区)由于远大于 cache 大小,也会造成可观的访存行为. 因此,该框架将新生代和元数据区直接映射到 DRAM 区域,如图 2 所示. 同时,为了限制使用过多的 DRAM,该框架会根据 DRAM 可用容量限制新生代的大小,使其可被 DRAM 容纳. 此时虽然会造成更频繁的 minor GC(新生代 GC),但相比于将部分新生代至于 NVM 造成的带宽瓶颈,仍旧会带来较好的性能.

经过针对 Storage Memory, Execution Memory 中数据的布局,在仅使用 20%的 DRAM 时,可以达到全部使用 DRAM 90%的性能,如图 10 所示. 此时,根据表 3 中列出的 DRAM, NVM 价格比,此时的"性能/价格"比例是全部使用 DRAM 的 2.9 倍. 更为重要的是,和 DRAM 不足必须使用磁盘时的性能对比,该框架将性能提高了 88%,即通过使用该编程框架来利用 NVM 有效缓解了因为 DRAM 容量难以提升而限制内存计算规模扩展的问题.

### 4.5 Spark 编程框架对更大规模的计算是否有效

在 4.2~4.4 节的分析中,实验规模单节点可达 128 GB,我们将在本节讨论如果计算规模持续增加到 TB 级别,该框架是否继续有效.

当通过多节点扩展计算规模时,虽然整体加速比可能有所变化,但是单个节点的计算行为仍和 4.2~4.4 节分析的相似. 除此之外,多节点间通信可能造成一些新的问题. 但是,根据文献[47]的分析,多节点间的数据传输并不是大数据计算的显著

瓶颈,其结论为通过优化网络,预期可以得到的加速比仅为 2%左右. 大数据计算的主要瓶颈仍旧在计算、访存、磁盘等方面. 因此,可以认为,上述对磁盘、GC、访存行为的优化,在众多节点时仍预计可以取得较好性能.

当单节点的计算规模继续增大时. 限制该编程框架的性能因素主要来自于,NVM 是否会因为性能较差和磁盘一样再次成为限制 CPU 计算的瓶颈. 本节通过 NVM 访存延迟较大、带宽较小 2 个方面来进行讨论.

### 4.5.1 NVM 高延迟的影响

当程序的访存不规律时,每次 cache miss 都可能都会造成单独的访存,同时也无法通过硬件、软件预取来降低访存开销. 因此,可以认为,最差情况下单位时间 NVM 内访存数目的增加便会造成性能成比例的下降. 因此,本节统计每执行千条指令时 NVM 访存数目是否随着计算规模增长而迅速增加. 图 11 展示了每千条指令中的 NVM Load Miss,和 Total Load Miss 随着计算规模增长的变化趋势. 可以看到,随着 Java Heap 和输入集大小的快速增加(指数级),NVM 上的每千条指令中读请求数目仅是轻微增加. NVM 之上的读请求和总的读请求比例基本保持一个恒定值. 根据分析,从 20 GB Java Heap 增加到 40 GB Java Heap 时,NVM 读请求的比例增加可能是由于随着输入集的增大,Cache 逐渐失效的原因. 随后规模的计算中,Total Load Miss/K-Ins,NVM Load Miss/K-Ins 逐渐稳定. 因此,有理由相信,即使计算规模继续指数增大,也不会因为 NVM 的较高访问延迟造成显著的性能瓶颈.

