中国科学院大学

University of Chinese Academy of Sciences

博士学位论文

面向托管式语言的异质内存管理和优化

作者姓名：王晨曦

指导教师：冯晓兵　研究员

中国科学院计算技术研究所

学位类别：学术型博士

学科专业：计算机系统结构

培养单位：中国科学院计算技术研究所

2018 年 6 月

中国科学院大学
University of Chinese Academy of Sciences

博士学位论文

面向托管式语言的异质内存管理和优化

作者姓名：王晨曦

指导教师：冯晓兵　研究员

中国科学院计算技术研究所

学位类别：学术型博士

学科专业：计算机系统结构

培养单位：中国科学院计算技术研究所

2018 年 6 月

Heterogeneous Memory Management and Optimization for Managed Languages

A dissertation submitted to

University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Doctor of Philosophy

in Computer Science and Technology

By

Wang Chenxi

Supervisor: Professor Feng Xiaobing

Institute of Computing Technology

Chinese Academy of Sciences

June, 2018

中国科学院大学

研究生学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。

作者签名：王晨曦

日　　期：2018.5.25

中国科学院大学

学位论文授权使用声明

本人完全了解并同意遵守中国科学院有关保存和使用学位论文的规定，即中国科学院有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：王晨曦　　导师签名：

日　　期：2018.5.25　　日　　期：2018.5.25

摘 要

随着数据中心、大数据应用的发展，DRAM (Dynamic Random-Access Memory) 技术由于工艺限制，其存储密度增速难以满足计算规模的快速增长。将存储密度大、价格低廉，但性能弱于 DRAM 的非易失性内存 (Non-Volatile Memory, NVM) 和 DRAM 构成异质内存是解决该问题的一个主流研究方向。此时，需将应用中的数据进行冷、热划分，并布局到异质内存中的 DRAM 和 NVM 部分。托管式语言，如 Java，Scala 等，被广泛应用于分布式系统、嵌入式系统等框架的开发。由于托管式运行时自身具有内存管理模块，其往往和操作系统、硬件的的异质内存管理机制策略产生冲突。

本文围绕“利用托管式运行时系统管理异质内存”的思想，来解决多系统层次同时移动数据造成的冲突问题。针对数据使用行为复杂的单机应用和计算行为规则的大数据应用，本文分别提供了基于程序分析的数据划分策略和数据布局编程接口。最终，该研究基于内存计算框架 Spark 和 Java 运行时 OpenJDK 实现了贯穿大数据应用、大数据框架、运行时系统、异质内存的一体化管理框架。

本文的创新点主要有：

- 基于运行时系统，针对单机应用提出了一种粗、细两级粒度的冷、热数据划分，并利用 GC 进行数据布局的方法。本文直接利用运行时堆现有的数据生命周期、数据类型等数据分类，进行粗粒度的冷、热数据划分；本文发现了程序中的物理访存主要由极少量函数造成的现象，并提出了基于函数访存密度 (Memory Access Density) 的细粒度冷、热数据划分方法。最终，针对 Dacapo 等程序，在异质内存中的 DRAM 比例为 15% - 25% 时，和使用与异质内存等量的 DRAM 相比，仅有 5% - 25% 的性能差。
- 本文对典型内存计算框架 (Spark) 的迭代计算应用的磁盘开销、GC、计算特征等进行了系统分析，发现其基本单元 (RDD) 之间具有显著的访存差异和生命周期差异，而单一基本单元内部的众多数据则具有类似的访存行为。基于此，本文提出了一套以自定义数据结构 (RDD) 为布局

粒度的异质内存编程接口。

- 提出了一种跨层的“大数据框架 - 运行时系统协同”的数据布局策略。通过本文开发的数据信息传递通道，应用开发者对单一自定义数据结构 (RDD) 进行位置 (DRAM、NVM) 标注后，标注信息首先在 Spark 层次中相互依赖的 RDD 之间进行传播，然后运行时层根据标注信息划分众多孤立的数据对象，并基于该数据划分进行异质内存布局和定向优化。使用本文提出的编程接口对现有 Spark 应用进行轻微修改，便可以 RDD 粒度在异质内存上进行数据布局。在使用 DRAM 比例在 25% - 35% 的异质内存时，可以达到全部使用 DRAM 时 80% - 97% 的性能。

关键词：非易失性内存；托管式语言；Java 虚拟机；大数据计算；内存计算

Abstract

The density of DRAM (Dynamic Random-Access Memory) can not meet the rapid growth of application computing scale for data center and big data applications. In order to utilize the high density of NVM (Non-Volatile Memory) and the performance of DRAM, combining the two into an integrated heterogeneous memory is a good alternative. It's important to divide the data and put them in DRAM and NVM properly. Managed languages, such as Java, Python, Scala etc., are widely used in distributed system and embedded system development. But the GC mechanism of runtime system may mess up the good data layout placed by OS and hardware.

In order to solve the conflict caused by data management in multiple system levels, this dissertation proposed to utilize the runtime system to manage the data layout for heterogeneous memory. This dissertation has proposed program analysis policies and programming interface to divide the data for PC benchmarks and big data applications separately. Finally, based on Spark and OpenJDK, we implement an integrated memory management framework that spans big data applications, big data framework, runtime system and heterogeneous memory.

The dissertation makes the following contributions:

- Based on the runtime system, this dissertation proposed a policy to manage the data layout in two different granularity. The coarse-grained hot and cold data partitioning policy is based on the existing data classification in Java heap which is split by the data life-time and data types. We find that most of the LLC Misses of a program are caused by a small number of functions. Based on this finding we proposed a fine-grained hot and cold data partitioning policy according to the memory access density of functions. For Dacapo and SPECjbb benchmarks, compared to run on DRAM, by using heterogeneous memory with 15% - 25% DRAM, the performance difference is 5% - 25%.
- Through analyzing the disk I/O overhead, GC and calculation behavior of typical iterative computing applications of Spark, we find that there are signifi-

cant differences in memory access behavior and life-time between the RDDs. But the objects within same RDD have similar memory access behavior and lief-time. Based on this finding this dissertation proposed a series of heterogeneous memory programming interfaces for application developers to control the data layout in RDD granularity.

- Proposed a cross-layer approach to manage the heterogeneous memory collaboratively by Spark and JVM. We developed a data information passage to transfer the RDD attribute annotated by application developers to objects in JVM. The annotated attribute is first spread between RDDs that are dependent on each other in the Spark hierarchy; After JVM obtains the attributes, it automatically exposes the fine-grained Java objects related to the annotated RDDs, and places these objects between DRAM and NVM using the migration-driven allocation approach. After applying our programming interfaces to existing Spark applications, the experimental results show that by using heterogeneous memory with 25% - 35% DRAM, we can achieve 80% - 97% performance of running on DRAM.

Key Words: Non-Volatile Memory; Managed Language; Java Virtual Machine; Big Data; In-Memory Computing System

目 录

图目录

表目录

第1章 引言

随着数据中心、内存数据库、内存计算应用的发展，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，并广泛部署内存计算框架[1]。随着业界对内存大容量、低功耗的需求逐渐增加[2][3][4]，服务器生产商已逐渐在单机节点堆叠了 TB 级别的 DRAM（Dynamic Random Access Memory），这不但造成了巨大的功耗开销，如 DRAM 功耗可占某些服务器的 40% 以上[5]，同时也导致了整机内存成本的增加。当前的 DRAM 技术已经接近了工艺水平的极限，存储密度、生产成本的改善难以满足计算规模的增长[6][7]。为了满足对内存大容量、低价格、低功耗日益增长的需求，新型的非易失性内存 (Non-Volatile Memory，NVM) 受到了持续的关注。然而，由于主流非易失性内存，如相变存储器 (Phase Change Memory, PCM) 的性能弱于 DRAM，并无法直接替代 DRAM。目前非易失性内存的主流应用方式为将其和 DRAM 组合为一体化的异质内存。因此，如何管理数据在异质内存上的布局是非易失内存的一个重要研究方向。

托管式语言（如 Java、C#、Scala 等）具有跨平台、面向对象、垃圾回收机制 (Garbage Collection, GC) 等特性，被广泛的应用在大数据系统开发，如 Spark[8]，Hadoop[9] 等。托管式语言的运行时系统 (Runtime System) 具有自己的内存管理模块，包括内存分配，垃圾数据自动回收等机制，这些机制会在虚拟地址空间移动数据，该行为会和操作系统、硬件的数据布局产生冲突，给异质内存的数据布局管理带来了新的挑战。

因此，研究基于托管式语言开发的应用的计算行为、数据使用特性，并据此提出一套针对托管式应用的异质内存管理方式是一个重要的研究课题。以下小节首先介绍了非易失性内存、托管式运行时以及大数据系统的相关知识，然后探讨了本文的研究动机和研究思路。

1.1 非易失内存带来的问题和机遇

非易失性内存带来的新机遇

非易失性内存 (NVM) 无动态刷新，静态功耗较低，同时其存储密度比 DRAM 大数倍[6]，单位存储成本大幅下降，相同容量的 NVM 造价仅为 DRAM 20% 左右[10]。因此，NVM 给解决大数据计算内存需求的问题带来了新的机会。除此之外，其所具有的非易失特性也让其具备了同时替代存储设备 (如 FLASH、HDD) 的潜力[11]，从而带来了进一步降低系统因频繁读写磁盘而造成性能损失的可能性。

由于 NVM 的读写性能、写寿命等均与 DRAM 有一定的差距。因此，为了达到性能、功耗、开销的平衡，同时使用 DRAM 和 NVM 构成异质内存是当今的一个主流趋势[12][13][14][15]。如图 1.1 为异质内存的两种主流构成方式，其中图 1.1(a) 为 DRAM 作为 NVM 缓存的层级结构；图 1.1(b) 为 DRAM 和 NVM 处于同一等级的平级结构。

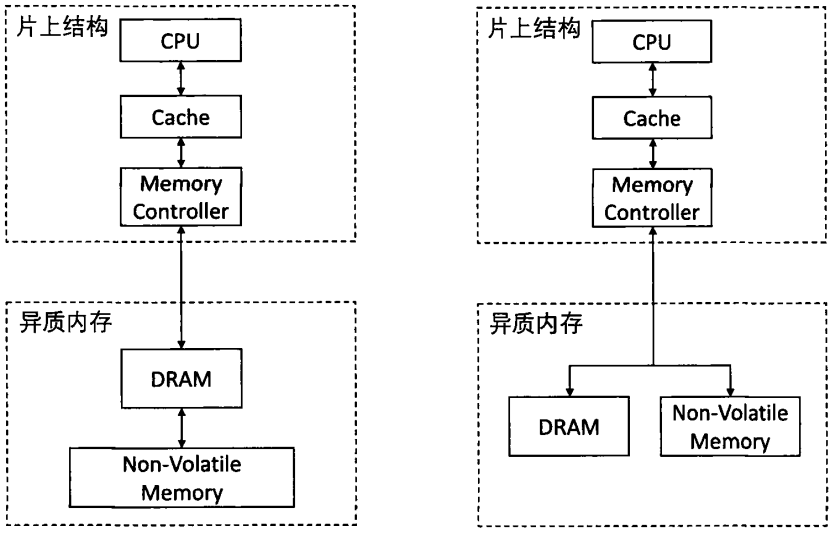
在层级结构中，CPU 无法直接访问 NVM 中的数据，DRAM 和 NVM 中的数据以 Inclusive 的形式存在。层级结构会牺牲掉 DRAM 的空间，并且需要将被访问的数据从 NVM 取到 DRAM 后，才可被 CPU 访问并用于计算。此时数据移动的方式可以是硬件的级联方式，也可以是由操作系统、编译等控制的软件方式。当使用类似于 Cache - Memory 形式的硬件管理策略时，Tag 位也会造成 DRAM 缓存容量的消耗，使用 1GB DRAM 作为 32GB PCM 的缓存，在 16 路组相联 (16-way set associativity), 256 Bytes 缓存块 (Cache line) 的配置下，Tag 等标志位的消耗就已经达到了 DRAM 容量的 13%[13]。

平级结构，即 CPU 可以同时访问 DRAM 和 NVM 的结构。此时数据以 Exclusive 的形式存在，数据在异质内存中只有一份，可以被储存在 DRAM 中或是 NVM 中，此时的数据在二者之间的布局管理以软件方式为主。如何按照应用场景划分数据，并布局到对应的 DRAM、NVM 是该种结构的主要挑战。

根据异质内存中的数据管理方式分类，常见的有硬件管理策略，硬件、操作系统结合的管理策略，以及运行时/编译/程序员管理策略，我们将在第2 章的相关工作中对此进行介绍。

非易失性内存带来的新问题

NVM 虽然具有以上优点，但其所具有的劣势亦需要在应用过程中进行特



(a) 层级结构　　(b) 平级结构

(a) Hierarchy architecture　　(b) Flat architecture

图 1.1 异质内存的两种主流结构

Figure 1.1 Two types of heterogeneous memory architecture

殊的管理。如相比于 DRAM，NVM 的写寿命较短，一些 NVM 的写寿命虽然比闪存高 1000 倍以上[6]，但仍和 DRAM 的写寿命有相当差距。因此可能由于程序中数据的空间局部性特征导致频繁写 NVM 的固定区域，致使该区域快速损坏。通过减少冗余写，Wear-Leveling 等内存管理技术，可以将 NVM 的写寿命可以延长 13 - 22 年[16]。而通过给 NVM 添加少量的 DRAM 作为缓存，并在二者之间进行合理数据迁移，减少对 NVM 的写操作，也可以将其寿命延长 3 - 9.7 年[13]。

此外，异质内存使目前单一的内存结构多层次化，引入了数据划分与布局的需求。而数据在不同层次的布局需要考虑到 NVM 的特性以及具体的应用环境需求。如 NVM 具有低于 DRAM 的读功耗和高于 DRAM 的写功耗[5][16]，因此对于功耗敏感的应用环境，将写频繁的数据迁移到 DRAM，读频繁的数据迁移到 NVM 是该场景下的数据布局目标。另一方面，对于性能敏感的应用，由于 NVM 的读写延迟、带宽均不如 DRAM，因此需要尽可能的将造成内存读写数据置于 DRAM。而由于 NVM 的寿命无法匹敌 DRAM，其寿命主要受限于每个内存单元有限的写次数。因此在需要延长异质内存寿命的应用环境下，不但需要将写频繁的数据迁移到 DRAM，还需要考虑到将 NVM 上的内存写平

均到各个内存单元。

因此根据应用的具体需求，准确识别出关键数据，并将数据迁移至 DRAM、NVM 是该类异质内存的核心问题。如果识别的数据不准确，不但会因为对 NVM 频繁访问造成性能下降，还会因为迁移大量无用数据带来显著的管理开销。

本文关注基于托管式语言开发的应用，并以解决大数据计算内存不足的问题为最终目标，因此，本文的首要关注点是应用在异质内存上的性能问题。虽然 NVM 的性能不如 DRAM，但其处于同一级别，而非固态硬盘 (SSD) 与 DRAM 之间的性能差距，同时为了合理利用所有内存空间，本文最终选择了平级结构来组织异质内存。本文面对的主要挑战为如何对 TB 级别的大量数据进行冷、热划分和布局，并解决分布式系统、运行时系统、操作系统内存管理模块之间的冲突问题。

1.2 托管式语言带来的问题和机遇

运行时系统带来的新问题

因为托管式语言的可移植性，主流的大数据平台均基于其开发，如 Spark、Hadoop 等。而使用托管式语言开发的程序需要基于运行时系统 (Managed Runtime) 执行，运行时系统具有自己的内存管理模块，其中包含了数据的分配、回收等功能。对于异质内存管理而言，运行时系统所具有的垃圾回收机制（Garbage Collection, GC）会打乱操作系统和用户所操控的数据布局，如图 1.2 所示。本节以 Java 语言的运行时系统为例来对该问题进行介绍。Java 运行时系统 (Java Runtime Environment) 主要包含了 Java 程序执行时所需要的库函数 (Java Libraries)，以及核心的 Java 虚拟机 (Java Virtual Machine)。其中，Oracle 公司的 Java 虚拟机 HotSpot[17] 为业界最主流的运行时系统，其被集成在开源的开发环境 OpenJDK 中。下面均以 OpenJDK 8 的 Parallel Scavenge GC 为例进行解析。

如图 1.2 所示，T 是一个被访问频繁的数据对象 (Object)，操作系统会将其所在 NVM 物理页 (Physical memory page) 中的数据迁移到新的 DRAM 物理页中，并重建页表的虚实对应关系。而此时如果发生 GC，由于需要维护数据的局部性，数据对象 T 将会被移动到新的虚拟地址 (Virtual memory) 中，如 To Space 或者旧生代区域 (Old Generation)。而该新的虚拟地址所对应的物理页可能是任意的 DRAM 物理页或者 NVM 物理页，从而打破了操作系统的数据布

局。正是这种操作系统和运行时同时在物理空间和虚拟空间进行的数据移动造成了数据布局冲突的问题。因此，运行时系统的存在给传统的编译、操作系统的异质内存管理方法带来了新的挑战。

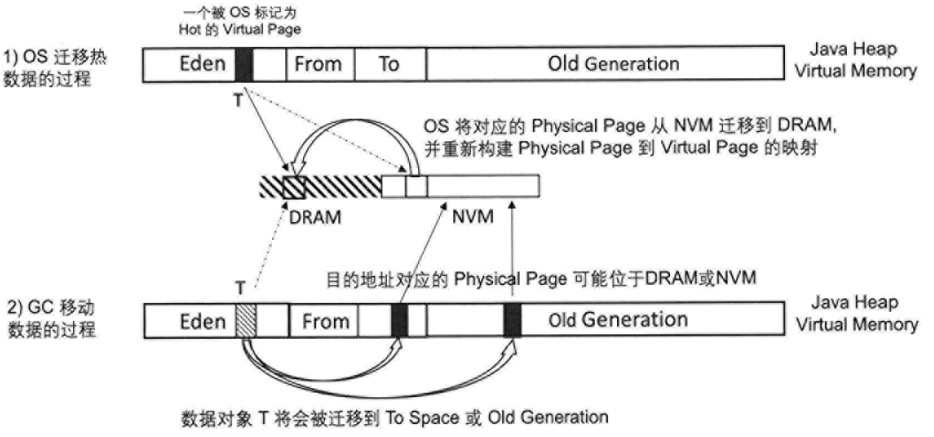


图 1.2 GC 对数据的移动会破坏操作系统做出的数据布局

Figure 1.2 GC can mess up the data layout placed by OS

运行时系统带来的新机遇

由于大数据应用中使用的内存容量通常较大，基于操作系统、硬件的固定数据粒度的异质内存管理策略往往具有扩展性问题[5][13]。运行时系统可以比操作系统、硬件获取更多的高层程序语义，能够在多种粒度进行冷、热数据划分、迁移，并利用 GC 进行低开销的数据迁移，因此，运行时系统的特性为异质内存管理带来了新的机遇：

I 利用硬件、操作系统在固定粒度，如缓存块 (Cache line)、页 (Page) 等，进行关键数据的识别、迁移所造成的开销和程序的内存使用量成正相关[5]，该种管理方式会给数据量巨大的内存计算应用带来显著的管理开销。而运行时可以利用上层程序语义在多种粒度进行冷、热数据的划分，比如直接通过语义准确判定数 GB 数据的冷、热属性，显著的节省在线 (on-ling)、离线 (off-line) 数据分析开销；亦或是根据应用需求在数据对象 (Object) 粒度灵活的选出特定属性的数据。而硬件、操作系统难以根据应用需求进行如此灵活粒度的调整。

II 当前主流的内存计算系统多基于托管式语言开发，在操作系统和大数据框架之间有运行时系统的存在。运行时系统自身的内存管理模块会在

GC 时移动数据，导致操作系统、硬件的数据布局被 GC 打乱，造成数据管理的低效性。但另一方面，GC 移动数据的行为可以被利用进行数据移动[18]。同时，GC 还是一个多线程并发的图遍历过程，可以被用来在数据对象之间传播数据属性信息。而且这些操作并不会引入新的数据移动开销，仅仅是改善原有的 GC 数据移动策略。

本文围绕“直接利用运行时系统管理异质内存”的思想开展研究，以 Java 虚拟机为例，对运行时系统内部机制进行了分析、探索。本文关注的首要问题为如何利用运行时系统自身存在的数据组织形式进行冷、热数据划分，以及如何获取上层程序计算语义来进一步指导数据的布局。为了解决该问题，我们充分挖掘了 GC 机制的能力，利用其特点进行数据属性的传播以及数据的移动。

此外，为了解决真实的应用问题，本文的研究虽从科研运行时系统 JikesRVM[19] 入手，但最终将策略实现在了主流的内存计算平台 Spark 和业界广泛采用的运行时系统 OpenJDK 之上。OpenJDK 具有相当精细、复杂的优化策略，但并未提供细致的说明文档，对原有机制的轻微修改便可能造成显著的性能下降，这也是本研究面对的困难之一。

1.3 内存计算系统中的问题和机遇

内存计算系统带来的新问题

目前大数据系统的设计目标主要为追求高可扩展性和容错性[20][21][22][23]，为了降低编程难度，其一般采用表达式相对简单易懂，但支持高并发的计算框架。这就使大数据系统在设计过程中提出了自己的复杂数据类型和区别于常规程序的计算行为。而这种有针对性的设计，导致了其数据的使用特点和运行时系统的管理机制不再匹配，如内存计算框架 Spark 以自定义的 RDD 结构为单位进行数据计算和管理，而运行时以传统的数据对象 (Object) 为粒度进行管理。这种机制的不匹配，导致对运行时系统中一些原本的设计假设不再成立。如“多数数据在新生代区域快速死亡”的假设在 Hadoop 等大数据应用中不再适用[24]。这不但给运行时系统的数据管理带来了不必要的开销，如在一些大数据应用中 GC 开销甚至占据了程序全部运行时间的 50% 以上[24]，也阻碍数据属性在上层应用、下层运行时之间的传递，给异质内存的数据管理带来了新的挑战。因此使运行时系统重新正确的感知上层应用的数据使用行为、计算行为，并合理利用这些语义来面向大数据应用进行特定的数据管理优化也是一

个重要研究课题。

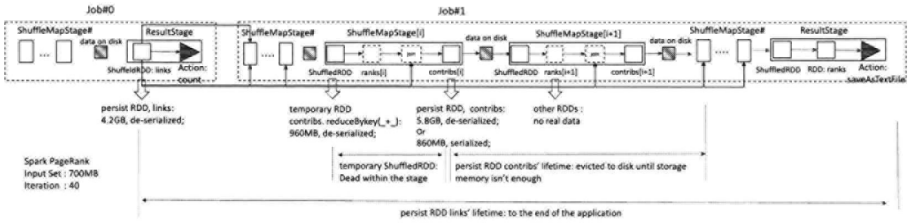


图 1.3 Spark PageRank 的计算流程

Figure 1.3 The calculation flow of Spark PageRank

另一方面，为了加速计算的过程，内存计算应用将大量的数据缓存到了内存中。该行为虽然会减少磁盘 I/O 带来的性能下降，但如果缓存的数据类型不当，却会带来显著的 GC 开销，拖慢整个程序的执行。我们将在第 4 章中对该现象进行进一步分析，并在最后提出的一体化编程框架中对该问题给出了应对策略。

内存计算系统带来的新机遇

以 Spark 应用为代表的内存计算应用具有清晰明了的计算过程，其基本数据单位为只读的 RDD (Resilient Distributed Datasets)，并且每一次计算都会生成一个全新的 RDD。通过对典型的 Spark 迭代计算应用进行分析发现，其每次迭代计算仅仅涉及了数个不同的 RDD，并且其上发生的计算十分简洁、明了，开发人员可以快速就此识别出 RDD 所具有的冷、热属性以及生命周期属性。

图 1.3 展示了 Spark PageRank 的计算流程，从中可以看到，RDD 变量 *links* 可以存活到程序结束，并几乎参与了每一个 Stage (Spark 的基本计算单元) 的计算过程。另一方面，在内存空间充足时，每一次迭代过程中生成的 RDD 变量 *contribs*，会因为容错性的需求而被持久化并储存到内存中。其虽然具有较长的生命周期，但是却仅参与了一个 Stage 的计算。而其他的大量临时 RDD 数据均无法存活过一个 Stage 单元的计算。

因此，从开发者角度可以快速的辨别出各种数据的冷热以及生命周期的长短。如果可以将这些信息传递到底层运行时系统，指导数据在异质内存上的布局，将会极大的减少数据的分析开销并增加迁移的准确度。但是，这里的挑战为 RDD 为一个复杂的自定义数据结构，从 Spark 应用层的 RDD 变量到运行时系统中的数据对象 (Object)，需要跨越 Scala、Spark、Java Object Model 三层

数据模型。此外，每个 RDD 在运行时系统层次对应了上百万独立的数据对象，运行时系统并无法感知这些数据对象之间的从属关系，以及其上的计算信息。因此，如何让运行时系统获取这些上层应用的语义信息，并据此优化数据的识别、布局过程，是本文的一个研究重点。

1.4 研究动机和思路

管理大数据应用在异质内存上的数据布局时，将会遇到 1.1 节，1.2 节，1.3 节中所描述的挑战。而本文旨在研究各个系统层次给异质内存管理带来的问题，并以托管式运行时系统为核心，提出贯穿大数据系统、运行时系统、异质内存的一体化数据管理方案。

首先，针对 GC 与操作系统数据布局管理冲突的问题，如图 1.2 所示，本文尝试通过直接利用 GC 来进行数据布局，而操作系统为运行时系统提供操作异质内存中 DRAM、NVM 的支持。和操作系统的数据布局将会带来额外的移动开销不同，我们只是改变了 GC 移动数据的目的地，并没有没有引入新的数据移动开销。该种策略的挑战为需要通过运行时系统，或者更上层的应用开发人员、编译分析来获取应用的数据信息，以便进行数据的冷、热划分。本文的核心关注问题为，通过合理布局数据来提高应用在异质内存之上的性能，由于 NVM 的读、写性能均弱于 DRAM，理论上对其读、写都会造成一定比例的性能下降。因此本文对“关键数据”的定义为造成众多物理访存的数据。本文的第一个研究面向单机应用，并基于研究型运行时系统 JikesRVM 来探讨利用运行时系统对异质内存进行全面管理的可能性。

大数据系统将会使用巨量的内存，这会给数据的监测、移动带来显著的开销，单纯的硬件管理策略，更会带来显著的 Tag 开销[13]。托管式运行时可以直接获取程序的语义，并将这些信息用于数据的粗、细多级粒度的划分[18]。多粒度的数据划分对于处理大容量的内存系统有良好的效果，如中科院计算所提出的，首先以数据分配位置 (Allocation site) 为标准对数据进行粗粒度的冷、热划分，然后再利用操作系统在页粒度 (Page) 进行细粒度数据布局的方式[5]。该种策略是针对 C/C++ 程序提出，并且该方式需要对程序进行离线的分析来获取数据对象之上的访存次数信息。由于数据对象在运行时系统中会被频繁移动，难以应用以上方式统计数据对象之上的内存读写信息，同时该方式亦不适用于含有巨量数据对象的的分布式的内存计算应用。因此，本文尝试直接利用

运行时系统固有的语义来进行粗粒度的初始数据布局，并通过获取上层程序计算语义来进行细粒度的数据划分。

最后，和计算行为复杂多样的单机应用相比，迭代式内存计算应用 (如 Spark PageRank, Spark K-Means 等) 的计算行为直观、简洁，如图 1.3 所示。因此本文尝试基于运行时系统开发相应的编程接口，允许应用开发人员将内存计算应用的数据信息传递到底层运行时系统，来以自定义的数据结构粒度在异质内存中进行数据布局。而前述 1.3 节中描述的大数据应用、运行时系统之间的数据抽象不匹配行为阻碍了数据属性信息的传递，为此本文尝试开发跨越多层的信息传递通道, 并提出一套面向托管式应用的异质内存管理策略。

1.5 本文的贡献：一体化的异质内存管理框架

本文以“利用托管式运行时系统管理异质内存”为核心思想，探索并分析了单机应用、内存计算应用和运行时之间的交互行为，和利用运行时堆 (Java heap) 现有特点以及程序语义进行多粒度数据划分的可能，最终面向单机应用、大数据应用分别提出了有针对性的冷、热数据划分策略。本研究最终在业界广泛采用的真实内存计算框架 (Spark) 和运行时系统 (OpenJDK) 中实现了一体化的异质内存管理系统。本文的贡献总结如下：

I 为了探索利用托管式运行时管理异质内存的可能性，本文第一个研究以科研型 Java 虚拟机，JikesRVM 为基础，针对 Dacapo、SPECJbb 等单机应用，系统的提出了一种粗细两级粒度的数据划分方式以及布局策略。该策略可以概括为：

i **粗粒度的数据划分。** 运行时系统按照数据对象 (Object) 的生命周期、数据类型等属性将其划分为了数个区域 (Space) 来进行管理，本研究通过在多配置下对各个区域进行物理访存密度的定量分析，提出了可以直接利用这些现有区域进行粗粒度的数据冷、热划分，以减少数据监测、分析、划分开销的策略。这些区域可以分为冷 (Cold)、热 (Hot)、一般 (Normal) 三类。其中冷（Cold）、热 (Hot) 区域将会直接被映射到 NVM 和 DRAM 。而对于处于“一般（Normal）”等级的区域需要进行细粒度的划分。

ii **细粒度的数据划分。** 针对运行时堆中的数据对象会被 GC 频繁移动，难以对其内存读写进行细致分析的问题，本文提出了一种基于

函数粒度的数据冷、热划分方式。通过对 Dacapo、SPECjbb 等单机应用的分析发现，一个函数中的主要访存行为均由少量“热函数”触发，其访问的数据对象会造成很高的物理访存密度 (Physical memory access density)。基于此，本文开发了“热函数”离线分析工具 (HMProf) 来自动识别该类函数，并利用即时编译器（JIT Compiler）从中选出热数据。

最终，针对以上单机应用，在异质内存中的 DRAM 比例仅为 15% - 25% 之间时，和全部使用 DRAM 相比，仅有 5% - 25% 的性能差。

II 本文通过对典型内存计算框架 (Spark) 的迭代计算应用和运行时系统之间的交互行为进行分析，提出可以基于运行时系统对内存计算应用中的数据进行粗粒度的冷、热划分。内存计算应用的“Map - Reduce”计算行为，使其数据的生命周期具有大数据应用的“世代 (epoch)”特性；同时托管式语言 (Scala) 的数据使用低效率特性亦让其保留了“新生代数据快速死亡”的传统数据生命周期特性。另一方面，由于运行时系统 GC 行为的存在，内存计算应用将大量数据置于内存可能会带来显著的 GC 开销，并无法加速程序执行。基于这些数据行为特征，本文提出可以按照临时 RDD 数据、容错 RDD 数据、热 RDD 数据来对内存计算应用中的数据进行粗粒度的冷、热划分，并针对每种 RDD 数据类型提供了对应的编程接口或运行时布局控制，允许应用开发人员以自定义结构 RDD 为粒度在异质内存中进行数据布局。

III 本文提出了一种“大数据系统 - 运行时系统协同数据布局管理”的策略，并在内存计算框架 Spark、Java 运行时系统 OpenJDK 之上实现了贯穿大数据、运行时、异质内存的一体化异质内存管理框架。通过开发数据信息传递通道，大数据系统的语义信息可以传递到底层运行时系统，从而使运行时系统中巨量孤立的数据对象按照上层程序的数据属性进行划分。运行时系统感知到上层程序语义后，不但可以在应用自定义数据数据结构的粒度上在异质内存中进行灵活的数据布局，还可针对具有不同属性的数据对象群进行定向的 GC 优化。通过基于内存计算框架 Spark 开发的编程接口，应用开发者对单一 RDD 变量进行标注后，标注信息首先会在 Spark 系统内部具有“内存数据依赖关系”的 RDD 之间

传播；然后，信息会利用 GC 机制在运行时系统中的数据对象之间进行传播，并按照本文提出的“Migration-Driven Allocation ”策略指导数据对象在运行时系统中的分配、迁移过程。使用本文提出的编程接口，对现有的 Spark 迭代计算应用 (如 Spark PageRank、SparkK-Means、Spark LR、Spark TC) 进行轻微修改后，便可使其以 RDD 粒度在异质内存中进行数据布局。在异质内存中的 DRAM 比例仅为 25% - 35% 时，可以达到全部使用 DRAM 时 80% - 97% 的性能。

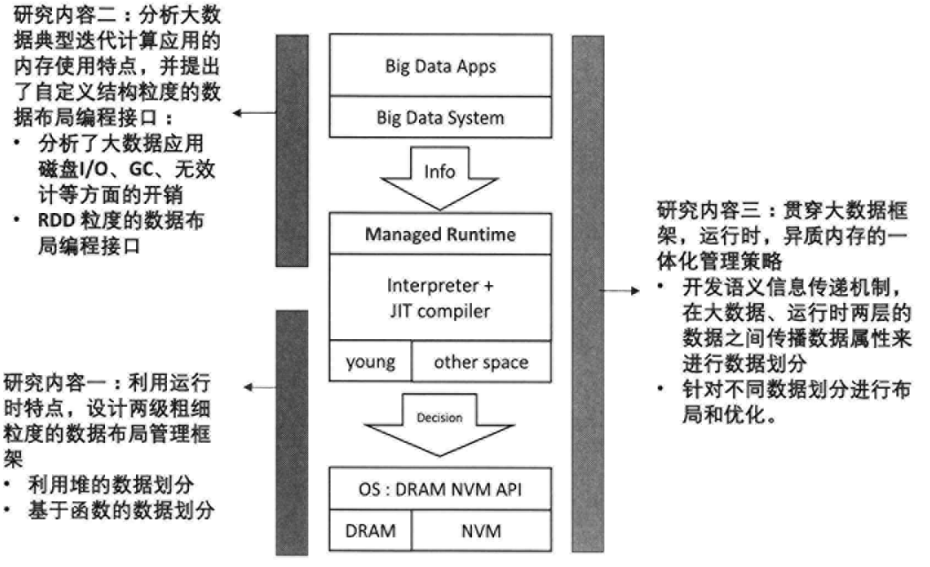


图 1.4 贯穿大数据系统、运行时、异质内存的的一体化管理系统

Figure 1.4 An integrated heterogeneous memory management system

1.6 本文的组织

本文分为六个章节，三个主要研究内容，各研究内容之间的关系如图 1.4 所示。

第 1 章概述了非易失性内存 (NVM) 的特性，以及由 NVM 和 DRAM 构成的异质内存给内存管理带来的复杂性，并阐述了基于托管式语言开发的大数据框架给内存管理带来的挑战和机遇。本章最后亦讨论了本文的研究思路、动机与在该领域所做出的贡献。

第 2章从异质内存管理、大数据应用和托管运行时之间的交互、利用托管式运行时管理异质内存三个方面阐述了当前的主流研究工作，并和本文的研

究思路进行了对比。

第3章介绍了利用托管式运行时管理异质内存的方式。该章首先介绍了基于NUMA架构对异质内存进行仿真的方法，并在随后小节中阐述了针对单机程序，利用运行时系统自身数据管理特点和程序语义信息对程序数据进行两级粒度划分、布局的策略。最后，在科研型Java虚拟机，JikesRVM之上实现了本章中的策略，并对其进行了性能测试。

第4章介绍了内存计算应用(Spark应用)计算行为、数据使用行为，并对内存计算应用和运行时系统内存管理模块之间的交互行为进行了探讨。该章亦介绍了根据以上分析结论，对内存计算应用进行自定义数据结构粒度(RDD)内存布局的可行性，并通过在实际内存框架Spark、运行时系统OpenJDK之上的初步实验对此进行了验证。

第5章介绍了贯穿大数据系统(Spark)、托管式运行时系统(OpenJDK)、异质内存的一体化异质内存管理框架的策略设计和实现。本章分别阐述了编程接口的设计和使用方法，跨系统层次的“大数据系统-运行时系统协同数据布局管理”的策略细节。并在最后，通过典型的Spark迭代计算应用展示了本文提出的框架的效果。

第6章对全文进行了总结，并提出了进一步研究的方向。

第 2 章 相关研究工作

本研究贯穿了应用层的大数据框架、运行时系统、异质内存系统三个层次。因此，本章节将从异质内存的管理策略、基于托管式语言开发的大数据系统相关优化、通过运行时系统对异质内存进行管理三个方面来进行介绍。只有了解 NVM 的相关物理特征，才能利用其解决相关应用场景的问题。因此，本章首先介绍了 NVM 的特点，以及针对 DRAM、NVM 构成的异质内存的数据管理工作，详见2.1 节。其次，需要了解应用的计算行为、数据使用特征，才能进行针对性的优化工作。因此，本章随后介绍了大数据系统计算特性、数据管理特性方面的相关工作，以及大数据系统和运行时系统之间的交互行为，详见 2.2 节。最后，本章介绍了目前已有的，和 NVM 相关的运行时系统优化，详见 2.3 节。在介绍相关工作的过程中，我们对比了现有工作和本文工作的异同，并指出了本文面对的新问题，和解决方法的创新之处。

2.1 针对异质内存的数据管理

本文在 1.1 节中介绍了 DRAM 和 NVM 之间的两种主流组成结构，层级结构和平级结构, 如图 1.1 所示。针对不同的异质内存结构，需要应用不同的数据划分与布局方式。根据异质内存中的数据管理方式分类，常见的有硬件管理策略，硬件、软件结合的管理策略，以及纯软件的管理策略。本章按照该顺序对现有管理策略进行介绍，其中，利用运行时系统 (Runtime System) 管理异质内存数据布局的策略将在本章最后集中阐述。

层级结构如图 1.1(a) 所示，此时 DRAM 作为 NVM 的缓存，DRAM 和 NVM 中的数据以 Inclusive 的形式存在，牺牲掉了 DRAM 的空间。数据移动的方式可以是硬件的级联方式，也可以是由操作系统、编译等控制的软件方式。平级结构如图 1.1(b) 所示，CPU 可以同时访问 DRAM 和 NVM。此时数据以 Exclusive 的形式存在，即此时数据在异质内存中只有一份，可以被储存在 DRAM 中或者是 NVM 中。此时的数据在二者之间的布局管理以软件方式为主。

2.1.1 硬件管理策略

硬件管理策略多采用层级结构，将 DRAM 作为 NVM 的缓存，通过类似于 Cache 的管理方式来将 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，Qureshi 等人在 [13] 中提出了由硬件管理的异质内存结构，并提出了延迟写、细粒度写回等策略在保证性能的前提下来提高 NVM 的使用寿命。最终，该研究认为针对单机应用，仅需使用 NVM 容量 3% 的 DRAM 做为缓存，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[25]、宾夕法尼亚州立大学[26]、微软研究院[27] 等也提出了采用硬件方式来对异质内存进行管理的策略，以节省功耗、提升 NVM 寿命。由硬件进行冷热数据划分、迁移的开销较小，但是硬件管理策略不适用于内存容量巨大的大数据应用，如 Wei 等人在 [5] 中提出，固定粒度的数据监测、迁移开销和应用的内存使用量成正比。另一方面，如果 DRAM、NVM 容量过大，DRAM 缓存的 Tag 标志位等亦会造成很大的浪费。如 [13] 中所用的配置，32GB NVM，1GB DRAM，16 路组相联 (16 Way), 缓存块 (Cache line) 大小在 256 Byte 时，便需要 13% DRAM 空间用作 Tag 标志位。此外，如果大数据程序的局部性不好, 如具有流式访存行为，会引起数据频繁的换入、换出缓存的行为，反而会造成应用性能下降。