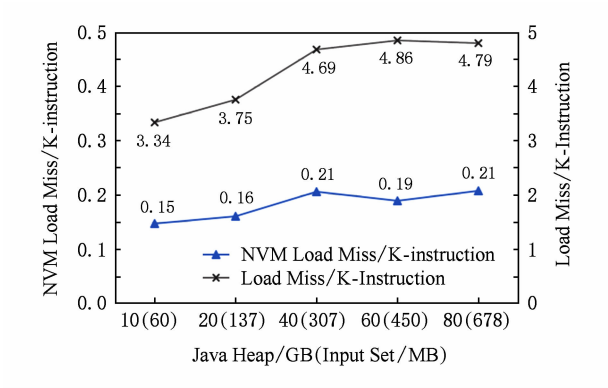


Fig. 11 NVM access frequency

图 11 NVM 访问频度

### 4.5.2 NVM 低带宽的影响

根据目前的资料,NVM 的带宽较 DRAM 较小,但是有的研究认为,NVM 的带宽可以通过工艺的

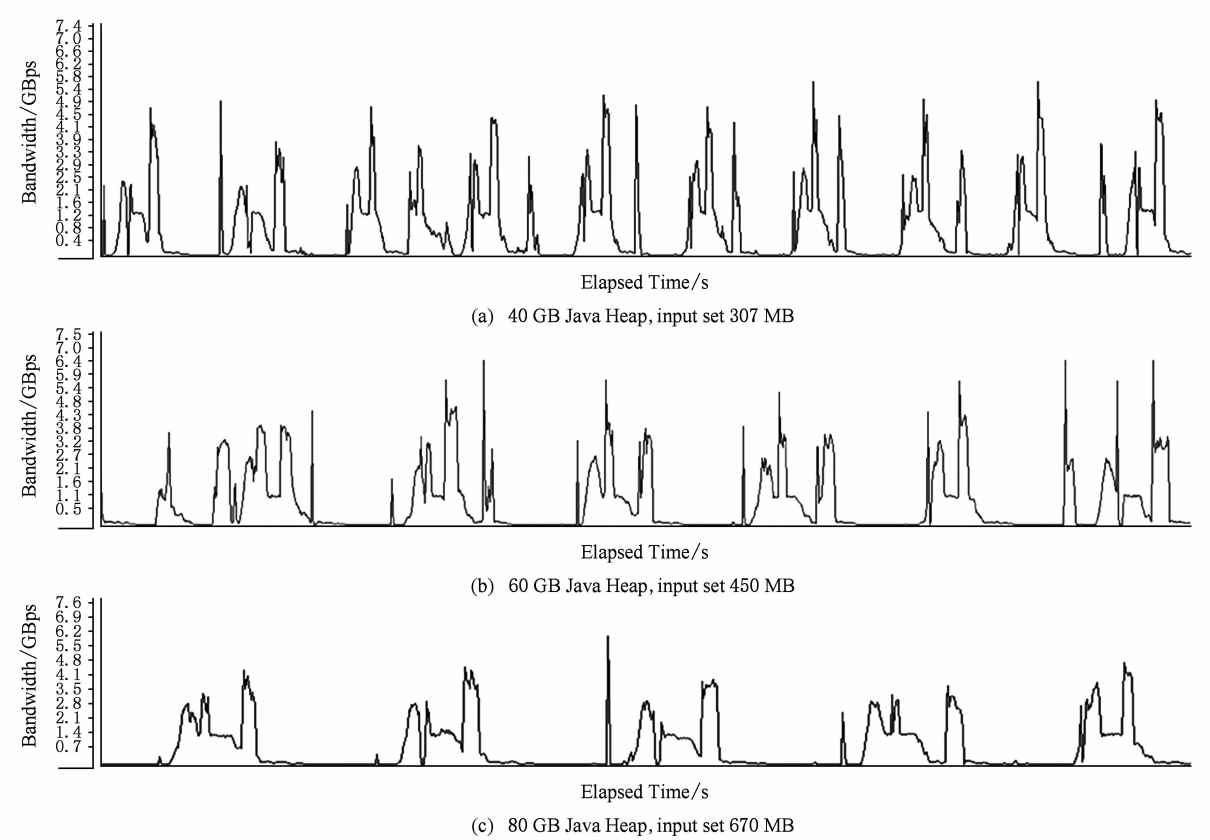


Fig. 12 NVM memory read bandwidth variation on different input data size and Java Heap size

图 12 NVM 读请求访存带宽在不同输入集和 Java Heap Size 下的变化

提升最终达到 DRAM 的水平. 本文假设 NVM 带宽为一个较小值,仅为当前 DRAM 带宽峰值的 1/6 左右[2-3](DRAM 1 867 MHz,4 通道,~60 GBps),为 10 GBps. 如图 12 中为 Java Heap 在 40GB/60GB/80GB 时 NVM 中读请求带宽趋势的对比.