2.1.2 软硬件结合的管理策略

硬件管理策略虽然在数据行为监测、移动方面开销低，却需要针对硬件进行特定的设计，花费较大，而且管理策略形式单一。使用软、硬件结合的方式可以应用更加复杂的算法，可对不同的应用类型进行特定的策略调整，更加灵活、高效。

该种方式可以由硬件提供数据监测信息并进行数据的冷热划分与迁移，最后由操作系统来进行页表的维护[12]。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[28][29][30]；亦或者由硬件、操作系统、程序分析联合完成数据的冷、热划分，再由硬件进行数据布局[5]。该种管理方式同时适用于层级结构和平级结构，较为灵活。如，Ramos 等人提出了软硬结合的 Rapp（Rank-based Page Placement）算法[12]，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，并据此将热页面换入 DRAM，冷页面替换出到 NVM。但是基于历史信息的数据访存行

为分析方式适用的应用类型有限，如其无法识别以随机访存为主的应用。另一方面，针对大内存容量的应用类型，该种在线监测方式也会带来显著的开销。Wei 等人通过对程序分析发现，由相同位置 (Allocation site) 产生的数据对象往往具有相似的访存特征[5]，并基于此提出了 Rapp 的改进算法，2pp：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。该策略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 [31] [32]。

硬件与操作系统结合的管理方式，多是以页 (Page) 为粒度。但是对于不同类型的程序，固定的页粒度往往不是最优的选择，如 Hassan 在研究 [33] 提出，在某些应用中 (jpeg) 以数据对象 (Object) 为粒度进行数据迁移可以比页粒度获得更好的功耗收益。此外，对单节点内存使用量在 TB 级别的大数据应用来说，页粒度的数据监测、迁移显得过于细粒度，会造成较大的监测开销。因此，根据应用需求，结合软件分析，进行多级粒度的数据划分、布局是一种更加灵活、高效的策略。

2.1.3 开发人员、程序分析控制的管理策略

程序的计算行为、数据的结构特征往往决定了程序的访存特征。而应用的开发人员往往对程序的计算行为具有清晰的认知，因此，通过编译支持、程序分析等手段帮助开发人员进一步理解程序行为，以便其做出更加准确的数据布局是一个重要研究方向[34]。另一方面，对于一些结构复杂的程序，开发人员并不能给出准确的数据布局指导信息，而且开发人员的经验也限制了程序的优化效果，因此开发编译自动分析、自动离线分析工具来划分冷热数据，并指导数据布局也是一个重要方向。

由于高性能计算 (High Performance Computing) 领域中的应用的计算规模快速增长，其对内存容量的需求非常迫切，并已经开始通过部署 NVM (NVMe SSD) 来缓解 DRAM 容量增长缓慢的问题[35][36][37]。Kim 等人针对 NVM 和 DRAM 构成的异质内存架构，基于 MPI 开发了一套面向高性能计算机 (High Performance Computer, HPC) 的 Key-Value 编程框架，PapyrusKV[38]。PapytusKV 将 DRAM 做为 NVM 的高速缓存使用，将频繁读、写的数据存储于 DRAM，而当 DRAM 空间不足时，便将对应的内存数据结构 (In-Memory Structure) 压缩为适合存储于 NVM 的数据结构。由于 NVM 的存储密度大，比传统磁盘的性能

好，因此可以显著增加应用的计算规模和处理速度。此外，该研究还利用 NVM 的非易失性，将其用来存储节点间共享的数据，避免使用传统磁盘造成的性能瓶颈。然而，该框架仍旧是将 NVM 视作存储级别的结构 (Storage class)，并使用 SSD (Solid-Storage Drive) 对其进行模拟。然而，本文的研究中，NVM 将会被直接用于内存层次 (Memory class) 来存储 In-Memory structure 的数据，比如 Java Object Model 。

Dulloor 等人通过对大数据程序进行分析，根据其访存的形式，将访存分为顺序访问 (Sequential access)、指针依赖类型数据访问 (Pointer-Chasing access)、随机访问 (Random access) 三类, 并提出每一类访存形式的开销亦各不相同。对于同样数量的读写，指针依赖类型数据访问的延迟开销远大于顺序访问类型。由于顺序访存易于实现软件、硬件预取，其可以利用高并发访存来掩盖 NVM 的长延迟带来的性能瓶颈。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[34]。但是该研究提出的数据信息采集方式和迁移方式仅适用于 C/C++ 程序，由于运行时系统中的数据对象会被 GC 反复移动，难以收集其上的访存类型，更难以手动控制其布局。因此，该方法并没有能力分析、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，如 Hadoop、Spark 等往往是基于托管式语言开发。

类似的, Piccoli 等人在研究 [39] 中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、硬件的管理策略往往以页 (Page) 为粒度的数据迁移，编译器、应用开发人员控制的数据布局策略往往可以做到更加灵活的数据管理粒度，如 [33] 提出了编译分析和操作系统相结合的方式，在数据对象 (Object) 级别进行细粒度冷热数据分析、迁移。上层应用若想直接管理 NVM 往往需要操作系统给予一定的支持，Dulloor 等人在 [34] 中提出了一套允许处理器可以直接访问 NVM 的文件系统 PMFS。但是，数据的访问特点往往是随着程序执行而不断变化的[40]，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，很多程序分析往往需要进行离线形式的分析 (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，另一方面，针对程序进行频繁的离线分析也会增加开发人员的工作量。

2.2 大数据框架的内存管理优化

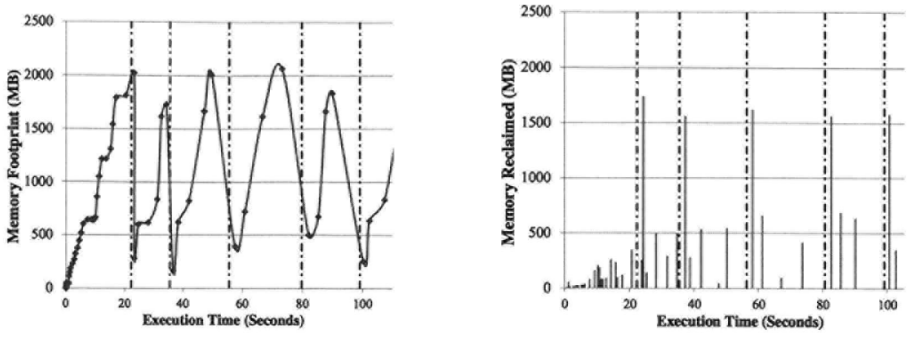
本节针对基于托管式语言开发的大数据框架的内存管理优化工作进行相关的介绍。随着上层软件系统的发展，越来越复杂的软件层次导致了程序执行特点、语义信息的多样性与复杂化。而上层程序语义在下传的过程中的逐渐丢失，导致运行时系统和上层软件系统之间出现了多个层次的不匹配行为。如垃圾回收机制的一个设计理论基础–“多数数据的生命周期是短暂的”和大数据应用的数据使用行为的不再匹配；运行时系统处理数据的粒度（Object）和大数据框架使用的数据粒度（如 Spark 中的 RDD）不匹配等。造成这些不匹配的原因在于，相对于单机应用，大数据应用的计算行为简洁、清晰，导致数据的生命周期也更加规则化。另一方面，为了追求高并发的计算，大数据框架往往以自定义的复杂数据结构（如 RDD）为粒度进行计算，而运行时系统仍旧以传统的数据对象 (Object) 为粒度进行数据管理。这些上下层软件的不匹配现象不但导致了运行时系统自身机制的低效性，也阻碍了运行时系统、操作系统利用上层程序语义来管理异质内存中的数据布局。只有了解大数据应用的计算行为和数据使用特征，以及大数据框架如何与运行时系统进行交互，才能针对该类应用设计出高效的冷热数据划分、布局策略。

以下从两个方面介绍了大数据框架和运行时系统之间的交互行为和内存管理优化：1）从运行时出发，修改运行时系统机制来匹配大数据应用的计算行为；2）从大数据框架出发，修改大数据框架的内存管理机制来重新符合运行时系统的特征。在阐述这两个问题的过程中，本章展示了大数据应用的内存使用特征。

2.2.1 优化运行时系统的内存管理

Nguyen、Xu 等研究者通过对 Hyracks, Hadoop, GraphChi 等大数据框架中数据的生命周期进行分析，提出了新的大数据生命周期假设，“epoch（世代）假设”，并在 HotSpot[41] 基础上实现了新的 GC 机制，Yak[24]。其研究发现，大数据的规则计算行为会导致数据生命周期具有 epoch （世代）的特点，即大多数的数据在一次循环迭代开始时产生，并在循环迭代结束时死亡。

当前运行时的 GC 策略使用的数据生命周期假设为：多数数据对象 (Object) 的生命周期都是短暂的。因此开发者将整个运行时堆 (Java Heap) 划分为两个 (或者多个) 生代区域，如新生代区域 (Young Generation)、旧生代区域 (Old



(a) GC 后 Java Heap 内的数据　　(b) 每次 GC 释放掉的数据量

(a) The alive data after each GC　　(b) The data released by each GC

图 2.1　大数据应用中 GC 行为的分析[24]

Figure 2.1　The GC behavior analysis of big data applications[24]

Generation)。针对新生代区域，可以通过频繁的轻量级 GC (Minor GC) 来回收短生命周期的数据，并将少量存活时间较长的数据放到旧生代区域，等待全局 GC（Major GC / Full GC) 处理[42][43]。而一些大数据系统的数据生命周期不再遵循这种假设。图 2.1 别展示了 Hadoop 等大数据应用中运行时堆 (Java Heap) 内部的数据存活量变化情况，以及每次 GC 释放的数据量。从图 2.1(b) 中可以看到，多数 GC 都没有能释放足够数据，而每次迭代结束后的第一次 GC 则释放了大量的数据。因此，在这些大数据应用中，很多被触发的 Minor GC 行为是非常低效的，只能回收极少量数据，同时带来巨大的分析开销 (每次 GC 都需要遍历对应的区域，选出存活的数据)。因此，研究 [24] 针对这些大数据系统提出了新的数据使用假设，epoch 假设，即在每次迭代计算的末尾执行 GC 行为是最高效的，并基于该假设，在运行时堆中划分出一个新的数据存储区域进行管理。

但是，本文经过分析发现，epoch 假设不适用于 Spark 等内存计算应用。图 2.2 展示了 Spark PageRank 执行过程中 GC 释放数据的情况，和运行时堆 (Java Heap) 内部数据的存活情况。横坐标表示，运行过程中的每一次 GC。纵坐标表示数据在内存中占据的容量。深色曲线展现了，每一次触发 GC 前运行时堆 (Java Heap) 内部的数据量。对应的浅灰色曲线展示了 GC 完成后，运行时堆 (Java Heap) 内部存活的数据量。柱子展现了二者之间之差，也即每次 GC 释放的数据量。从图中可以看到，对于 Spark PageRank 来说，每一次 GC 都会释放较多的无用数据，同时其仍保留了 [24] 中所提到的“世代 (epoch)”特征，

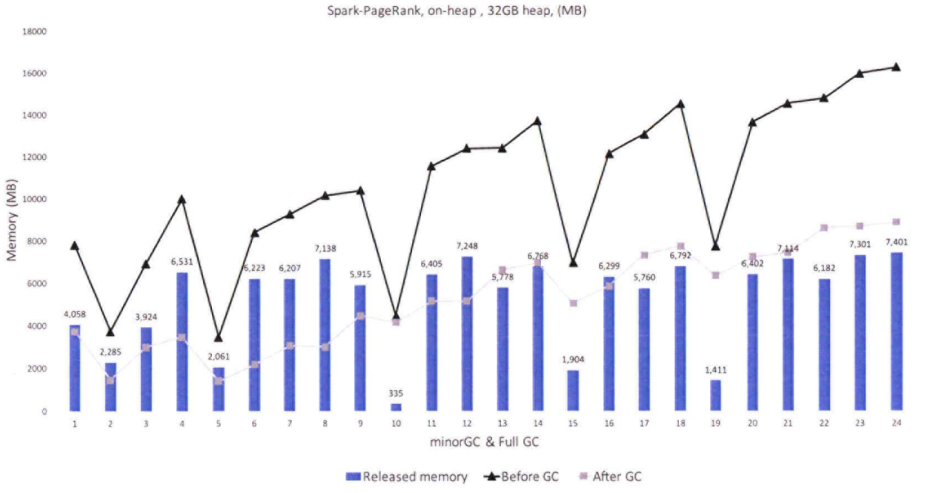


图 2.2　Spark PageRank GC 行为的分析

Figure 2.2　The GC behavior of Spark PageRank

但这仍和图 2.1 中展现的大数据应用中的 GC 行为有很大差距。

在 Spark PageRank 执行的过程中，每一次 GC 均会回收较多的无用数据。这是由于 Spark PageRank 在计算过程中的数据生命周期可以大致四类，短生命周期 RDD、长生命周期的“热”RDD、长生命周期的“较热”RDD、长生命周期的“冷”RDD，如图 1.3 中所示。首先，被开发人员显式持久化的 RDD，如 RDD 变量 *links*，*contribs* 将会长期存活于运行时堆中，并不遵循“epoch”假设。从图 2.2 中可以看到，随着程序的执行，运行时堆中的数据稳步增加。其次，内存计算应用往往具有大量的内存可以使用，Java Heap 设置较大，应用需要经过数次迭代后才可能触发一次 MinorGC, 低频的 GC 行为有利于回收大量的临时数据。最后，Spark 内部的一次迭代，可能包含多个“窄依赖”构成的计算单元，每一个计算单元被称为“Transformation”，每次“Transformation”计算都会生成较多的临时数据。综上所述，即使在一次迭代计算过程中触发 GC，也会回收较多的临时数据。因此，针对大数据应用的传统运行时优化[24] 可能并不再符合 Spark 这类新型内存计算引用的要求，针对 Spark 类型的应用，需要再次分析其特点并做定向的优化。另一方面，内存计算框架 (Spark) 的这种数据使用行为也启发了本文提出的粗粒度异质内存管理策略。

2.2.2 优化大数据框架的内存管理

传统的 GC 策略难以应对大数据系统产生的巨量数据对象[44][21][24][45]。每次 GC 被触发时，均需要以图遍历形式找出特定区域 (新生代区域、旧生代区域) 内所有存活的数据对象，并将其移动到新位置以保证良好的局部性。因此，当运行时堆 (Java Heap) 内部存活的数据对象 (Object) 数目巨大时，GC 机制将会带来显著的分析、迁移开销。某些情况下，GC 消耗的时间甚至占据了整个程序运行时间的 50% 左右[45]。因此，如何优化大数据应用的数据使用行为，并对计算行为进行调整，来减少 GC 开销也是一个重要研究方向。

研究者发现，大数据应用产生的数据对象生命周期较为规律，并基于此提出了由开发者手动进行数据释放的 Off- Heap 的思想。即将大量的、具有规律生命周期的数据分配到本地空间 (Native memory)，由程序员或者编译器来进行释放，不再由 GC 进行自动管理。Databrick 和加州大学，尔湾分校的研究者分别提出了自己的 Off-Heap 框架[46][45] 。其中 Nguyen、Xu 等研究者在 [45] 中提出使用编译器对代码进行分析和变换，并使用 Region-Based 的内存管理思想[19][47][48] 来进行数据管理。而 Databricks 的方法[46] 对程序员提供了 Off-Heap 接口，可以让程序员根据需求将对应的数据分配到 Off-Heap。不过本文认为，将数据的创建、销毁任务重新交给程序员，增加了编程难度，减弱了复杂系统的稳定性，因此仅仅适用于生命周期非常规律的数据类型。另一方面，通过编译对数据状态进行变换的手段，仅能支持较少的大数据系统，无法对采用托管式语言实现的 Spark 系统进行变化。

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 (Java object model) 压缩为字节流 (Byte array)，并直接在字节流上进行计算的思路[46][21]。通过这种数据类型的转化，不但减少了数据所占用的内存空间大小，还极大地减少了数据对象的数目，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，对 Java 数据对象的结构有特定的要求，限制了大数据系统所使用的数据表达方式的范围。同时，这种计算行虽然节省了 GC 开销，但要比直接在 Java 数据对象上带来更大的计算开销，因为在计算之前需要按照一定规则去解析、反序列化字节流数据。当 GC 的开销并不显著时，这种框架并不一定能取得优势。

以上所述工作给本文的研究来了一些新思路：解决上层应用和运行时系统

之间的问题不一定只能从底层出发，通过对应用本身的计算特性做出调整，可以在兼顾计算效率的前提下，再次让新型计算框架重新匹配底层运行时系统的原有设计思路，达到提升整体性能的目的。本文不但通过调整底层的运行时系统来解决问题，同时也在上层的 Spark 系统做出了相应的修改。最终形成了一套贯穿大数据框架、运行时系统、异质内存的一体化异质内存管理策略。

2.3 通过运行时对异质内存进行管理

常规的异质内存管理策略如前文所述，主要为基于硬件、操作系统、应用层接口等方式进行数据的划分和布局。而随着托管式语言在大数据领域、嵌入式领域以及单机平台应用领域的流行，运行时系统所带来的问题和机遇越来越被大家关注。本文的所关注的问题为如何管理托管式语言开发的应用在异质内存上的数据布局。本节汇聚了使用运行时系统来管理 NVM、异质内存的策略，涉及了 NVM 的性能、非易失性、使用寿命等方面。

2.3.1 托管式应用的性能优化

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的应用于嵌入式开发、分布式系统开发，而基于托管式语言开发的应用需要在运行时系统之上执行。如，Java 语言开发的源程序会首先被转化为字节码 (Byte Code)，然后该字节码便可以直接在任何安装了 Java 虚拟机 (Java Virtual Machine, JVM) 的机器上运行。无需和 Native Language, 如 C、C++ 一样，需要针对不同平台进行多次编译。此外，为了解决指针带来的访问越界问题、难以调试的内存泄露等问题，托管式语言取消了指针类型，并提出了面向对象的 Java 对象模型 (Java Object Model) 和对应的操作，以及垃圾回收 (Garbage Collection) 来进行数据管理。GC 这种数据自动管理的机制不但减轻了开发人员的压力，还有效提高了程序的稳定性。

托管式语言的这些特性带来良好易用性的同时，也带来了一些执行效率低下的问题。在指令层面，这些字节码无法直接被处理器执行，而是需要经过解释器（Interpreter）和即时编译器（Just-In-Time Compiler）的处理后才能被执行。这无疑降低了指令的执行效率。在数据自动管理方面，在目前的 OpenJDK 8 策略中，每一次触发 GC 时应用都会被暂停，等待数据回收、压缩完毕后，再次恢复执行，这种垃圾回收方式被称为“Stop the World GC”。对于数量巨大的

应用程序，GC 带来的开销甚至可以达到程序整体运行时间的 50% 左右[24]。因此，如何减轻托管式语言应用中 GC 带来的开销也是一个研究重点，第 2.2 节对此进行了详细的分析，该节不再进行赘述。

然而，正是由于托管式运行时中存在诸多复杂的机制，因此也带来了多样的优化机遇。在提升计算效率方面，如 He 与 Grossman 等人在 [49] [50] 中提出了用 GPU 来加速基于托管式语言开发的大数据平台的方法。此外还有 Ghasemi 等人在 [51] 中提出的使用 FPGA 来加速计算的方式。这些工作的核心思想都是通过对托管式语言进行转化，使其可以在一些专用的计算设备上运行来加速计算。这种多种计算资源混合的“异质计算”问题不是本研究的重点，因此这里不做过多展开。在提升数据管理效率方面，运行时系统首先按照数据的生命周期、结构特点等特征将其置于了堆 (Heap) 中各个不同的位置。同时，运行时系统又通过 GC 机制来动态调整数据在堆中的位置，以保持良好的数据局部性。因此，科研人员思考是否可以通过调整现有 GC 机制来达到新的目的。

2.3.2 利用运行时系统管理非易失性内存

在 NVM 使用寿命方面。 澳大利亚国立大学的 Gao、Blackburn 等人提出了一套硬件、操作系统、运行时系统结合的 NVM 管理策略，以提升其使用寿命[52]。其设计思想为利用 GC 移动数据的特点来合理布局数据在 NVM 上的位置，通过写均衡、避免使用损坏区域等策略，达到延长 NVM 的使用寿命的目的。当硬件监测到 NVM 中的数据局部损坏时，相比起直接舍弃掉整个页 (4KB)，运行时会避免使用对应的数据块 (Data line, 256 Bytes)。在该种策略下，即使 NVM 的损坏率达到了 50% 时，程序的性能仅会有 12% 的下降。

在利用 NVM 非易失性的方面。 上海交通大学的 Wu、Chen 等人利用 NVM 来直接扩展 Java 运行时堆 (Java Heap), 并提出了 Persistent Java Heap 的堆设计来对其进行管理[53]。为了让应用开发人员可以主动控制数据对象 (Object) 在 DRAM 和 NVM 上的布局，其提供了 NVM 布局接口，以及对应的持久化数据对象 (Persistent Java object)。该工作的目的在于加速 Java 编程语言的持久化特征 (Java Persistent Programming)[54][55]，以便增加程序的容错性，而并非区分冷、数据，以及在异质内存中对数据进行合理的布局。然而，正如我们 1.3节中所述，一个自定义的大数据结构，在 JVM 层可能对应了上百万独立的数据对象 (Object)，这些数据对象的创建很多都隐藏在了编程框架中，如 Spark 框

架内部，让应用开发人员去逐个的将这些数据对象手动布局到 NVM 需要非常大的投入，并不现实。

异质内存管理方面。 筑波大学的 Nakagawa 等人[56][57][58][59] 提出了可以使用 GC 机制来管理异质内存中数据布局的思想。其认为可以将运行时堆 (Java heap) 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。其还提出了利用 Write barrier 机制来监测每一个数据对象（Object）被写的次数来作为冷热数据的选取标准。但是该种策略并没有考虑访存命中 Cache 的情况，同时监测每一个 Object 之上的写操作并进行冷热分析会带来较大的开销。其研究的对象为单机应用程序[60] 和研究型的运行时 JikesRVM，这些应用的计算行为、数据管理方式和基于 HotSpot[41] 运行时系统的大数据应用具有较大的差异。此外，马萨诸塞州阿姆赫斯特大学的 Yang、Hertz 等人[61][62][63] 提出了如何在执行 GC 时避免发生 Swap 数据到硬盘现象，当异质内存为层级结构时，可以借鉴这些研究的成果。由于为了充分利用内存空间，本研究针对的是“平级结构”，如图 1.1(b)所示。

利用 JIT 进行冷热数据识别。 运行时系统中具有的即时编译器（Just-In-Time Compiler）可以在程序执行过程中对频繁执行的函数进行编译优化，而这为本文的研究提供了动态分析、识别性能关键数据的机会。IBM 东京研究院的 Inoue 等人提出了一种通过 Patten match 来识别出容易引起 L1、L2 Cache Miss 的病态 Java 源码的方式[64]。该研究通过编译分析来识别性能关键数据的方式对本文的管理策略具有一定的启发，但是该研究只针对少量的单机 Java 程序进行了分析，本文在大量的单机程序中使用该 Pattern 进行了实验，并未取得良好的效果。此外，本文最终的目的是针对大数据应用，而非单机程序，但是该种研究思路给本文的工作提供了良好的启发。如，本文针对单机的 Java 程序，亦开发出了一套可以在函数粒度进行 LLC Miss 分析的工具，并在多个应用中取得了良好的效果。

2.4 国内外研究总结

上述小节介绍了异质内存管理相关的工作、解决大数据框架和运行时系统之间交互问题的工作、以及运行时系统领域和 NVM 相关的一些研究工作。由此可以看到，运行时系统被广泛的应用了各类应用中，同时研究者围绕着运行时系统相关的问题做了大量的研究工作。但是，针对基于运行时系统的大数

据应用在异质内存上的管理工作却十分匮乏。原因之一便是因为该研究涉及到了包括应用编程、大数据框架、运行时系统、操作系统、异质内存在内的多个领域。本文尝试提出一套贯穿各个层次的一体化管理策略，如图 1.4 所示。

为了提出这样的一体化管理策略，本文将整个研究划分为了三大部分。在每个部分解决对应的问题，并在最后整合所有研究成果为一体化的内存管理策略。虽然当前并没有这种一体化管理策略的相关研究，但是我们仍旧可以通过对各个方面的学习来获取启发，并将其用于各个研究部分。本文通过对现有研究的分析、实验，最终创新性的总结出了本文所使用的方法和策略，最后在真实内存计算平台 Spark、运行时系统 OpenJDK 之上进行了实现。

非易失性内存管理的传统研究。

在优化异质内存管理策略研究领域中，硬件管理方式往往是针对 DRAM 作为 NVM 缓存结构的组织方式。不但需要引入复杂的硬件结构，还会因为 Tag 标志位等开销限制了 DRAM 容量，造成扩展性问题。此外，当大数据应用的工作集较大时，会引起数据在 DRAM 缓存和 NVM 主存之间频繁的换入换出，造成性能和功耗的损失。而硬件、操作系统构成的软硬结合的方式所采取的页 (Page) 粒度，以及在线数据监测、分析的方式同样不适合工作集巨大的大数据应用，会造成明显的数据识别开销。托管式语言中的 GC 机制会根据自身需要频繁的移动数据，而传统的软硬件结合的管理方式缺乏对此的考虑，因此其进行的数据布局会被 GC 频繁打乱。而通过静态程序分析来将数据进行分类，并在数据分配阶段便进行初步布局以节省迁移开销的方式，往往针对的是非 JVM 平台的应用。其对数据对象、结构的访存行为分析方法，以及数据布局方法并不适用于具有 GC 机制的托管式语言应用。通过对该类文献的研究和分析，我们对 NVM 的物理特性具有了良好的认知。同时，通过对一些程序分析工作的研究[5][10]，也让我们了解了 NVM 对哪些访存类型更加敏感，这加速了本文数据分类标准的设计过程。

大数据框架和运行时系统之间交互问题的研究。

大数据平台是异质内存的重要应用场景，而由于当前流行的大数据平台往往是基于托管式语言实现的。而当前的异质内存研究领域中尚缺少贯穿大数据平台、托管式语言运行时系统、异质内存的一体化研究。目前针对大数据框架和运行时系统之间交互的研究，多集中在解决 GC 机制不再适合大数据

应用会制造巨量数据对象的情况。而本研究的目的是解决 GC 打乱数据在异质内存布局的的问题。虽然，本文无法直接从目前的研究中获取解决问题的方式，但是仍旧可以得到很多启示。如从加州大学，尔湾分校的 Nguyen、Xu 等人的工作[24] 中，我们获知了大数据应用对数据的使用特征和单机程序迥异的思想，促使我们进一步研究了 Spark 系统的计算行为、数据使用行为特点。而 Tunsten[46] 的研究，更是本文将 Spark 数据布局到 NVM 的工作基础之一。

面向异质内存的运行时系统研究。

运行时系统因为具有 GC 机制的存在，能够自动的回收死亡的数据，无需开发人员手动进行空间释放。该机制不但给开发人员带来了便捷性，更减少了因为内存泄露而导致的程序崩溃问题。然而，该机制的存在却也给应用的内存管理带了新的挑战，如其会给内存计算应用带来显著的开销，亦或是打乱操作系统、硬件的数据布局。其中，本章在 2.2 节中对 GC 开销问题以及现有研究进行了详细阐述[21][24][45]，而本文旨在解决 GC 打乱异质内存数据布局的问题。

目前主流的 GC 方式为基于“多数数据对象会快速死亡”假设而开发的 Generation GC (分生代垃圾回收)[43][65]，如 Oracle HotSpot[17]。其运行时堆 (Java heap) 被划分为了新生代区域 (Young Generation) 和旧生代区域 (Old Generation)，并被赋以了不同的数据回收策略，分别为轻量级的 MinorGC 和彻底的 MajorGC (Full-GC)。由于该种设计中的 GC 具有数据对象 (Object) 级别的数据移动能力，因此往往被开发者利用来进行数据布局[52]。而本文亦充分挖掘了 GC 机制的能力，来使其具有在异质内存中布局数据、在数据对象之间传递数据属性的能力。

在运行时系统层次对异质内存进行管理的研究较少。其中，比较主要的有 Gao 等人提出的利用 GC 可以移动数据的能力，调整数据在 NVM 上位置，来进行写均衡操作并避免使用损坏区域，以增加 NVM 寿命的目的[52] 。以及 Wu 等人提出的，可以利用 NVM 来扩展运行时堆，并给应用开发人员提供面向 NVM 的编程接口来实现 Java Persistent API 的设计思路[53]。但是，这些研究并没有涉及到利用运行时系统来划分应用中冷、热数据，以及进行数据布局的思路，更没有探讨上层大数据系统的计算行为、数据使用行为，以及其对数据布局造成的影响，但是本文仍旧可以从这些研究中得到很多启示。如从 Inoue 等人提出的可以利用即时编译器 (JIT Compiler) 来对程序进行模式分析 (Code

pattern)，并选择出热数据的思路[64] 得到启发，本文最终开发了一套以函数为粒度，对程序进行离线分析 (Off-line profiling) 的冷、热数据自动识别工具。

面向非易失性内存的编程接口

为了使应用开发人员可以灵活的将特定数据布局到异质内存中的 DRAM、NVM，很多研究者提出了对应的异质内存编程接口[66][53][67]。如 Kannan 等人提出的 pVM 系统[66]，其允许应用开发者可以同时在 NVM 之上进行普通的非持久化数据分配和持久化的数据分配 (Persistent storage)。类似的，Wu 等人亦针对运行时系统提出了 Espresso 架构，其在数据对象 (Object) 粒度提供了对应的数据布局编程接口。然而，这些编程接口仅仅是针对单个数据对象 (Object) 或者是直接分配一块连续的区域 (nvmmap)，其无法自动识别一个大数据框架的自定义结构之中包含的所有数据对象 (Object)，并将这些相关数据对象 (Object) 置于对应的 NVM 或 DRAM 中。如，对于 Spark 的自定义计算单元 RDD 来说，其往往对应了数百万独立的数据对象 (Object)，难以让开发者利用以上接口去逐一手动标注所有的数据对象 (Object)。而本文针对托管式应用开发的异质内存编程接口，仅需让开发者标注一个自定义数据结构 (RDD) 的单一的数据对象 (Object)，便可以利用运行时系统自身的特性去自动传播开发者标注的属性，并将相关的数据对象 (Object) 移动到开发者指定的位置 (DRAM、NVM)。

第 3 章 基于运行时系统的异质内存管理策略

随着 DRAM 技术工艺瓶颈的到来，其容量密度难以持续增加，同时其功耗持续增加，导致 DRAM 很难满足大数据计算等新兴应用的低功耗、大容量内存的需求[6][5]。因此，工业界逐渐将注意力投入到了新型内存材质的研究中[7][13][16][27]。其中，非易失性内存 (Non-Volatile Memory ,NVM) 作为一类新型内存技术受到了越来越多公司和科研人员的关注。但是，目前比较成熟的 NVM 材质，如相变存储器 (Phase-Change Memory)，因为自身物理特性带来了一些新的问题。如，其写寿命难以和 DRAM 技术相比。同时，其访存带宽和延迟性能也难以达到 DRAM 的水平。

但是，NVM 具有高存储密度、低成本、低功耗，数据非易失的特点。这些特性可以被广泛的应用于对功耗敏感的嵌入式设备、对内存容量需求巨大的内存计算等诸多领域。同时，将 NVM 和 DRAM 结合为异质内存也是当今的一个趋势。由于托管式语言被广泛的应用于嵌入式计算、大数据系统开发, 异质内存之上的诸多应用均依赖于运行时系统，而这会给数据布局管理带来新的挑战。如图 1.2 展示了 GC 机制和操作系统异质内存管理之间对数据的移动冲突。

本章的研究是本文工作的基础与第一步探索。在构建贯穿大数据框架、运行时系统、异质内存的一体化内存管理策略时，首先需要解决的问题便是如何解决图 1.2 中所示的数据移动冲突问题。本章提出直接利用运行时系统来管理数据在异质内存上的分布，从而避免多层次管理带来的冲突问题。那么我们需要在运行时系统对应用数据进行冷、热划分；并根据划分结果对数据进行移动。此时的操作系统只需要提供必要的功能支持便可，如允许运行时系统向其申请 DRAM 和 NVM 空间。

3.1 研究动机与概要

为了解决第 1.2 节中所述的，运行时系统中的 GC 机制带来的数据移动冲突问题，本章尝试直接利用运行时系统来对异质内存进行管理，对应用中的数据进行冷、热划分，并将其布局到合理位置。相对于操作系统，运行时系统可以

更好的获取上层程序的语义，并将此用于数据划分。同时其亦具有通过 GC 移动数据的性能力。此时，我们通过 NUMA 架构来对异质内存进行仿真，并利用 mmap 和 mbind 库函数来让运行时系统可以从操作系统获取 DRAM 和 NVM 空间。由于本研究的目的是通过数据布局来让应用在异质内存中获取更优的性能，因此本文对热数据的定义为“对程序性能影响最大的的数据 (Performance Critical Data)”。

3.1.1 GC 机制带来的机遇

目前针对托管式语言提出的 GC 策略众多，如主流的 Generation GC (分生代垃圾回收)[43][65], Mark-Sweep GC (标记-清除垃圾回收)[19]，Reference Count GC （引用计数垃圾回收[68]，Concurrency GC（并发垃圾垃圾回收），Region-based GC （基于区域的垃圾回收)[69] 等。每种 GC 策略都是通过对特定类型的应用进行数据生命周期分析后，根据不同的需求做出的不同设计。

目前工业界的主流 Java 运行时系统，Oracle HotSpot[17]，以及流行的科研用 Java 运行时系统，JikesRVM[19]，均为基于 Generation GC 策略开发的 Parallel-Scavenge-GC (并行扫描垃圾回收)。该策略的设计思路是根据一个统计假设：程序中大多数的数据对象都会在创建后极短的时间内死去。因此，该 GC 策略便将整个运行时堆（Java Heap）划分为了新生代区域（Young Generation Space/ Nursery Space）和旧生代区域（Old Generation Space / Mature Space）两部分来存储不同生命周期的数据。并使用不同的 GC 策略来处理这两类数据：对于新生代区域，因多数数据都会快速死亡，因此需要对其进行频繁的扫描来进行空间释放，该过程被称为 MinorGC/Young GC；而对于旧生代区域，因为其中的数据存活时间较长，因此使用一个低频率的、彻底的扫描来回收其中的数据，该过程被称为 MajorGC / Full GC。

当新生代中的数据存活过一定次数的 MinorGC 时，便会被移动到旧生代区域，如图 3.1 所示为一次 MinorGC 中，将长期存活的数据迁移到旧生代区域的过程。受此启发，我们可以利用高频的 MinorGC 来将已经分类的冷、热数据布局到异质内存中的对应位置：DRAM 或 NVM。另一方面，由于 GC 原本就会对数据进行移动，本文只需要根据被移动数据的冷、热属性，调整其目的位置便可。并不会引入额外的的数据移动开销。本章以科研型运行时系统 JikesRVM 为基础来实现相应的管理策略，经测试发现，利用 GC 来移动数据

引入的额外移动开销小于 1%。

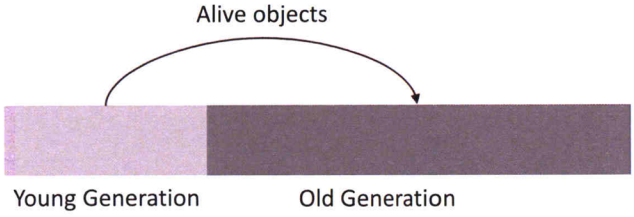


图 3.1　MinorGC 对数据的移动

Figure 3.1　The object movement during MinorGC

3.1.2　运行时堆数据划分机制带来的机遇

运行时系统不但具有通过 GC 机制移动数据的能力，其还管理数据的空间申请、创建和初始化。如图 3.2 展示了数据分配进入 JikesRVM 运行时堆的过程，可以看到，不同数据类型被置于了堆中不同的区域：单个实例化对象（Single object instance），数组 (Array , Type array / Object array)，元数据（Meta data）等都会被运行时系统识别，并被置于运行时堆中不同的位置。此外，分生代垃圾回收（Generation GC）也按照生命周期将数据划分到了不同的区域。

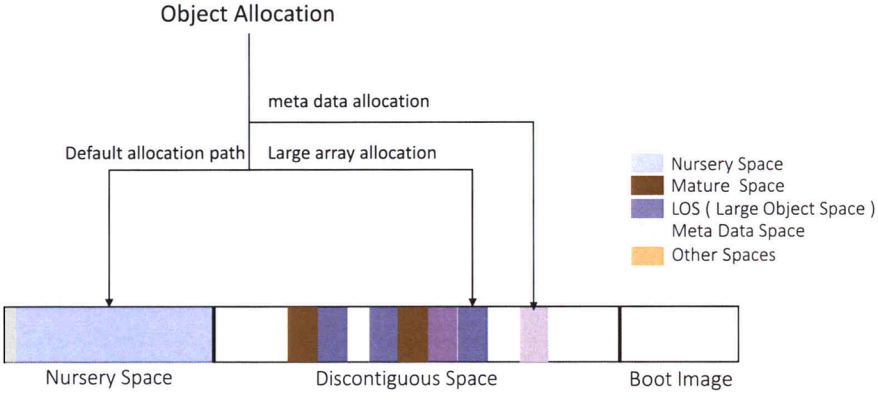


图 3.2　数据对象创建和运行时堆的构成

Figure 3.2　The creation of object and the organization of Java heap

这些数据划分虽然是基于数据类型、生命周期等特征进行，但是经过分析，这种划分和数据的冷、热划分具有强相关性。比如，新生代区域 (Young Generation Space/Nursery Space) 展现出了更高的访存频度和大量的内存写特征；而大数组区域 (Large Object Space, LOS) 则展现出了较低的内存使用效率，

其访存频度不高。各个区域的访存特点给本文进行粗粒度的数据划分带来了新的机遇。经过分析，这种粗粒度的冷、热数据划分可以带来相当高的精度。

3.1.3 即时编译器 (JIT Compiler) 带来的机遇

若想利用运行时系统来管理异质内存数据布局，除必须具备移动数据的能力外，还需要其具备分析、划分数据的能力。相比于硬件、操作系统具有的在线访存行为监测能力 (On-line profiling)，运行时系统可以根据上层程序语义来划分数据，并利用 JIT 编译器来对数据对象进行动态的标注。

以 JikesRVM 为例，为了可以动态的优化频繁执行的函数。运行时系统会监控函数的执行频度，并对执行频度不同的函数进行不同程序的编译优化。这种操作让我们有机会在执行过程中，动态的分析这些执行频度很高的函数，而直接忽略掉那些执行次数少，对性能影响不大的函数。此外，可以按照已经获取的程序语义来利用 JIT 编译器对数据对象进行自动属性标注，而非让应用开发人员去手动的对数据对象进行 DRAM、NVM 标注。