从图 12 中可以看到,带宽峰值都在 7 GBps 左右,没有随计算规模增大而迅速增大的趋势. 我们认为,这是因为在一定的计算行为下,访存带宽受到了 CPU 计算能力的限制. 在现阶段,单 CPU 的计算能力很难维持摩尔定律成指数增长. 因此,在单个节点上,NVM 10 GBps 左右的带宽,已经可以满足内存计算规模的继续增加. 如果随着工艺发展,NVM 的带宽继续增加,那么其带宽可能将不再构成内存计算的性能瓶颈.

综上分析,我们认为本文提出的面向异质内存的 Spark 计算框架具有良好的扩展性. 可以解决 DRAM 容量无法满足内存计算需求的问题.

### 4.6 如何平衡序列化开销和 GC 开销

将内存中的数据序列化虽然会减少 GC 开销和磁盘 I/O 开销,但是会给 CPU 带来额外的计算压力. 当内存容量足够、GC 较少发生、磁盘 I/O 开销较少时,序列化并不会带来整体的性能提升. 因此如何平衡二者,需要根据具体的应用特点进行具体分析. 同时,对于一些程序,其对 NVM 的使用主要通过扩展 Java Heap 来减少 GC 开销. 经 Off Heap,序列化写入 NVM 的数据相对较少,因此使用本文的异质内存编程框架后,并不会带来过多的序列化开销.

根据目前的研究表明,对于内存使用量较大的内存计算应用来说,由于其需要处理巨量的、长期存活的 object,GC 开销甚至占据了大数据应用运行时间的 50%以上[5-6],是限制其性能的主要瓶颈. 而内存计算应用由于需要将一些中间结果缓存于内存中,进一步加剧了 GC 开销. 因此华中科技大学[38]、Spark Tungston[34] 甚至分别提出了将大量数据序列化后压缩为 byte array,并直接在其上进行计算的方式来减少 Java Heap 中 object 数量,以便减少 GC 开销,提升应用整体性能.

针对该问题,我们在 128 GB 内存规模上进行了实际的实验和分析. 对于 Spark PageRank 程序,当将绝大多数 RDD 改变为序列化方式缓存时(从 MEMORY_ONLY 改为 MEMORY_ONLY_SER),如图 13 所示,程序的整体性能提升了 29%,即使将序列化后的数据置于 NVM Off Heap,也会带来 23%的性能提升,如图 7 所示.

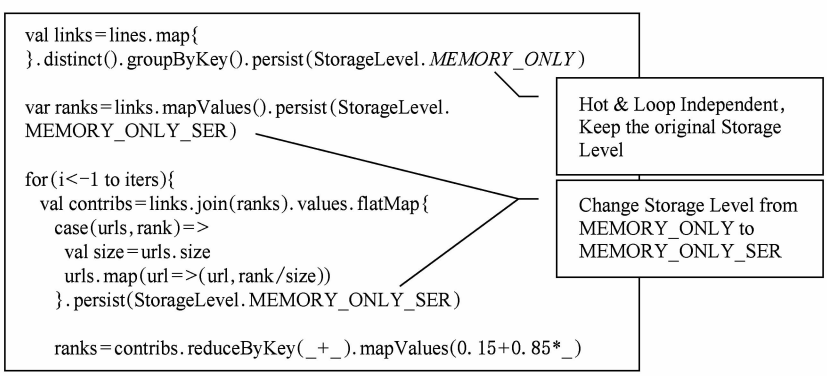


Fig. 13 Change PageRank's RDD storage status to serialization mode

图 13 将 PageRank 中的 RDD 改为序列化形式缓存

这是由于,GC 过程中,其需要按照 object 引用关系来寻找所有存活的 object,而大数据应用中存活的 object 数量巨大,每次 GC 均会造成显著的 GC 开销. 而只需要经过一次序列化,大量的 object 被压缩为单一的 byte array,减少了每次 GC 的开销;而经过序列化压缩后,数据大小显著降低,便减少了 GC 的频率;最后,GC 需要将长期存活的 object 在 eden,from,to,old 区域之间移动. 而序列化显著地压缩了这些长期存活的 RDD 数据大小,减少了数据移动、拷贝带来的开销.