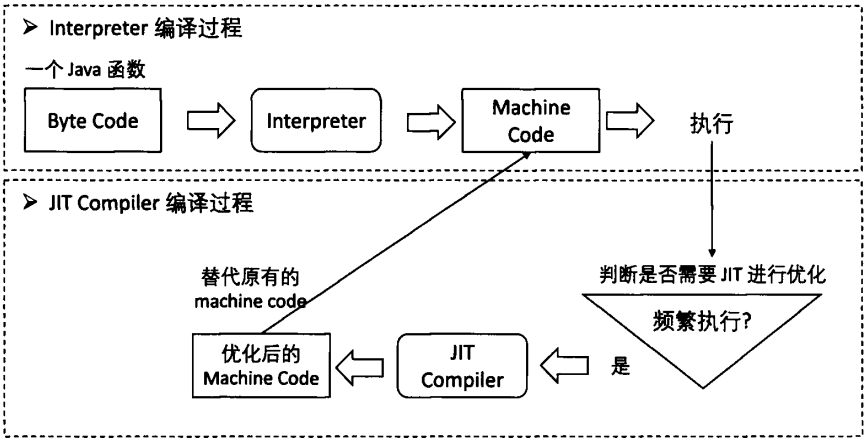


图 3.3 解释器和即时编译器处理函数的过程

Figure 3.3 The behavior of Interpreter and JIT compiler

图 3.3 展示了一个 Java 函数被首次编译、执行，以及被即时编译器（JIT Compiler）优化的过程。从中可以看到，一个处于字节码（Byte Code）状态的函数，被编译执行后便会直接执行。执行过程中，运行时系统会动态的监测函数的执行频度，并利用即时编译器对高频函数进行深度的优化。然后，运行时系统会用优化后的执行代码（Optimized machine code）替换掉旧版本的执行代码。在后续的研究中将利用即时编译器（JIT Compiler）来标记根据程序语义

划分好的数据对象。

3.1.4　本节总结

基于以上所述，可以看到运行时系统不但具有移动数据的能力，甚至可以通过自身的堆组织特点、或者程序语义来对数据进行冷热划分。此时，操作系统仅需给以基本支持，即允许运行时系统向其申请 DRAM、NVM 空间便可。为了让运行时系统可以直接管理异质内存上的数据布局，运行时堆中的每一个区域都需要可以灵活的获取 DRAM、NVM 空间，如图 3.4 所示。本文在随后的实现中，均利用现有的 mbind[70] 机制和 NUMA 架构来实现操作系统的支持工作。

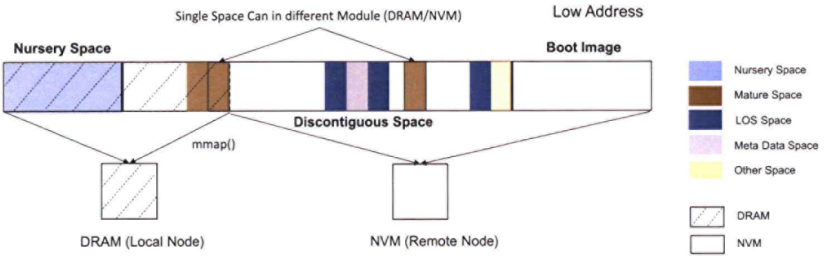


图 3.4　支持异质内存的运行时堆

Figure 3.4　Extend Java Heap to support Heterogeneous Memory

3.2　非易失性内存仿真平台

由于市场中并没有可以在主存级别 (Memory) 使用的非易失性内存产品，而目前最受关注，以及最接近产品的非易失性内存材质为相变存储器（Phase Change memory），因此本研究需要根据现有研究中相变存储器（Phase Change Memory）的参数[5][6][10] 提出一种可以运行托管式语言应用、大数据应用的非易失性内存模拟方式。传统研究中基于 Trace 模拟器的实验方式[5][13] 并不适合运行访存量巨大的大数据系统应用、亦或是基于托管式运行时的应用。因此，本节提出了一种基于非一致性访存架构（Non-Uniform Memory Access Architecture, NUMA Architecture ）的非易失性内存模式方式，如图 3.5 所示，并在真实研究中得到了实际验证[18][71]。

在以性能为目的的 NVM 研究中，对 NVM 进行仿真、模式时，最主要的两个指标为访存延迟 (Memory access latency) 和访存带宽（Memory bandwidth

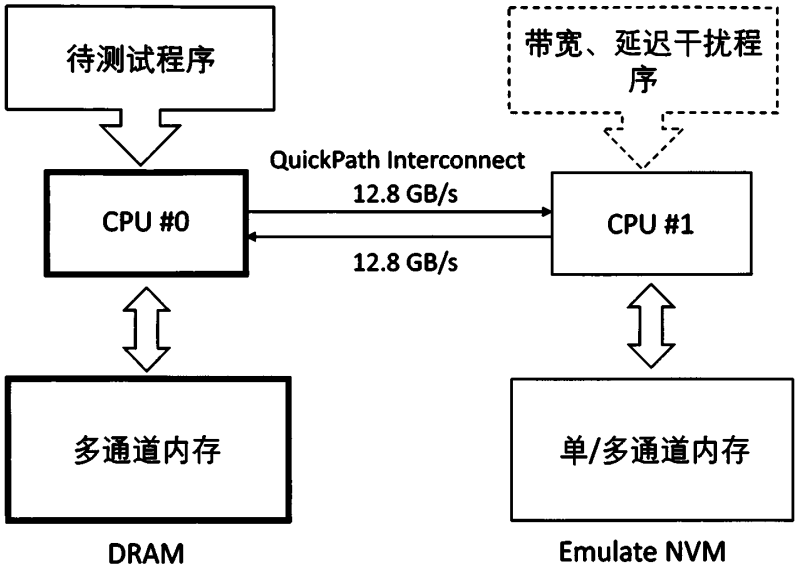


图 3.5 非易失性内存仿真平台

Figure 3.5 The Non-Volatile Memory Emulator

)。本文以研究 [10] 中所使用的参数，如表 3.1 所示，作为标准来调整非易失性内存仿真平台。

3.2.1 访存长延迟仿真原理

在一个拥有两个处理器的 NUMA 架构的 Intel 服务器中，一个处理器访问另一个处理器的所管理的内存时，将会跨过一个被称作 QPI (QuickPath Interconnect) 的链接，然后通过另一个处理器的内存控制器（Memory Controller）来访问其内存。此时，与访问本地内存相比，访问远端内存的物理延迟可以达到访问本地内存延迟的 2 - 3 倍。如图 3.5 展示了该过程。当将计算固定在 CPU #0 上执行时，其访问本地的内存模块延迟只有 120ns, 而访问 CPU #1 的内存模块时，访存延迟达到了 300ns，为访问本地的 2.5 倍。本文采用 Intel 的 mlc[72] 工具来测量内存的单次访存物理延迟，其同时提供了不同读写比例下的访存带宽测试功能。

相应的，在使用 AMD 处理器的 NUMA 架构服务器中，处理器之间也有类似于 QPI 的 HT (HyperTransport) 点对点 (Points to Points) 连接通道，同样会造成高访存延迟和较低的访存带宽效果。但由于并没有相应服务器，因此并未在该种处理器上验证本文的仿真策略。

另一方面，当在远端处理器 (CPU #1) 处理器之上运行一个“干扰程序时”，

可以通过增加 CPU #1 的访存队列压力，而进一步增加该节点之上的访存延迟。我们将在第 3.2.3 节中对此做进一步的讨论。

3.2.2 访存带宽控制原理

在 NUMA 架构的 Intel 服务器中，访问远端内存时，访存的带宽也受到了 QPI 的限制。本文所用服务器的 QPI 的带宽被限制在了 12.8GB/s。由于 QPI 为全双工，因此可以认为访问远端内存时，理论读带宽不会超过 12.8GB/s, 同时写带宽也不会超过 12.8 GB/s。另外，可以更改远端内存的通道 (Memory channel) 数目、添加远端干扰程序来进一步限制带宽。远端内存 12.8 GB/s 的读写带宽仅为处理器本地内存带宽的 40% 左右，因此本文在第 4 章、第 5 章等访存带宽使用量较大的大数据的应用场景下，并未使用干扰程序进一步压缩访存带宽。而在第 3 章针对单机程序的研究中，将带宽压缩为了本地带宽的 1/10 左右。

一些特定的 Intel 处理器可以通过调节 BIOS 参数来精确限制远端内存的带宽[66][34]，但该方法的适用范围受到了处理器型号的限制。本文使用“干扰程序”来控制带宽的方式的理论基础是“程序资源竞争”，对于单一的干扰程序来说，带宽压缩效果会随着宿主程序的带宽变化而变化。然而，同类大数据应用的访存带宽的差异往往不大，如本文针对的 Spark 迭代计算程序的平均访存带宽均在 10 - 16 GB/s 之间，因此单一的干扰程序仍旧适用。

采用干扰程序、调节内存通道数目等仿真方式来限制内存带宽，将不会受到处理器型号的限制，具有更好的灵活性。但是，NVM 一般具有读写不对成的特点，比如其写带宽会显著低于读带宽，然而本节提出的 NVM 仿真方式以及 [66] [34] 中的 NUMA 仿真方式均难以展现该种特点，其具有相同的读、写带宽。然而，理论上可以根据内存控制器的访存请求调度策略，合理调节干扰程序的读写访存请求比例来实现精准调控对宿主程序内存读写请求的干扰效果，我们将在随后的工作中对此进行探索。

3.2.3 添加干扰程序

在一些不同配置的机器中，带宽和延迟的仿真水平并不能达到实验者的要求。针对该问题，本文基于 Stream 程序[73] 开发了一套“干扰程序”。其设计原理为：当被测试程序运行于 CPU #0 时，将干扰程序运行于 CPU #1，如图 3.5所示。此时，多线程、高并发的干扰程序便会发出大量的读、写内存的请求来阻

表 3.1 DRAM 和非易失性内存的性能参数对比

Table 3.1 Comparison of DRAM and NVM technology

	DRAM	NVM
Read Latency(ns)	1x	2x to 4x
Bandwidth (GB/s)	1x	1/8x to 1/4x
Capacity per CPU	100s of GBs	Terabytes
Cost	5X	1X

塞 CPU #1 的访存队列。从 CPU #0 对 CPU #1 的访存请求，延迟会大大增加，读写带宽都会显著降低。此外，本文提供了一个参数，其可以控制向干扰程序中插入空指令 (Nop instruction) 的密度，来调整对 CPU #1 的访存队列的压力，从而达到调整 PCM 仿真延迟、带宽的目的，

NVM 仿真带宽、延迟的效果如图 3.6 所示。从中可以看到，NVM 访存的读带宽随着 Nop 指令密度的降低而逐渐增加，并最终可以在本文所用服务仿真 1/3 - 1/5 DRAM 带宽的效果。正如我们在前述小节中所述，干扰程序的理论基础是“程序资源竞争”，因此单一的干扰程序对于具有不同访存行为的宿主程序会有不同的效果。对于同一程序来说，其仿真行为较稳定，而对于具有不同访存带宽、访存读写比例不同的程序，可以通过调节干扰程序中的 Nop 指令密度达到的满意的效果。

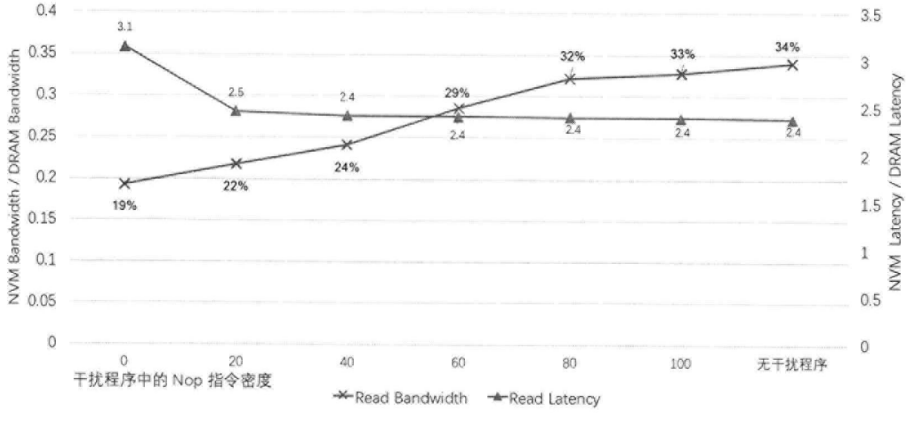


图 3.6 干扰程序对访存带宽、延迟的仿真影响

Figure 3.6 The effect of interference program on bandwidth and latency

在访存延迟仿真方面，除了 Intel QPI 造成的 2.5 倍左右物理访存延迟差

距，干扰程序亦能进一步压缩远端内存的访存延迟，如图 3.6 中的折线所示，当干扰程序中没有任何 Nop 指令时，NVM 端的访存延迟增长为了 DRAM 端的 3.1 倍。然而，干扰程序会同时造成延迟和带宽两方面的效果，相应的，此时的 NVM 端内存为 DRAM 端的 19%，而当向干扰程序中插入 Nop 指令来放松带宽压缩效果时，访存延迟会立刻恢复到 2.4 - 2.5 倍左右的效果。

而对于 NVM 读写不对称，即其写性能较读性能更差的问题，可以通过调整干扰程序中的读写比例来调整对内存读写不同的干扰效果。但本文并未能对此进行精准的控制。在接下来的工作中，我们将结合内存控制的的调度策略、程序的读、写访存行为影响，来提供支持读写不对称性的干扰程序。

3.2.4 本节总结

至此，本文开发了一套利用 NUMA 架构服务器仿真非易失性内存的平台。该仿真方式并不会受限于处理器的具体型号和内存的规格。而且，相比于基于 Trace 的模拟器，本文提出的仿真平台可以直接在其上运行复杂的程序，如分布式程序，或者基于运行时 (如 Java Virtual Machine) 的程序。本文所有的研究均是基于该平台进行仿真。

3.3 一种面向托管式应用的两级粒度数据划分方法

基于硬件和操作系统的异质内存管理策略多为单一粒度。比如，Rapp[12] 是一个利用硬件进行数据划分迁移，并利用 OS 维护页表映射关系的异质内存管理策略，其以页 (Page) 为粒度进行数据管理。而一些单纯利用内存控制器来进行监测和数据移动的研究，也是在一个固定的粒度在 DRAM 和 NVM 之间交互数据，如 [13] 是将 DRAM 作为 NVM 的缓存，并利用硬件控制在页 (Page) 粒度进行数据迁移。对于单机程序，这种单一粒度的管理模式可以取得令人满意的效果，例如，Qureshi 等人在研究 [13] 中宣称，在合适的数据替换策略下，在异质内存中只使用 3% 的 DRAM 作为缓存就可以达到和全部使用 DRAM 近似的性能。而对于单节点内存使用量在 TB 级别的内存计算应用来说，这种纯硬件组织方式的开销无法接受，如第 2.1.1 节中的分析，另一方面，数据的动态监测、迁移开销也会随着应用内存使用量的增大而迅速增加[5]。

有的研究提出了利用多级数据粒度的划分方式来降低数据的监测开销，如中科院，计算所的 Wei 等人[5] 提出了，可以首先利用离线程序分析的方式，根

据数据被创建的位置 (Allocation site) 来将其划分为冷、热、无法分辨三类。然后，再利用 Rapp 对“无法分辨”的数据进一步的划分和迁移。使用该种策略，其极大的降低了数据的动态监测和迁移开销。

由于本文是在操作系统之上的运行时系统来对数据进行冷、热划分，相对于硬件和操作系统可以获取更多的程序语义，也具有更灵活的数据划分、迁移粒度。对于基于托管式语言开发的应用，由于运行时系统的存在，[5] [10] 中提出的基于 Trace 的数据对象访存行为分析的方式难以实现。这些基于 C/C++ 程序提出的分析方式、工具不但难以获取托管式应用数据对象的地址，且由于 GC 会频繁的移动数据对象，更难以分析这些数据对象之上的访存行为。本文提出了利用运行时系统自身的特性来进行粗粒度的数据划分的方式，并针对托管式应用提出了全新的细粒度数据划分策略。最终，提出了一套两级粒度的数据划分方式，和利用 GC 进行数据迁移的完整异质内存管理框架。

本章的研究为整个工作的基础，我们首先在研究型 Java 虚拟机，JikesRVM 上进行了相关策略和方法的实现，并将该章的部分策略应用于了后续的研究中。数据的两级识别方式如图 3.7 所示。本章使用的测试用例为 Dacapo[60]、SPECjbb，不同测试用例的特点如表 3.2 所示。

表 3.2 Dacapo 和 SPECjbb2005 测试集

Table 3.2 Dacapo and SPECjbb2005 benchmarks

Workload	Description	Working set(MB)
antlr	Parser and translator generator	44.1
bloat	Java bytecode optimization and analysis tool	63.6
fop	Output-independent print formatter	43.3
jython	Interprets the Python benchmark	92.4
lusearch	Text search tool	397.7
sunflow	Photo-realistic rendering system	62.7
avrora	Simulates the AVR microcontroller	43.3
eclipse	Integrated development environment	499.1
hsqldb	Transaction processing	112.7
luindex	A text indexing tool	40.1
pmd	Source code analyzer for Java	119.7
xalan	Transforms XML documents into HTML	337.6
SPECjbb2005	Transaction processing	326.8

第一级：粗粒度的冷、热数据划分

根据第 3.1.2 节中的分析，运行时系统根据数据对象的类型和生命周期等特点对其进行了分类。JikesRVM 将不同的数据对象置于了不同的区域 (Space)，如存放短生命周期数据对象的新生代区域 (Young Generation Space/ Nursery Space)，存放大数组的 LOS 区域 (Large Object Space)，存放常规长生命周期的旧生代区域 (Old Generation Space / Mature Space)，以及元数据区域 (Meta Data Space) 和栈区 (Stack) 等区域。

虽然这些区域的建立是以数据对象的生命周期、种类、作用为标准进行的，但是经过本章的分析，不同区域之间的访存密度 (Memory Access Density) 相差悬殊。基于此发现，本章直接将各个区域划分为：热数据区 (Hot Space)、冷数据区 (Cold Space)、一般数据区 (Normal Space) 三种。其中，热数据区 (Hot Space) 将会被直接静态布局到 DRAM，相应的冷数据区 (Cold Space) 则会被直接静态布局到 NVM 。最后，一般数据区 (Normal Space) 中的数据需要做进一步的细粒度划分。

第二级：细粒度的冷、热数据划分

对于无法直接分类的数据区域，“一般数据区 (Normal Space)”，本章提出了一种基于函数进行细粒度冷、热数据划分的方法。在实验过程中，本章发现一个程序的多数访存仅由极少量的函数造成，如 pmd 中 0.4% 的函数造成了超过 50% 的访问行为，更重要的是，这其中的一些函数仅访问了为数不多的数据对象。基于此发现，本章提出了以函数为单位进行细粒度数据划分的策略，并开发了一套自动的离线分析工具。只需要将一个特定程序的字节码 (Byte Code) 作为输入，便可以输出冷、热数函数。最后，经过我们修改的 Java 虚拟机，JikesRVM 便可识别出这些函数访问的冷、热数据。

本章策略框架以及优势概要

图 3.7 展示了冷、热数据识别的两级架构。从图中可以看出，我们首先在区域 (Space) 粒度，对数据的冷热进行了划分和静态布局，这部分数据不再需要进一步的数据监测和移动。对于无法粗粒度划分的“一般数据区 (Normal Space)”，本章以函数为粒度从其中选出冷、热数据，并借助于 GC 机制，将其布局到对应的 DRAM 和 NVM 。最终，完成数据的两级粒度管理。