因此,对于大数据应用、内存计算应用该类消耗内存数量巨大、产生的众多 object 的应用来说,当确定 GC 为其主要开销后,将数据序列化虽然会给 CPU 带来一定的压力,但是会减少 GC 带来的开销;同时,对数据压缩也会减少磁盘 I/O 的开销;因此,其会显著地提升程序的性能.

此外,对于一些程序,在使用本文的异质内存编程框架后,其通过 Off Heap 写入 NVM 的数据并不多. 如第 5 节中的 GraphX 程序,由于其缓存的 RDD 较少,因此在使用本文的框架后并没有太多的数据

写入到 Off Heap 中.此时 NVM 主要用来对 Java Heap 进行扩展,减少 GC 的频度和磁盘 I/O 的使用.

最后,开发人员可以通过 OpenJDK 输出的 GC Log 信息快速判断 GC 造成的开销比例.当 DRAM 内存充足,GC、磁盘 I/O 并非程序主要开销时,开发人员只需要不调用我们提供的 API,便不会带来额外的序列化开销.此时,仍可以通过命令 -XyoungGenDRAM 来开启 OpenJDK 层次的数据自动布局.

## 5 编程框架的多用例性能测试

在第 4 节的分析中使用了搜索引擎离线分析程序,Spark PageRank,进行性能分析.本节将分析在不同的 Spark 应用中,提出编程框架是否仍然有效.

### 5.1 Spark-GraphX 图计算应用

首先,针对图计算 Spark-GraphX 进行性能分析.这里选用其中的 ConnectedComponents 算法来进行性能分析.其亦为迭代计算,不同于 Spark PageRank,Spark-GraphX 计算过程中,其仅仅保留上一次迭代的持久化 RDD,并将前面的持久化 RDD 释放掉.因此,这里并没有太多持久化的 RDD 在 Java Heap 中,没有必要使用 NVM_OFF_HEAP 接口.为了使应用的 Memory Footprint 达到 120 GB,这里选用了更大规模的输入集:3 GB 大小.

在测试中发现,仅使用 32GB 物理内存时,应用运行时间过长,因此将其扩展为 60 GB,并将 Executor 限制为 40 GB.即使该配置下,Connected Components 消耗的时间也过长,这里并没有最终执行完.由于并没有 OFF_HEAP 空间的使用,因此异质内存的对比试验设置为 30 GB DRAM+98 GB NVM,Executor (Java Heap)直接设置为 120 GB.运行结果如图 14 所示.可以看到,此时虽然 Spark-GraphX 没有大量的使用 Storage Memory,但是本文提出的框架在用 25%的 DRAM 情况下,性能比全部使用 DRAM 仅慢了 9%.而且比 DRAM 不足(仅为 40 GB DRAM)时,加速可达 300%以上.由于利用 Spark-GraphX 的应用的访存行为较为类似,因此本节在这里仅测试了 ConnectedComponents.从此可以看到,即使 Spark 应用为 Storage Memory 使用较少,NVM 主要用于扩展 Java Heap 时,本文提出的面向异质内存的 Spark 编程框架仍旧可以取得很好地效果.

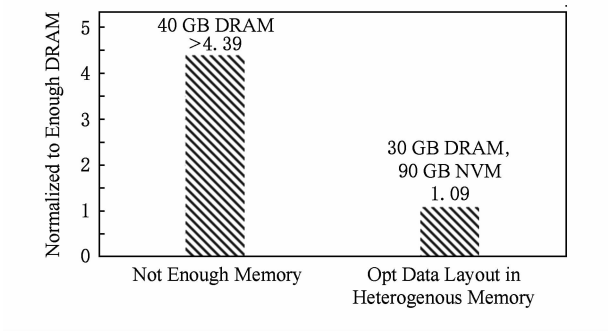


Fig. 14 Spark-GraphX performance comparison of utilizing enough DRAM, not enough DRAM and heterogeneous memory

图 14 Spark-GraphX 在充足 DRAM、DRAM 不足、异质内存情况下的性能对比

### 5.2 Spark-SocialNetwork: *K*-means 应用

本节使用 Big DataBench 中的 *K*-means 进行性能分析,其同样为迭代计算.其中的持久化 RDD 分为 2 类,直接以非序列化形式置于 Java Heap 中的频繁访问 RDD.Storage Level 为 MEMORY_AND_DISK 的,可以存放于磁盘的 RDD.在异质内存中,本节将后者替换为 NVM_OFF_HEAP 来进行试验.但是经过分析发现,其产生的 Storage memory 仍然明显少于 Spark PageRank.实验结果如图 15 所示,在使用 25%的 DRAM 后,其性能仅仅降低了 6%,比 DRAM 容量不足时加速 52%.

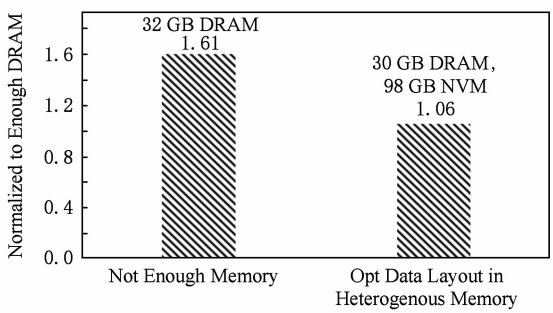


Fig. 15 Spark *K*-means performance comparison of utilizing enough DRAM, not enough DRAM and heterogeneous memory

图 15 Spark *K*-means 在充足 DRAM、DRAM 不足、异质内存情况下的性能对比

经过本节的分析可以看到,异质内存编程框架在 Spark,OpenJDK 两个层次进行的异质内存数据布局,在搜索引擎离线分析(Spark PageRank)、图计算(Spark GraphX)、社交网络分析(Spark *K*-means)领域具有较好的结果.这是由于,在 Spark 层次,由于 Spark 程序的主要数据结构(RDD)简洁、计算行为

规则,Spark 应用开发人员较易识别出冷热数据,并调用本文提供的接口来将特定的 RDD 置于 NVM 中,降低 GC 开销;在 OpenJDK 层次,由于 Spark 运行于 JVM,因此这些应用的访存特性均受到了 OpenJDK 对内存管理的影响. 因此本文通过分析 Java Heap 内存管理特点给出的数据映射方案对这些应用均具有一定的效果. 将热数据置于 DRAM、冷数据置于 NVM 后,并没有因为 NVM 的性能不如 DRAM 而造成较大的性能下降. 同时,利用 NVM 扩展 Java Heap 容量后,显著地降低了 GC 的频率. 由于时间限制,我们没能测试更多的用例,在随后的工作中我们会扩大实验对象,进一步分析方法的局限性.

## 6 总结与展望

本文提出的框架可以很好的处理 Spark 离线分析应用,尤其是对于迭代形式的 Spark 程序具有很好的效果. 在仅使用 20%~25%的 DRAM 时,相比于全部使用 DRAM,性能仅下降 10%左右. 对比 DRAM 不够频繁访问磁盘时的情况,使用 NVM 扩展内存容量后,性能加速可达 88%. 但对于一些在线处理的应用,比如在线查找等,其需要快速返回结果给用户,该框架可能导致一定比例的用户得到结果的延迟较大,从而造成较差的使用体验. 对于该类应用,需要动态地对数据进行迁移来进一步提升性能. 我们将在随后的工作中继续探讨如何解决该类问题.