相比于 Wei 等人提出的基于 C/C++ 程序中数据对象生成位置 (Allocation

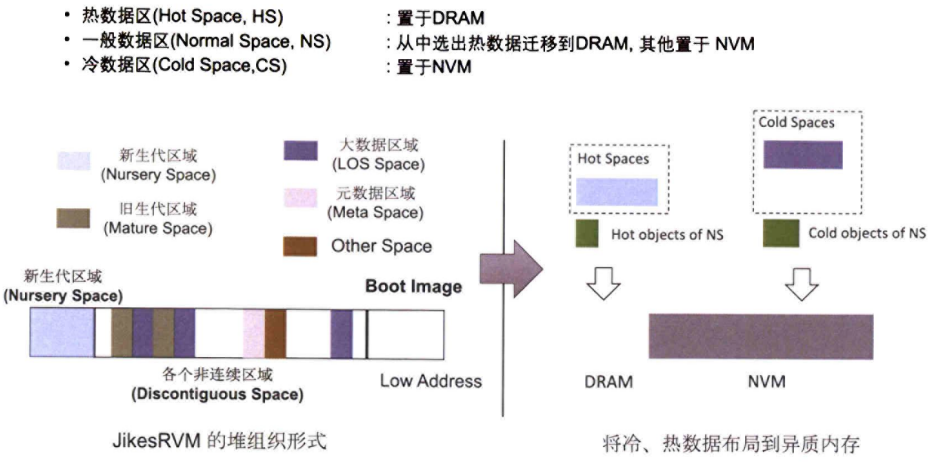


图 3.7　数据的两级识别和划分

Figure 3.7　The framework of data management in two granularity

site) 和 Rapp 的两级粒度数据划分[5]，本章的方法可以解决以上策略无法分析托管式应用数据对象访存行为的缺点，而且无需对现有服务器的内存控制器、操作系统内核进行任何修改。而对比于 Nakagawa 等人针对运行时系统提出的两级划分策略[56]，本文不但不对各个区域 (Space) 以访存密度 (Memory Access Density) 为标准进行了定量分析，更是在粗粒度层次将区域 (Space) 划分为了冷数据区域 (Cold Space)、热数据区域 (Hot Space)、一般数据区域 (Normal) 三类，而并非只有 [56] 中的热区域、一般区域两类。在细粒度划分层次，相比于 Nakagawa 等人利用 Write Barrier 机制监测每个数据对象之上的内存写次数作为迁移标准，本章的策略考虑了 Cache 带来的问题以及内存读密度对性能的影响。本章将在接下来的 3.4节、3.5 节中分别对粗、细两级粒度的数据划分策略进行详细介绍。

3.4　基于运行时堆特性的粗粒度冷、热数据划分

本节将阐述粗粒度数据划分的实现原理和实验分析。本文的关注点为通过合理的数据布局来优化程序在异质内存之上的性能，所以本文的冷、热数据划分标准为数据对程序性能的影响。如 Ramos、Bhattacharjee 等人在研究 [12] [74] 中所述，LLC Miss (Last Level Cache Miss)，也即末级缓存缺失，可以精准的描述出一个程序的性能。也就是说，如果一个程序访问某些数据时造成了大量的 LLC Miss, 那么我们便认为这些数据为该程序的性能关键数据 (Performance critical data)，也即本文所称的“热数据”。每一次 LLC Miss 均可导致实际的

物理访存，因此该值亦可作为物理访存行为的指标。相比于 LLC Miss 的绝对值，本文认为访存密度 (Memory Access Density) 是一个评价数据冷、热程度更合理的标注，其定义如公式 3.1 所示。该节便使用访存密度 (Memory Access Density) 在不同配置下评价各个区域 (Space) 的冷、热程度, 并通过程序实际时间加以验证。

$$Access\ Density = \frac{Physical\ Memory\ Access}{Memory\ Size} \quad (Read\ or\ Write) \qquad ... (3.1)$$

新生代区域大小对访存分布的影响。

运行时系统中，新生代区域 (Young Generation Space / Nursery Space) 的大小是一个动态可调的区域。同时，新生代区域的大小会显著影响 GC 的频率、开销，而其最佳值往往依赖于具体的程序计算行为。为了避免该值对本章程序分析的干扰，我们首先针对本章的测试用例，分析了不同新生代大小对其访存行为的影响，测试结果如图 3.8 所示。而其他区域 (Space) 的大小由程序的数据使用行为决定，并无法手动调节。

本节利用 NVM 的 NUMA 仿真平台来分析新生代区域大小对 LLC Miss 分布的影响。新生代区域被映射到 DRAM 之上，并将其他的区域映射到 NVM 之上，在运行时堆大小一定的情况下调整新生代区域的大小，并观测新生代数据造成的 LLC Miss 比例的变化，结果如图 3.8 所示。该图纵轴为 NVM 数据造成的 LLC Miss 数量和整体 LLC Miss 数量之比，横轴为新生代区域的大小，从中可以看到，当新生代区域逐渐增大时，NVM(其他区域) 中数据造成的 LLC Miss 比例逐渐降低。并可看到，对于本章所使用的 Dacapo、SPECjbb 测试集来说，新生代区域在 16MB 时，其造成的 LLC Miss 比例趋于稳定，如图 3.8 中的平均值曲线 (Avg) 所示。因此在本章随后的实验中，如无特别说明，新生代区域的大小被固定在 16MB 。

LLC (Last Level Cache) 压缩程序

本文使用的服务器缓存（Cache）相对于单机程序的工作集 (Working set) 来说偏大，无法反映出这些单机程序在 PC 机之上的真实访存行为。因此，在进行 LLC Miss 分析过程中，为了增加实验分析的普适性，本章研究了不同处理器的 LLC 大小对实验结果的影响。在本文使用的 Intel 多核心处理中，其一共有三级缓存，L1、L2、L3 Cache。前两级缓存（L1、L2 Cache）为每个核心

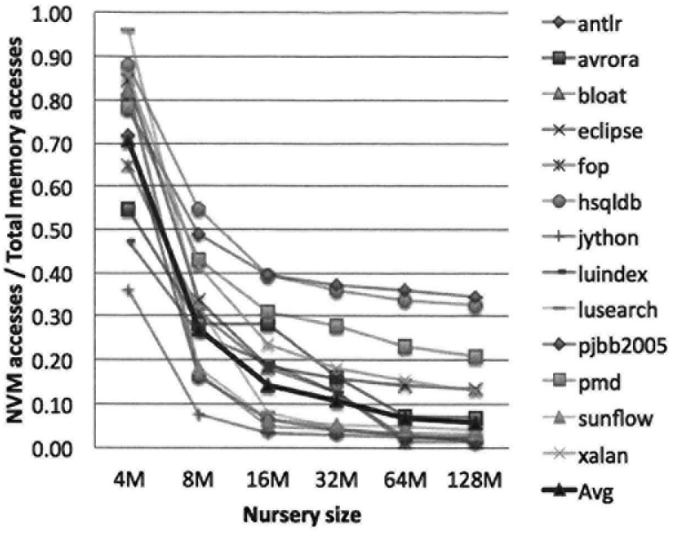


图 3.8　新生代大小对 LLC Miss 分布的影响

Figure 3.8　The LLC Miss distribution of different size of nursery

独享，而第三级缓存（L3 Cache），也即该处理器的 LLC，为所有核心共享。本文基于卡内基梅陇大学的 Ferdman 等人[75] 提出的 LLC 压缩方式开发了对应的程序来改变 LLC 的大小。

这里仍以上文所述的具有三级缓存的 Intel 处理器来解释该压缩程序的设计原理。该压缩程序的机制为，在一个多核处理器中，利用一个或者两个核心来执行压缩程序，该程序的访存特点为，几乎每次访存都会造成 L1，L2 Cache Miss，但是几乎全部命中 L3 Cache (LLC)。这样，由于压缩程序频繁的命中在 L3 Cache (LLC)，其中的数据几乎不会被替换到主存，达到了压缩该处理器 L3 Cache (LLC) 的目的。对于更多级或更少级的缓存，也是类似的原理，需要注意的是该机制仅适用于以 LRU 为缓存替换策略的处理器。在本章的实验采用该方法将处理器的 LLC 从 12MB 压缩到 4MB，并研究了各个区域上的 LLC Miss 分布。后续小节将对此进行详细介绍。

3.4.1　运行时堆各个区域中的访存分布

多级数据粒度的冷、热数据划分是降低数据分类开销的有效手段，然而由于运行时系统的存在，运行时系统外界的分析工具难以看到堆内部的数据对象的访存情况，导致传统 C/C++ 的数据分析方法[5][10] 均无法应用于托管式语言开发的应用。虽然 Nakagawa 等人在 [56] 中提出了可以粗粒度的将运行时堆

中的新生代区域直接映射到 DRAM，却其并未在真实运行时环境中测试该区域之上具体的访存行为 (如未考虑 Cache 对访存的影响)。另一方面，除了以数据对象生命周期为标准划分的新生代区域、旧生代区域外，还应进一步分析以数据对象类型、用途为标准进行划分的其他区域，如大数据区域 (Large Object Space, LOS)、元数据区域 (Meta Data Space)、栈 (Stack) 等区域，以便进一步减少细粒度数据划分的开销。

本节便在本文提出的基于 NUMA 架构提出的异质内存仿真平台上，对科研型 Java 虚拟机 JikesRVM 的堆区域 (Space) 进行了透彻的冷、热分析，并得出了粗粒度数据划分的策略。在实验过程中，根据 3.4 节中的分析，新生代区域的大小被固定在 16 MB。由于使用的单机程序测试集中，程序的工作集 (Working set) 不大，表 3.2 展示了各个程序的工作集。因此，为了避免服务器超大缓存容量 (12MB Last Level Cache) 对程序性能的影响，本研究使用3.4 节中的缓存压缩程序，将 LLC 压缩到了 4MB。

LLC Miss 的分布。

图 3.9 展示了各个区域 (Space) 中的 LLC Miss 分布。绝大多数的 LLC Miss 由新生代数据造成，平均值为 76%，其访存密度 (Memory Access Density) 是旧生代数据的 12.5 倍。这是由于在运行时系统中，所有新生成的数据都会首先分配进入新生代区域，并大概率在新生代区域中被 GC 回收。因此，造成了新生代区域所占用的内存被高效的反复使用。

对于不同应用来说，旧生代数据造成的 LLC Miss 差异较大。其中，对于 SPECjbb2005 来说，有将近 45% 的 LLC Miss 命中了该区域，而对于 jython 来说，只有不到 5% 的 LLC Miss 命中了旧生代区域，平均来看，旧生代数据造成了 13% 的 LLC Miss。该现象的原因在于，只有生命周期较长的数据对象才会迁移到旧生代区域, 而不同程序的计算行为不同，所以数据对象的存活率也就不尽相同。另外，在固定的新生代区域大小下，每个程序的工作集不同，自然也会导致了被迁移到旧生代区域中的数据量不同。当旧生代区域中的数据量较少时，其上也就很难发生 LLC Miss。比如，SPECjbb2205 的工作集可以达到 330MB, 其旧生代区域的使用最高可达 273MB，所以，所以程序造成的 LLC Miss 的 45% 位于旧生代区域。而 antlr 的工作集只有不到 45MB, 在监测中，其旧生代区域最多只会被使用 3.6MB，自然导致几乎没有 LLC Miss 发生在其旧

生代数据上。

虽然元数据区域和栈中的数据会被频繁的访问，然而由于这些区域 (Space) 使用的内存空间较小，大部分的据都被预取到了 Cache 中，并不会造成过多的 LLC Miss。而对于大数据区域 (LOS, Large Object Space) 来说，由于其中的数据以大数组为主，对这些数据的访问多以顺序遍历形式进行，而该种访存模式，由于软、硬件预取等机制的存在，并不会频繁的造成 LLC Miss。

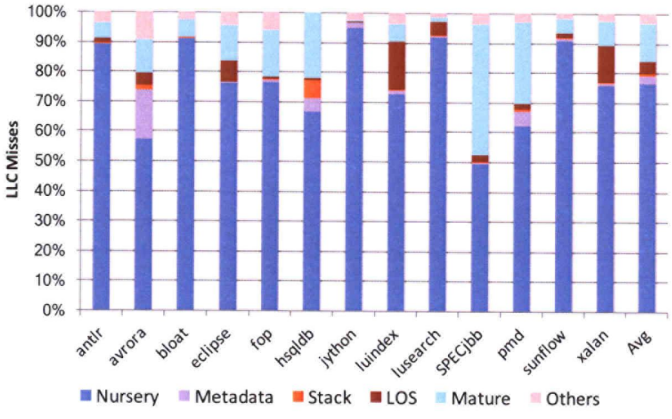


图 3.9　运行时堆中的 LLC Miss 分布，16MB 新生代，4MB 三级缓存

Figure 3.9　The LLC Miss distribution in each heap space, 16MB Nursery space, 4MB LLC

为了进一步验证该结论，本节在 32MB 新生代区域大小，默认的 12MB LLC 下重复了以上实验。从图 3.10 中可以看到，加大新生代区域后，avrora 和 luindex 两个应用新生代数据造成的 LLC Misses 显著增加，比如，新生代区域从 16 MB 增加到 32MB 后，对于 avrora 和 luindex 两个应用，其新生代数据造成的 LLC Miss 从 57% 和 72% 增加到了 83% 和 89%。然而，对于其他应用来说该变化不大。同时，LLC 在 12MB 和 4MB 两种配置下，虽然 LLC Miss 在各个区域 (Space) 中的分布有一些变动，但是元数据区域、栈区域均未造成频繁的 LLC Miss，其数据仍旧处于 Cache 之中。

LLC Store Miss 的分布。

NVM 具有读写不对称性，相比于读性能，其写性能较差，而且写功耗更大，因此一些功耗敏感应用场景，倾向于将大量的写迁移到 DRAM [5]。经过前述分析可知，对于 Dacapo、SPECjbb 等单机应用，其 LLC Miss 主要分布在新生代区域和旧生代区域，所以这里进一步分析了两个区域中的 LLC Store Miss

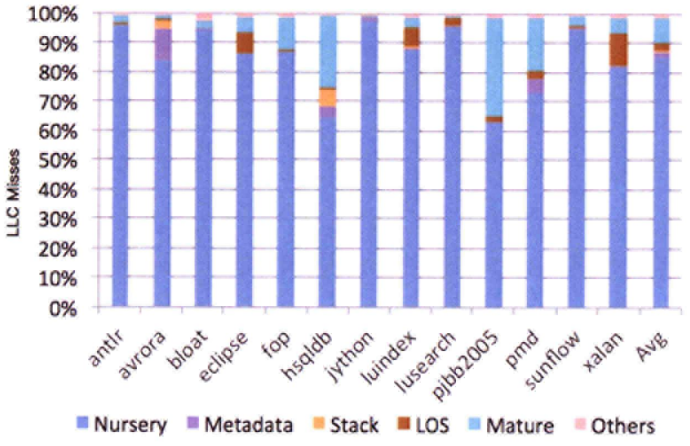


图 3.10　运行时堆中的 LLC Miss 分布，32MB 新生代，12MB 三级缓存

Figure 3.10　The LLC Miss distribution in each heap space, 32MB Nursery space, 12MB LLC

分布情况。这里的实验仍旧将新生代区域固定在 16MB，LLC 固定在 4MB 进行。

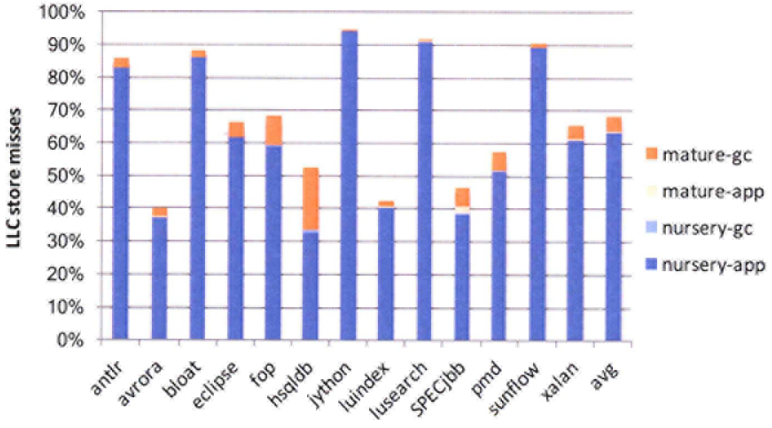


图 3.11　LLC Store Miss 在新生代和旧生代中的分布

Figure 3.11　The distribution of LLC Store Miss in Nursery and Mature Spaces

图 3.11 展示了 LLC Store Miss 在新生代区域和旧生代区域中的分布，同时，该图也展现了由于应用 (app) 和 GC 造成的的 LLC Miss 分布。通过该图可知，多数的 LLC Store Miss 仍旧主要发生在了新生代区域，而且多是由于应用的执行导致的。这是由于数据对象默认分配到新生代区域，而数据的初始化过程中会发生大量的写操作。对于旧生代区域而言，其覆盖的 LLC Store Miss 主要由 GC 行为导致，这是由于将数据从新生代区域迁移到旧生代区域过程中导

致的写操作，并无法通过调整数据布局来进行消除。

LLC Miss 分布情况总结。

经过以上分析，可以得出一个针对 Dacapo、SPECjbb 等单机应用，在不同硬件结构中具有一定普适性的数据划分结论。相对于 Nakagawa 等人提出的将新生代区域映射到 DRAM，并堆其他区域中的数据进行细粒度冷、热划分的策略[56]，本节基于 NUMA 架构的异质内存真机仿真平台进行了定量分析，并提出了进一步的粗粒度划分来减少需要细粒度冷、热划分的数据量。

首先，对于根据数据对象生命周期不同而划分的新生代区域和旧生代区域来说，本节通过不同配置下的 LLC Miss 定量分析得出，新生代数区域中的 LLC Miss 是旧生代区域的 12.5 倍，并且该区域内集中了程序中主要写操作；而对于存放以大数组为主的大数据区域 (Large Object Space, LOS)，其上的 LLC Miss 分布并不大；对于元数据区域、栈等区域，虽然直观上通常认为其上的访存较多，但由于其空间较小，其数据会位于 Cache, 并未造成明显的 LLC Miss；而 LLC Miss 在旧生代区域之上的分布随应用变化而显著变化，其中的数据需要进一步的细粒度的划分。

3.4.2 运行时堆各区域对性能的影响

经过上一节的分析可以看到各个区域 (Space) 上的 LLC Miss 分布，以及基于此得到的初步区域 (Space) 划分。本节将各个区域 (Space) 逐一的从 DRAM 迁移到 NVM 并观察程序性能的变化，来进一步的证实前述的结论。

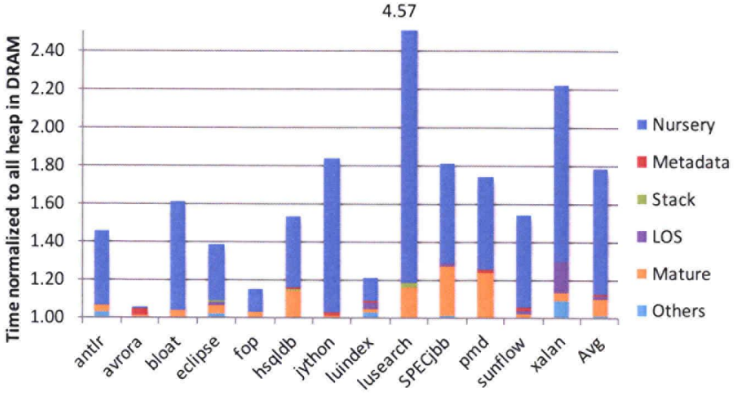


图 3.12 各个内存区域 (Space) 对性能的影响

Figure 3.12 The execution time effect of each heap space

图 3.12 展示了，逐渐从 DRAM 迁移到 NVM 后性能的变化，其纵轴是将各个程序的执行时间与程序在 DRAM 上的运行时间之比。从图中可以看到，新生代数据确如前述分析主导了应用性能，而对于 SPECjbb、lusearch、pmd 应用来说，旧生代数据也对性能起到了较大的影响作用。但是，由于该区域所使用的内存区域较大，并不能直接将该区域直接布局到 DRAM，我们将在后续的3.5 节对该区域进一步的讨论分析。

本节结论和 3.4.1 节中 LLC Miss 分布的分析结论基本一致。至此，可以得出结论，针对 Dacapo、SPECjbb 这一类单机应用来说，运行时堆对数据的划分可以在一定程度上区别出数据的冷、热。在进行细粒度的数据划分、迁移之前，可以利用该结论对数据做静态的粗粒度布局。如，将新生代区域直接映射到 DRAM，而将元数据区、栈直接映射到 NVM，无需对这些区域内部的数据做进一步的数据划分。