## 参 考 文 献


[1] Grandpierre M, Buss G, Esser R. In-Memory Computing technology. The holy grail of analytics? [R]. Berlin: Deloitte, 2013

[2] Zilberberg O, Weiss S, Toledo S. Phase-change memory: An architectural perspective [J]. ACM Computing Surveys, 2013, 45(3): 1-33

[3] Dulloor S R, Roy A, Zhao zheguang, et al. Data tiering in heterogeneous memory systems [C] //Proc of the 11th European Conf on Computer Systems. New York: ACM, 2016: No. 15

[4] Apache Spark. A fast and general engine for large-scale data processing [EB/OL]. [2017-09-10]. http://spark. apache. org/

[5] Nguyen K, Lü Fang, Xu Guoqing, et al. Yak: A high-performance big-data-friendly garbage collector [C] //Proc of the 12th USENIX Conf on Operating Systems Design and Implementation. Berkeley, CA: USENIX, 2016: 349-365

[6] Nguyen K, Wang Kai, Bu Yingyi, et al. FACADE: A compiler and runtime for (Almost) object-bounded big data applications [C] //Proc of the 12th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2015: 675-690

[7] BigDataBench. A Big Data Benchmark Suite, ICT, Chinese Academy of Sciences [EB/OL]. [2017-09-10]. http://prof. ict. ac. cn/

[8] Wei Wei, Jiang Dejun, Sally A M, et al. Exploiting program semantics to place data in hybrid memory [C] //Proc of the Int Conf on Parallel Architecture and Compilation. Piscataway, NJ: IEEE, 2015: 163-173

[9] Apache Hadoop. Open-source software for reliable, scalable, distributed computing [EB/OL]. [2017-09-10]. http://hadoop. apache. org/

[10] Lu Youyou, Shu Jiwu. Survey on flash-based storage systems [J]. Journal of Computer Research and Development, 2013, 50(1): 49-59 (in Chinese)
(陆游游, 舒继武. 闪存存储系统综述[J]. 计算机研究与发展, 2013, 50(1): 49-59)

[11] Shu Jiwu, Lu Youyou, Zhang Jiacheng, et al. Research progress on non-volatile memory based storage system [J]. Science & Technology Review, 2016, 34(14): 86-94 (in Chinese)
(舒继武, 陆游游, 张佳程, 等. 基于非易失性存储器的存储系统技术研究进展[J]. 科技导报, 2016, 34(14): 86-94)

[12] Xia Fei, Jiang Dejun, Xiong jin. Evaluating and analyzing the performance of nonvolatile memory system [J]. Journal of Computer Research and Development, 2014, 51(Suppl 1): 25-31 (in Chinese)
(夏飞, 蒋德钧, 熊劲. 影响非易失性内存系统性能的因素分析[J]. 计算机研究与发展, 2014, 51(增 1): 25-31)

[13] Xu Yuanchao, Yan Junfeng, Wan Hu, et al. A survey on security and privacy of emerging non-volatile memory [J]. Journal of Computer Research and Development, 2016, 53(9): 1930-1942 (in Chinese)
(徐远超, 闫俊峰, 万虎, 等. 新型非易失存储的安全与隐私问题研究综述[J]. 计算机研究与发展, 2016, 53(9): 1930-1942)

[14] Zhang Tiefei, Chen Tianzhou, Wu Jianzhong. Exploiting memory access patterns of programs for energy-efficient memory system techniques [J]. Journal of Software, 2014, 25(2): 254-266
(章铁飞, 陈天洲, 吴剑钟. 基于程序访存模式的低功耗存储技术[J]. 软件学报, 2014, 25(2): 254-266)

[15] Qureshi M K, Srinivasan V, Rivers J. Scalable high performance main memory system using phase-change memory technology [C] //Proc of the 36th Annual Int Symp on Computer Architecture. New York: ACM, 2009: 24-33

[16] Zhou Ping, Zhao Bo, Yang Jun, et al. A durable and energy efficient main memory using phase change memory technology [C] //Proc of the 36th Annual Int Symp on Computer Architecture. New York: ACM, 2009: 14-23

[17] Kgil T, Roberts D, Mudge T. Improving NAND flash based disk caches [C] //Proc of the 35th Annual Int Symp on Computer Architecture. Piscataway, NJ: IEEE, 2008: 327-338

[18] Mangalagiri P, Sarpatwari K, Yanamandra A, et al. A low-power phase change memory based hybrid cache architecture [C] //Proc of the 18th ACM Great Lakes Symp on VLSI. New York: ACM, 2008: 395-398

[19] Lee B C, Ipek E, Mutlu O, et al. Architecting phase change memory as a scalable dram alternative [C] //Proc of the 36th Annual Int Symp on Computer Architecture. New York: ACM, 2009: 2-13

[20] Ramos L, Gorbatov E, Bianchini R. Page placement in hybrid memory systems [C] //Proc of the Int Conf on Supercomputing. New York: ACM, 2011: 85-95

[21] Lebeck A, Fan Xiaobo, Zeng Heng, et al. Power aware page allocation [J]. ACM SIGPLAN Notices, 2000, 35(11): 105-116

[22] Huang Hai, Pillai P, Shin K. Design and implementation of power-aware virtual memory [C]//Proc of the Conf on USENIX Annual Technical Conf. Berkeley, CA: USENIX, 2003: 5-5

[23] Zhang Wangyuan, Li Tao. Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures [C]//Proc of the 18th Int Conf on Parallel Architectures and Compilation Techniques. Piscataway, NJ: IEEE, 2009: 101-112

[24] Pandey V, Jiang W, Zhou Y, et al. DMA-aware memory energy management [C]//Proc of the 12th Int Symp on High-Performance Computer Architecture. Piscataway, NJ: IEEE, 2006: 133-144

[25] Sivathanu M, Prabhakaran V, Popovici F, et al. Semantically-smart disk systems [C] //Proc of the 2nd USENIX Conf on File and Storage Technologies. Berkeley, CA: USENIX, 2003: 73-88

[26] Mesnier M, Akers J. Differentiated storage services [J]. ACM SIGOPS Operating Systems Review, 2011, 45(1): 45-53

[27] Hassan A, Vandierendonck H, Nikolopoulos D. Software-managed energy-efficient hybrid DRAM/NVM main memory [C] //Proc of the 12th ACM Int Conf on Computing Frontiers. New York: ACM, 2015: No. 23

[28] Piccoli G, Santos H, Rodrigues R, et al. Compiler support for selective page migration in NUMA architectures [C]// Proc of the Int Conf on Parallel Architectures and Compilation. New York: ACM, 2014: 369-380

[29] Dulloor S, Kumar S, Keshavamurthy A, et al. System software for persistent memory [C] //Proc of the 9th European Conf on Computer Systems. New York: ACM, 2014: No. 15

[30] Huang Hai, Shin K, Lefurgy C, et al. Improving energy efficiency by making DRAM less randomly accessed [C] // Proc of the 2005 Int Symp on Low power electronics and design. New York: ACM, 2005: 393-398

[31] Appel A W. Simple generational garbage collection and fast allocation [J]. Software Practice and Experience, 1989, 19(2): 171-183

[32] StefanoviC D, McKinley Kand J, Moss J. Age-based garbage collection [C] //Proc of the 14th ACM SIGPLAN Conf on Object-Oriented Programming, Systems, Languages, and Applications. New York: ACM, 1999: 370-381

[33] Bu Yingyi, Borkar V, Xu Guoqing, et al. A bloat-aware design for big data applications [C] //Proc of the 2013 Int Symp on Memory Management. New York: ACM, 2013: 119-130

[34] Databricks. Project Tungsten: Bringing Apache Spark Closer to Bare Metal [EB/OL]. [2017-09-10]. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

[35] BeebeeJr W, Rinard M. An implementation of scoped memory for real-time Java [G] //LCNS 2211: Proc of Int Workshop on Embedded Software. Berlin: Springer, 2001: 289-305