3.4.3 粗粒度的静态数据布局

经过以上分析，最终得出结论：虽然运行时堆中各个区域 (Space) 的组织并非是出自数据访存行为的特点，但是仍将可以利用这种组织形式作为数据冷、热划分的标准。Nakagawa 等人在 [56] 中亦针对托管式应用提出了两级划分策略，其将运行时堆粗粒度的划分为热区域、一般区域，并仅对一般区域中的数据进行细粒度划分的策略。而本章提出的策略，进一步的将运行时堆中的区域划分为了热数据区、冷数据区、一般区域三类，该策略不但利用了运行时堆中的新生代、旧生代划分，并提出亦无需进一步细粒度划分大数据区、元数据区、栈等区域，进一步减少了需要进行细粒度划分的数据量。另一方面，该章节基于异质内存真机仿真平台，在不同 LLC 大小、新生代区域大小等配置下，对各个区域的 LLC Miss 分布进行了定量分析、性能分析，充分的验证了粗粒度划分的效果。而 Wei 等人在 [5] 中提出的粗粒度划分策略并不能适用于托管式应用。经过以上分析，我们认为，对于 Dacapo 、SPECjbb 这类单机托管式应用：

- 具有大量短生命周期数据的新生代区域，由于其空间频繁的被 GC 释放、重新利用，因此具有很高的使用效率。其中的数据导致了一个程序主要的 LLC Miss 和大部分 LLC Store Miss。因此，可以将其视为一个“热数据区 (Hot Space)”，并将其静态映射到到 DRAM，而无需对其进

一步的进行细粒度的数据识别和划分。

- 大数据区 (LOS, Large Object Space)、元数据区 (Meta Data Space)、栈结构 (Stack) 等区域因为不同的原因，并不会造成较多比例的 LLC Miss, 对整个程序执行时间的影响较小。因此，可以直接将这些区域标记为“冷数据区 (Cold Space)”，并将其直接映射到 NVM。
- 虽然旧生代区域 (Mature Space) 在很多应用造成了较多的 LLC Miss，但是由于其占据的内存空间较大，因此需要对其进一步分析，选出其中的热数据，并迁移到 DRAM。基于此，我们将该区域列为留待进一步处理的“一般数据区 (Normal Space)”。

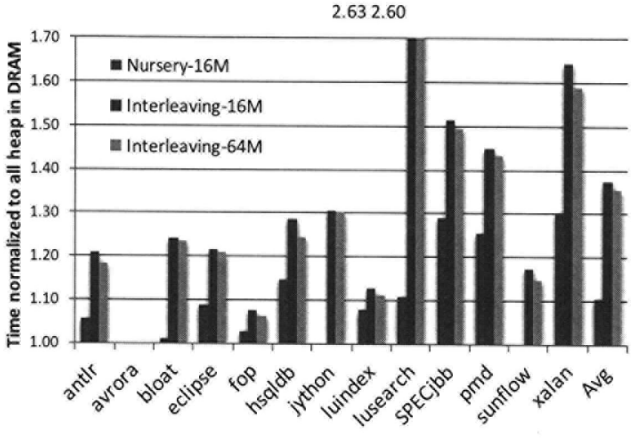


图 3.13　将新生代映射到 DRAM 与无任何管理情况下的性能对比

Figure 3.13　time comparison between nursery in fast-space and OS interleaving allocation

图 3.13 展示了将热数据区 (Hot Spaces)，此处为新生代区域 (Nursery)，映射到 DRAM，其他区域均在 NVM 时的性能。蓝色柱子展示了将 16MB 大小的新生代 (Nursery Space) 置于 DRAM，其他区域 (Space) 置于非易失性内存时的运行时间，红色柱子和绿色柱子分别展示了，当异质内存中有 16MB DRAM 和 64MB DRAM，但没有任何布局策略管理时的执行时间。从图中可以看到，粗粒度的数据映射便可以得到较好的性能。对于 Dcapo、SPECjbb 等单机应用来说，新生代区域映射到 DRAM 后，和全部执行在 DRAM 上相比，平均只有 10% 的性能差距。在没有任何管理的情况下，和全部使用 DRAM 的性能差距在 38%，即使将异质内存中的 DRAM 大小上升到 64MB，也和全部运行于 DRAM 有将近 35% 的性能差距。因此可知，粗粒度的数据映射策略对于单机

应用，Dacapo、SPECjbb 测试集来说具有良好的效果。

3.5 基于函数的细粒度冷、热数据划分

在前述章节中，基于运行时堆的组织结构将堆区域 (Space) 划分为了：热数据区 (Hot Space)、冷数据区 (Cold Space)、一般数据区 (Normal Space) 三类。其中，热数据区 (Hot Space)、冷数据区 (Cold Space) 分别被直接映射到 DRAM 和 NVM，而一般区域 (Normal Space) 中的数据需要进一步的细粒度划分。

基于操作系统、硬件的传统的细粒度异质内存管理策略会和运行时系统中的 GC 行为产生数据布局冲突，如图 1.2 所示，因此，本节采用程序分析的方式来对运行时堆的特定区域 (Space) 中的数据进行细粒度的冷、热划分。而常规的 C/C++ 数据访存分析方式，如 Dulloor 等人在 [10] 中提出的利用 Trace 分析不同数据结构之上访存行为来进行冷、热划分的方法，并不适用于托管式应用。这是因为运行时系统中的 GC 机制会频繁的迁移数据对象，外部工具难以统计其上的访存信息。而 Inoue 等人提出一种易于造成 LLC Miss 的代码特征 (Code pattern)，并提出了可以利用 JIT 编译器通过 Pattern Match 的方式识别出该代码特征的方法。经本文在 Dacapo、SPECJjbb 等单机应用中广泛的测试，该代码特征的效果并不明显，而且，该方式并不能针对特定的堆区域 (Space) 中的数据进行冷、热划分。针对以上难点，本节提出了一种基于函数的，细粒度冷、热数据划分方式。其不但可以针对特定的堆区域 (Space) 中的数据进行有针对性的分析，而且可以考虑到 Cache 的存在，直接统计相应数据的物理访存行为。

在程序分析过程中，我们发现一个程序虽然会执行成千上百的函数，但是绝大多数的 LLC Miss 均由少数的特定函数造成。例如，pmd 中 0.4% 的函数造成了其 51% 的 LLC Miss, 而 hsqldb 中 1% 的函数则造成了其 54.8% 的 LLC Miss, 基于该发现，本节提出了以函数为粒度的冷、热数据划分方式。并开发了一个名为 HMprof 的离线分析工具来分析函数的 LLC Miss 特征，并从中选出造成访存密度 (Memory Access Density) 较大的函数，该工具的执行过程如图 3.14 所示。

这些选出的函数被称为热函数 (Hot Methods)，其访问的数据对象被称为热数据 (Hot Objects)。最后，本文利用 JIT 编译器来标记热函数 (Hot Methods) 访问的热数据 (Hot Objects)，并利用 GC 机制来将被标记的热数据迁移到的

DRAM 区域。本节最后对该理论进行了验证，基于该理论选出的热函数，其访问的数据造成了密集的 LLC Miss，是整个程序平均 LLC Miss 的 3 倍以上。将这些数据迁移到 DRAM 后，会带来显著的性能提升。

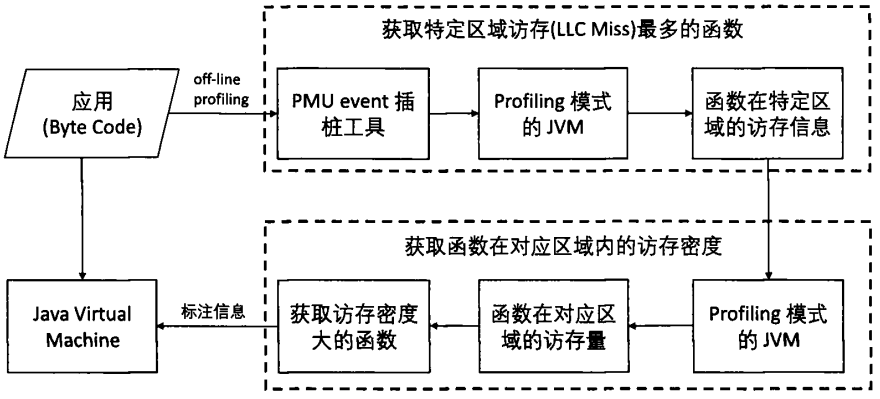


图 3.14　从待定区域中选出热函数及其访问的热数据的过程

Figure 3.14　Select the Hot Methods and the accessed Hot objects from the Normal Space

3.5.1　获取函数在特定区域的访存信息

为了基于函数的粒度来识别出特定堆区域 (Space) 中的冷、热数据，需要统计函数对应的访存密度 (Memory Access Density) 信息，也即需要获取该函数在对应堆区域 (Space) 中造成的 LLC Miss 和 Memory footprint 信息。对于托管式应用而言，一个函数访问的数据对象可能散布在整个运行时堆的不同区域中，而且频繁的被 GC 移动，针对该问题，本节提出了利用基于 NUMA 的异质内存仿真平台来获取函数在特定堆区域 (Space) 的方法：首先，将需要分析的堆区域 (Space) 置于 DRAM，而将其他的区域 (Space) 置于 NVM。然后，向需要分析的函数中插入想要统计的事项 (Performance Counter Event)，如 LLC Miss。最后，便可通过离线分析 (Off-line profiling) 的方式获取函数在特定内存区域的 LLC Miss 信息，也即物理访存信息。由于 NUMA 架构是当前主流的服务器架构，因此该方式具有较好的普适性。

每个托管式应用对应了成千上万的函数，而且每个函数又可能会因为“函数调用”、“返回”等方式离开其执行过程，另一方面，应用使用者可能无法直接获取应用的源码，因此，手动完成 LLC Miss 事项的插入工作并不现实。针对以上问题，本文基于二进制插桩工具 ASM[76] 开发了一套针对 Byte Code 文件，对函数访存信息进行分析的工具，HMProf。其主要作用为对一个 Class 文件中

的函数进行插桩操作，在每一个函数的入口 (entry point) 和出口 (exit point) 均插入了采集 Performance Counter 信息的函数。向函数的入口 (entry point) 插入的代码如图 3.15(a) 所示，首先，要向其中插入一个临时的局部变量用来存储指定的 Performance Counter 当前数值，然后，在函数退出、下一个函数调用之前，再次读取指定的 Performance Counter 的值，如图 3.15(b), 并将两次读取数值之间的差值记录在函数序号，576, 对应的全局变量中。Java 程序多以多线程执行来提高并发性，因此 HMProf 在读取指定的 Performance Counter 值时以线程为单位。

```
/** read counter 0, save
to local variable e.g. 3 **/

iconst_0
invokestatic perf.readPerf(I)J
lstore_3
```

(a) Inserted instructions before entry point

(a) 函数入口插入的指令

```
/** Update local variable
3 for method 576 **/
iconst_0
sipush  576
lload_3
invokestatic perf.updatePerf(IIJ)V
```

(b) Inserted instructions before exit point

(b) 函数出口插入的指令

图 3.15 HMProf 向一个函数中插入的信息

Figure 3.15 The instrumentation in a method by HMProf

HMProf 对一个函数的指令插入点如图 3.16 所示，一个函数的入口可以是函数的开始、内部调用的返回等，如图 3.16 中绿色标注所示。此时，都需要向该函数插入图 3.15(a) 中的指令，来读取对应的 Intel Performance Counter 值。而一个函数的出口不只包含最后的“返回 (return)”，也包含函数调用前的跳转，如图 3.16 中红色标注所示。此时，需要再次读取对应的 Intel Performance Counter 值并和入口的值做差值，将获取的差值写入函数对应的全局变量。经过这些插桩操作，便可监测一个函数的代码所造成的所有特定事项 (Performance Counter Events)，如 LLC Miss，Store Miss，Load Miss 等。

因为本文是在 NUMA 架构的服务器上进行对应的实验分析，所以监测的事项 (Performance Counter Events) 支持区分访存发生的位置，该设计是为了支持分析特定的运行时堆区域 (Space) 上的访存情况。如，将程序固定在 CPU #1 上执行，并将需要分析的旧生代区域固定在 CPU #1 对应的本地内存 (Local Memory)，而将其他的区域 (Space) 固定到远端内存 (Remote Memory)，那么此时通过指定的 Performance Counter 获取的事件即为这些函数在旧生代区域上造成的 LLC Miss。目前，该工具仅集成了表 3.3 中所列的事项。此时已经可以

```
Method entry:
=> insert execution entry point instructions

        other instructions

=> insert execution exit point instructions
invoke Method#1
=> insert execution entry point instructions

        other instructions

=> insert execution exit point instructions
Method exit.
```

图 3.16 HMProf 对一个函数进行的插入点

Figure 3.16 The instrumentation point for a method by HMProf

满足本研究的分析，如果有后续需求，我们也可以扩展该工具支持更多的硬件检测。

表 3.3 HMProf 支持的硬件监测事项

Table 3.3 The performance counter events supported by HMProf

Performance Counter Index	Description
0	Local LLC Miss for current thread
1	Local Store Miss for current thread
2	Local Load Miss for current thread

通过以上方式，便可分析程序中所有函数在特定堆区域 (Space) 上的 LLC Miss 信息，经过分析，一个程序的 LLC Miss 仅仅集中在少量的函数中，如 0.4% 的 pmd 函数和 1% 的 hsqldb 分别导致了 51% 和 54.8% 的 LLC Miss。因此，接下来仅需要针对极少量的函数进行 Memory footprint 信息分析，便可从中选出具有高访存密度 (Memory Access Density) 的热函数 (Hot Methods)。在后续测试中，对每个程序一般选取了不超过 32 个造成 LLC Miss 最多的函数作为候选热函数 (Hot Method Candidates) 来进一步分析。

3.5.2 获取函数的 Memory Footprint 信息

经过前述分析，我们获取了少量在特定区域 (Space) 造成众多 LLC Miss 的函数作为候选热函数 (Hot Method Candidates)。但是，这些函数中有诸多用于数据结构遍历的函数，如 next()、prev() 等，其访问的数据量巨大，并因此导

致了很多 LLC Miss，并不能认为其访问的数据都是热数据。因此，本节需要分析候选热函数 (Hot Method Candidates) 的 Memory footprint，并按照公式 3.1 来进行访存密度 (Memory Access Density) 分析。这里的步骤可以概括为：

- 将运行时系统 (OpenJDK) 切换为离线分析模式，此时每一个数据对象 (Object) 都会被扩展出一定数量的标记位 (method bits)，如 32 bits，用做函数的索引，每个被分析函数对应一个 bit。
- 以3.5.1 节中分析出的候选热函数 (Hot method candidates) 为输入，JikesRVM 会给每个函数以索引序号，如 0 - 31。同时 JikesRVM 会利用 JIT 向这些函数中插入指令，来将对应的索引序号写入这些函数访问的数据对象 (Object)。
- 最后，在程序结束时，统计特定区域，如旧生代区域 (Mature Space)，中每个函数索引对应的数据对象的总大小。

由于本节需要统计一个函数在特定区域 (Space)，如旧生代区域 (Mature Space)，所具有的 Memory Footprint 信息，其必须要借助于修改运行时来完成该问题。本节给 Java 数据对象模型 (Java Object Model) 增加了一种新的离线分析模式 (off-line profiling model)，该种模式下，所有数据对象 (Object) 的 Header 部分都被扩展出了 32 bits 的大小用于信息记录，其结构如图 3.17 所示。根据 3.5.1节的分析，虽然一个托管式应用通常包含数千个函数，但是主要的 LLC Miss 集中在了不足 1% 的极少量函数中。另一方面，即使被扩展出的标记位不足，我们也可以对其进行迅速的扩展，因此该设计并不会带来扩展性问题。

至此，将按以下过程来自动获取每个程序中的热函数 (Hot methods)：首先，利用 3.5.1 节中的离线分析工具 HMProf 选出被分析应用在特定堆区域 (Space) 造成 LLC Miss 最多的 32 个函数作为候选热函数 (Hot method candidates)。并按照 LLC Miss 总量顺序排列为 0 - 31 作为其索引序号，每一个函数对应数据对象中扩展出的一位。

然后，利用 JIT 向选出的候选热函数 (Hot method candidates) 中插入标记指令，其向函数中插入的指令和 § 3.5.3中介绍的相同，然而标记的位置和值不同。插入的指令会根据函数的索引序号来标记函数访问的数据对象 (Object) 中对应的标记位 (method bit), 如图 3.17 所示。为每一个函数都设置单独的标志位是因为这些函数可能会访问同样的数据对象 (Object)，一个数据对象会被多个

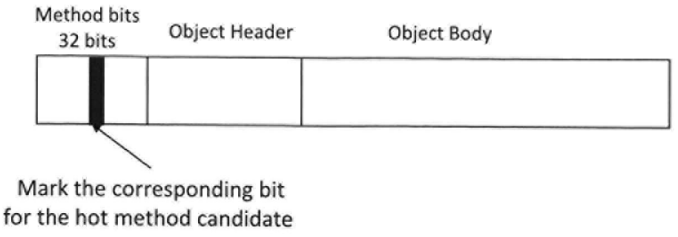


图 3.17　处于离线分析模式的 Java object model

Figure 3.17　The extended Java object model for off-line profiling

函数同时标记。

最后，需要统计特定堆区域 (Space) 中数据对象被标记的情况，下面以旧生代区域 (Mature Space) 为例进行解析。旧生代区域 (Mature Space) 中的数据均是新生代区域中的数据经过 MinorGC 移动到其中的。一旦发生 MajorGC (Full-GC)，旧生代区域 (Mature Space) 中的数据可能会被回收，此时无法准确统计一个函数访问的数据有多少被迁移到旧生代区域 (Mature Space) 中。为了避免该问题，在进行分析的过程中，整个运行时堆 (Java Heap) 被设置为一个较大值，避免 MajorGC(Full-GC) 的发生。当程序结束时，去扫描一遍旧生代区域 (Mature Space) 中的数据，并统计各个函数索引序号对应的数据量。

至此，便可以根据公式 3.1 计算出候选热函数 (Hot method candidates) 中真正的热函数 (Hot methods)。这些函数访问的数据量较少，但是却造成了数量众多的 LLC Miss。通过对这些函数做了进一步的分析，发生这些函数的访存行为具有一定的相似性。比如，pmd 中造成 LLC Miss 最多的几个函数都是在集中访问一个树形结构 (Tree Structure), 而 SPECjbb 中造成访存的数据结构为一些 Object Array，在托管式应用中，这些结构均由大量的数据对象 (Object) 构成，而数据对象之间通过 Reference 连接。由于运行时系统并无法感知这些独立数据对象之间的语义关系，在 GC 过程中，会将同一个数据结构的数据对象散布到不同位置。因此，当一些频繁执行的函数访问这些数据结构时，自然就会造成大量的 LLC Miss。而本文提出的基于函数粒度的冷、热数据划分方式，会找到访问这些数据结构的高频函数，并最终选出这些热数据。

Dulloor 等人曾在研究 [10] 中提到，Pointer-chasing 形式的访存行为对内存延迟敏感，因此建议置于异质内存的 DRAM 中，并针对 C/C++ 应用提出了基于 Trace 的分析方法来寻找该类数据结构，然而，该策略并不适用于托管式

应用。首先，由于托管式应用中的数据对象 (Object) 会被 GC 频繁的移动，难以用 [10] 中提出的基于 Trace 的策略来分析每个数据对象 (Object) 之上的访存信息。其次，托管式应用中，一个数据结构由大量独立的数据对象 (Object) 构成，而且这些数据对象 (Object) 并不一定会聚集在一起，更难以在数据结构粒度统计其上的访存信息。而本文提出的基于函数粒度的分析方法和工具则可以良好的解决托管式应用的这些问题，寻找出具有高访存密度 (Memory Access Density) 的热数据。