[36] Blackburn S, McKinley K. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance [C] //Proc of the 29th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2008: 22-32

[37] Boyapati C, Salcianu A, Beebee W, et al. Ownership types for safe region-based memory management in real-time Java [J]. ACM SIGPLAN Notices, 2003, 38(5): 324-337

[38] Lu Lu, Shi Xuanhua, Zhou Yongluan, et al. Lifetime-based memory management for distributed data processing systems [J]. VLDB Endowment, 2016, 9(12): 936-947

[39] Gao Tiejun, Strauss K, Blackburn S, et al. Using managed runtime systems to tolerate holes in wearable memories [C] //Proc of the 34th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2013: 297-308

[40] Nakagawa G, Oikawa S. An analysis of the relationship between a write access reduction method for NVM/DRAM hybrid memory with programming language runtime support and execution policies of garbage collection [C] //Proc of the IIAI 3rd Int Conf on Advanced Applied Informatics. Piscataway, NJ: IEEE, 2014: 597-603

[41] Nakagawa G, Oikawa S. Language runtime support for NVM/DRAM hybrid main memory [C] //Proc of the IEEE COOL Chips XVII. Piscataway, NJ: IEEE, 2014: 1-3

[42] Nakagawa G, Oikawa S. NVM/DRAM hybrid memory management with language runtime support via MRW queue [C] //Proc of the Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). Piscataway, NJ: IEEE, 2015: 1-6

[43] Nakagawa G, Oikawa S. Preliminary analysis of a write reduction method for non-volatile main memory on jikes RVM [C] //Proc of the 1st Int Symp on Computing and Networking. Piscataway, NJ: IEEE, 2013: 597-601

[44] Wang Chenxi, Cao Ting, Zigman J, et al. Efficient management for hybrid memory in managed language runtime [G] //LNCS 9966: Proc of the IFIP Int Conf on Network and Parallel Computing. Berlin: Springer, 2016: 29-42

[45] Yang Ting, Hertz M, Berger E, et al. Automatic heap sizing: Taking real memory into account [C] //Proc of the 4th Int Symp on Memory Management. New York: ACM, 2004: 61-72

[46] Hertz M, Feng Yi, Berger E. Garbage collection without paging [C] // Proc of the Programming Language Design and Implementation. New York: ACM, 2005: 143-153

[47] Ousterhout K, Rasti R, Ratnasamy S, et al. Making sense of performance in data analytics frameworks [C] //Proc of the 12th USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX, 2015: 293-307

**Wang Chenxi**, born in 1990. PhD candidate of the University of Chinese Academy of Sciences. Student member of CCF. His main research interests include Java virtual machine, garbage collection, memory management, etc.

**Lü Fang**, born in 1975. PhD. Senior engineer at the Institute of Computing Technology, Chinese Academy of Sciences. Her main research interests include architecture-oriented performance analysis and optimization, compiler optimizations, etc.

**Cui Huimin**, born in 1979. PhD. Associate professor at the Institute of Computing Technology, Chinese Academy of Sciences. Member of CCF. Her main research interests include programming framework for heterogeneous datacenters, parallel compilation, etc.

**Cao Ting**, born in 1984. PhD. Post-doc at the Institute of Computing Technology, Chinese Academy of Sciences. Her main research interests include efficient implementation of high level languages and novel computer architectures.

**John Zigman**, born in 1967. Senior software engineer at the ACFR University of Sydney, with a PhD from the Australian National University. His main research interests include distributed computation, orthogonal persistence and virtual machine implementations, and machine learning.

**Zhuang Liangji**, born in 1992. Master candidate at the Institute of Computing Technology, Chinese Academy of Sciences. His main research interests include graph computation, big data, etc.

**Feng Xiaobing**, born in 1969. Professor and PhD supervisor at the Institute of Computing Technology, Chinese Academy of Sciences. Member of CCF. His main research interests include compiler optimization and binary translation.