3.5.3 利用 JIT 标记函数访问的数据

针对选出的热函数 (Hot Methods)，我们需要再次标记出其访问数据，并将这些数据迁移到旧生代 (Mature Space) 的 DRAM 区域。该标记过程仍是利用即时编译器 (Just In Time Compiler, JIT) 完成的。JIT 的目的是在程序执行过程中，对高频函数进行深度的优化，包括循环展开、循环切块等主流优化。因此，其具有修改执行程序的能力。本节便利用其该特性对第 3.5.2 节中选出的热函数 (Hot Methods) 访问的数据进行标注工作。

```
getfield    Result(RefType) = ObjRef, Offset, field
byte_load   t0(B) = ObjRef,HOT_BIT_OFFSET
int_or      t1(B) = t0(B),HOT_BIT_MASK
byte_store  t1(B), ObjRef,HOT_BIT_OFFSET
```

(a) getfield

```
null_check  Result(GUARD) = Result(RefType)
byte_load   t0(B) = Result(refType),HOT_BIT_OFFSET
int_or      t1(B) = t0(B),HOT_BIT_MASK
byte_store  t1(B),Result(refType),HOT_BIT_OFFSET
```

(b) null_check

图 3.18 利用即时编译器来标记热函数访问的数据对象

Figure 3.18 Mark the objects accessed by Hot Methods via JIT

为了尽快布局热函数访问的数据对象到 DRAM 区域，需要其访问的数据在热函数开始执行时即被立刻标记，但 JIT 的触发策略却需要对执行程序进行一定时间的观测。因此，我们修改了 JIT 的触发机制，在程序开始执行时，即以第 3.5.2节中输出的热函数 (Hot Methods) 为输入，触发 JIT 立刻对其访问的数据对象进行相应的标注。在实验过程中，并没有计算即时编译器 (JIT) 的标记时间，但是却包含了插入语句执行的时间。由于即时编译器 (JIT) 的标记过程是和执行无关的，因此其程序的整体运行时间影响不大。

一个 Java 程序中有多种数据，例如在栈 (Stack) 中的临时变量 (local variables)、静态区的全局变量 (stack variable) 等，还有在运行时堆 (Java Heap) 中的数据对象 (Object)。本文只关注堆中的数据对象 (objects)，因为其消耗了主要的内存空间，即使在 C/C++ 类型的程序中也有类似的结论[5]。函数在执行过程中，会以多种形式访问数据对象 (Object)，在对其 Object Header 进行标记前，需要确定数据对象是存在的，否则会造成程序的崩溃。因此，本文选择了 getfield, putfield, object null_check, Function Invoke 等操作之后，对可以正常解析的数据对象 (Object) 进行标记操作。即 JIT 插入的标记指令如图 3.18 所示，该图展示了 getfield 和 object null_check 的标记过程。首先获取一个已经解析过的数据对象的 (Object) 的首地址，然后利用 HOT_BIT_MASK 将对应的 Hot Bit 标记为 1。因为并没有必要区分热数据来自于哪个特定的函数，因此只需要 1 bit 作为标志来区分一个特定的数据对象 (Object) 是被迁移到 DRAM，还是被迁移到 NVM 。我们从原有的 Java Object Header 中压缩出了 1 bit，并没有增加额外的空间开销。

至此，已经完成了对待定区域 (Normal Space) 中冷、热数据的细粒度数据划分，并通过 Object header hot bit 予以标注。接下来，GC 在移动数据的过程中只需要根据数据的标记情况将数据置于对应的 DRAM 或 NVM 便可。

3.6 利用垃圾回收机制布局数据以及效果分析

3.6.1 利用 MinorGC 来移动数据

JikesRVM 和 OpenJDK 8 均采用了 Generation GC 的策略。但是二者具体的实现方式又略有不同。在 JikesRVM 中，只要一个数据对象 (Object) 可以成功的存活过一次 MinorGC，那么其便会被迁移到旧生代 (Mature Space)。而在 OpenJDK/Oracle HotSpot 中，只有当一个数据对象在 MinorGC 中多次存活，其才会被迁移到旧生代 (Old Generation, Object Space) 中。本研究便是利用 MinorGC 的这次数据迁移来将已经划分好的数据对象 (Object) 布局到对应的 DRAM、NVM 区域。作为后续研究的基础，本研究是基于 JikesRVM 的一个探讨性工作，因此，这里仍旧使用 JikesRVM 的 GC 机制为例来进行分析。但是，我们在这里获取的策略，仍旧适用于后续章节中基于 OpenJDK 8 的研究。

图 3.19 展示了一次 MinorGC 过程中，对需要迁移到旧生代 (Mature Space) 中的数据进行识别，并分别布局到该区域 (Space) 对应的 DRAM 部分和 NVM

部分的过程。第3.5.3节中描述的机制会向热函数 (Hot Methods) 中插入对应的标记语句，然后当该函数执行时，插入的语句便会标记其访问数据的 Object Header 中的 Hot Bit 。在此之后，一旦发生 MinorGC，那么被标记的数据对象便会迁移到旧生代 (Mature Space) 的 DRAM 区域。

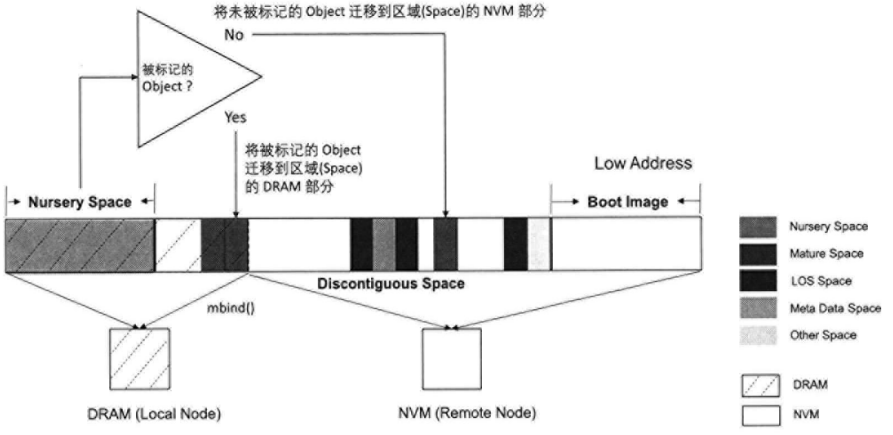


图 3.19　利用 MinorGC 来布局划分的数据

Figure 3.19　Utilize MinorGC to place the data layout

在异质内存中，DRAM 部分所占的比例并不高。当 DRAM 不足时，剩余的数据将会被置于旧生代 (Mature Space) 的 NVM 区域。只有当触发 MajorGC，释放掉 DRAM 区域内无用的数据后，后续标记的数据才会继续被 MinorGC 置于旧生代 (Mature Space) 的 DRAM 部分。需要注意的是，一个数据可能仅仅在程序执行过程中的某些时刻是热的 (Performance Critical)，但是使用本方法并无法做到灵活的将其换入、换出 DRAM 区域。而且，静态编译分析的方式也无法分析出其会在哪个时刻变为热数据，哪个时刻变为冷数据。这些问题，我们将会在后续的研究过程中尝试去解决。

3.6.2　性能分析

本节根据 § 3.4.3 中的分析，选出了经过粗粒度映射后，和全部使用 DRAM 相比仍旧有性能差距的三个应用 SPECjbb、pmd、hsqldb 进行了细粒度的识别和迁移。图 3.20 展示了利用 MinorGC 迁移 SPECjbb、pmd、hsqldb 三个应用中的热函数 (Hot Methods) 所访问的数据到 DRAM 后，旧生代 (Mature Space) 中 DRAM 部分所覆盖的 LLC Miss 的变化。横轴表示旧生代 (Mature Space) 所使用的 DRAM 比例，纵轴表示旧生代 (Mature Space) 中的 DRAM 上所覆盖的 LLC

Miss 比例。绿色方块展示了 SPECjbb 的覆盖情况，当迁移 Access Density 最高的 12 个热函数 (Hot Methods) 访问的数据到 DRAM 时，仅使用了 6% 的 DRAM 空间，却覆盖了 31 % 的 LLC Miss。进一步迁移后续的热函数 (Hot Methods) 访问的数据，最终在旧生代 (Mature Space) 使用 12% 的 DRAM 时，其覆盖了 36% 的 LLC Miss。该应用对应的数据迁移趋势是一个斜率逐渐变小的过程，表示后续热函数 (Hot Methods) 覆盖的数据的 Access Density 逐渐降低。

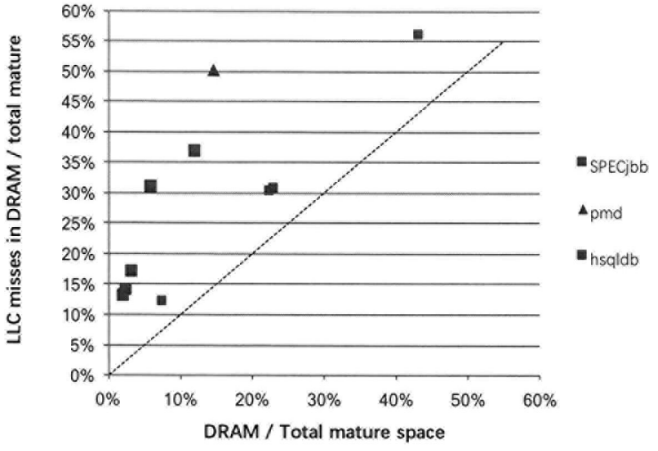


图 3.20　热函数访问的数据所覆盖的 LLC Miss

Figure 3.20　The LLC Miss covered by the objects accessed by Hot Methods

pmd 的行为和 SPECjbb 不同，如蓝色三角所示，其 Access Density 最高的前十个函数访问的均为同一个数据结构。经过分析，该结构为一个树形结构 (Tree Structure)，不同的函数通过遍历该结构并进行不同的计算。当将该结构迁移到 DRAM 后，其旧生代 (Mature Space) 在仅仅使用 15% DRAM 的情况下，便覆盖了盖区域将近 50% 的 LLC Miss。而 hsqlbd 的行为和二者均不相同。在迁移其热函数 (Hot Methods) 访问数据的过程中，其旧生代 (Mature Space) 的使用比例和覆盖的 LLC Miss 比例几乎成线性关系，如在迁移了 43% 的数据到 DRAM 后，却仅仅让 DRAM 覆盖了 56% 的 LLC Miss。

最后，图 3.21 展示了 Specjbb、pmd、hsqldb 经过两级数据布局后的性能。纵轴为程序执行时间与当内存全部为 DRAM 时的比值。可以看到，和无任何管理相比，经过两级映射后，SPECjbb 和 pmd 在分别使用了占其工作集 (Working set) 16% 和 23% 的 DRAM 后，性能和全部使用 DRAM 相比只有 23% 和 12% 的差距。而对于 hsqldb 来说，在使用了占工作集 (Working set) 51% 的 DRAM

后，性能和全部使用 DRAM 相比只有 8% 的差距。

对于 Dacapo 中的其他应用来说，如图 3.13所示，在仅仅将新生代区域 (Nursery Space) 映射到 DRAM 后，和全部使用 DRAM 相比，平均只有 10% 的性能差距。由此可知，针对 Dacapo 和 SPECjbb 应用，利用托管式运行时，在两级粒度划分冷、热数据，并利用 GC 机制，在数据对象粒度 (Object) 来精准的布局数据可达到令人满意的性能。

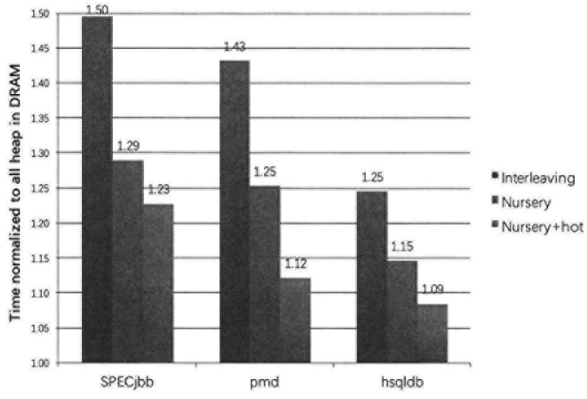


图 3.21 使用运行时进行两级粒度数据布局后的性能收益

Figure 3.21 The performance gain after applying own approach

3.7 本章总结

由于托管式应用需要基于运行时系统执行，并具有不同于 C/C++ 程序的数据对象模型, 如 Java Object Model，和管理方式，导致一些原有的基于 Trace 的数据访存行为分析方法[5][10]，以及基于操作系统、硬件的异质内存管理策略等[12][13] 不再适用。

对此，本章提出了直接利用运行时系统本身的机制，而非操作系统、硬件，来对异质内存进行数据管理的策略。该策略不但可以避免图 1.2 中展示的数据移动冲突问题，还解决了传统 C/C++ 的数据访存行为分析方法在托管式应用中不可用的问题。最终，本文提出了一种基于程序语义来进行冷、热数据划分，并利用 GC 机制进行数据移动的，两级粒度的异质内存管理策略。首先，通过不同配置下的访存行为定量分析，将运行时堆现有的区域粗粒度的划分为了冷、热、一般三类。然后，基于本文的发现-“托管式程序多数的 LLC Miss 由极少量函数造成”，提出了一种基于函数的细粒度冷、热数据划分方式；最后，

通过修改 GC 机制来低开销的迁移划分好的数据到对应的 DRAM 和 NVM 。

本研究提出的管理方案并不是孤立的，其可以和原有的操作系统、硬件方案进行结合，比如，利用操作系统、硬件来做第二级细粒度的划分，而不是利用程序分析的方式。后续的研究将尝试这些思想来解决更多的数据布局问题。

后续的研究将针对内存计算应用，如 Spark 应用，和通用的商业化运行时系统 OpenDJK 8，进行研究。本章的研究是基于 Dacapo、SPECjbb 等单机应用进行，虽然具有一定的局限性，但给本文后续的研究带来了很多新思路。

第 4 章　内存计算应用的内存使用特征分析和管理

为了解决传统磁盘带来的计算性能瓶颈，越来越多的大数据计算框架通过将数据置于内存来加速计算，并提出了内存计算的概念。内存计算框架的设计初衷为将参加计算的数据直接置于内存中，减少硬盘 I/O 操作，甚至将用于容错的数据也置于内存中，来加速分布式程序的错误恢复速度[8]。相比于传统基于磁盘的大数据框架，基于内存的计算平台可以将整体性能加速 20 倍以上[8]。然而，DRAM 技术已经无法满足内存计算框架对内存容量快速增长的需求[6]。因此研究人员逐渐将注意力投向一些新的内存材质，如非易失性内存(Non-Volatile Memory, NVM)。目前较为成熟的非易失性内存技术，如正在逐渐产品化的相变存储器 (Phase-Change Memory)，其读、写性能虽然比传统 SSD 高一个数量级，却仍旧和 DRAM 有一定的差距。因此，将 DRAM 和 NVM 构建成一体化的异质内存，并辅以恰当的数据布局管理是一个研究重点。

对于托管式应用而言，将大量数据缓存虽然会减少磁盘 I/O 开销，但仍可能由于增加 GC 开销而引起程序整体性能的下降。另一方面，由于内存计算应用的工作集 (Working set) 庞大，操作系统、硬件传统的以页 (Page) 为粒度的异质内存管理策略会带来显著的数据划分、迁移开销[5]。针对以上问题，本章分析了内存计算框架 Spark 的计算行为和内存使用特点，以及其和运行时系统 OpenJDK 之间的交互行为，并提出以下结论：和单机应用相比，内存计算应用 (Spark 应用) 具有简洁、清晰的计算行为和数据使用逻辑。因此，仅需提供必要的接口，让应用开发人员可以在自定义数据结构粒度 (RDD 粒度) 进行粗粒度的数据布局，便可带来良好的数据布局效果。

只有透彻了解应用的计算行为、数据使用行为的特点才能提出合理的数据布局策略，本章着重阐述针对 Spark 应用的分析和最终提出的异质内存管理框架设计，并将部分难点的分析、讨论和提出的方案置于了下一章 (第 5 章) 进行详细阐述。本章将部分策略实现在了被业界广泛采用的真实平台，内存计算框架 Spark 和运行时系统 OpenJDK 之上，并进行了充分的实验论证。

4.1 研究动机与概要

本节以 Spark 作为代表来研究内存计算框架的内存使用行为和内存性能瓶颈，只有了解内存不足时给 Spark 应用造成了哪些性能瓶颈，才能有针对性的提出解决方案。

Spark 是使用 Scala[77] 语言开发，基于运行时系统执行的分布式内存计算框架，并广泛的被工业界所采用。Spark 通过将计算过程中的中间数据缓存到内存中，来加速计算、错误恢复的过程。为了增加计算行为的并发性、容错性，Spark 提出了一个分布式内存抽象，RDD (Resilient Distributed Datasets)，作为计算和容错的基本数据单元。同时，在运行时系统之上 (Java Runtime)，Spark 也提出了一套基于自身的内存管理模型。作为一个分布式计算框架，Spark 由 Master 和 Slave 两部分构成，而在 Slave 节点中，实际的计算由 Spark Executor 进行。一个 Slave 节点可以有一个或者多个 Executor 进行计算，每一个 Executor 均为一个独立的 Java 虚拟机，(Java Virtual Machine, JVM)，也即拥有自己独立的运行时堆 (Java heap)。对应的，在 Spark 层次，一个 Spark Executor 内部的内存空间被划分为了 Spark Execution Memory 和 Spark Storage Memory 两大部分，来分别存储计算过程中生成的临时数据，以及被开发人员显式缓存到内存中的计算数据。

本章分为三个部分，第一部分分析了 Spark 应用的计算行为以及内存不足时给其带来的性能瓶颈；第二部分介绍了本章针对 Spark 框架提出的自定义数据结构粒度 (RDD) 的布局策略；第三部分通过实验对以上布局策略进行了初步的验证。

内存不足时带来的性能瓶颈

Spark 对内存容量的需求随着应用计算规模的增大而快速增加。但是由于 DRAM 工艺瓶颈的限制，其容量增加和功耗控制难以满足内存计算应用持续增长的需求，常常出现由于内存不足而导致 Spark 程序性能严重降低的问题。比如，在 700 MB 的输入集，32GB 的配置 Executor 的配置下，Spark PageRank 的 GC 开销便达到了 35.3% 。与此同时还有大量的 RDD 因为无法置于内存中而被 Spark 替换到了磁盘中，这不但显著增加了磁盘 I/O 操作，还会导致大量的无效计算产生，给 CPU 带来额外的压力。本章在第 4.2 节中从磁盘 I/O、无效计算、GC 三个方面系统分析了内存不足时，会在哪些方面导致性能瓶颈，并

探讨了如何解决这些性能瓶颈。

内存计算应用的粗粒度数据划分

存储密度十倍于 DRAM 的 NVM[10] 给解决大数据计算内存不足的问题带来了新的机遇。但是，目前主流的 NVM 性能无法和 DRAM 相比，因此如果直接使用 NVM 来扩展主存，虽然会带来一定的性能提升，但收益并不明显。

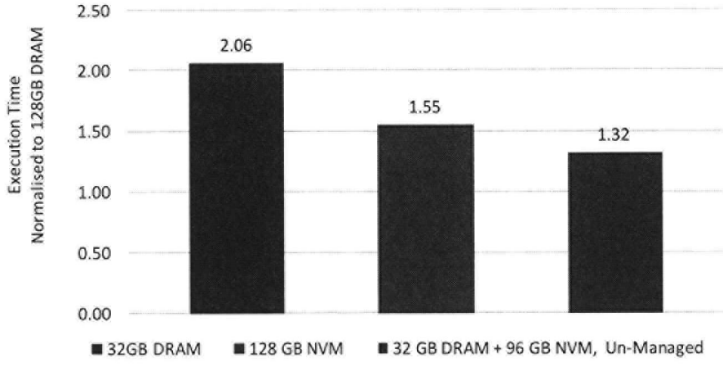


图 4.1　运行于异质内存的应用需要恰当的数据布局

Figure 4.1　Utilize heterogeneous memory randomly can't achieve good performance

如图 4.1 展示了使用非易失性内存扩展主存后，在一定配置和输入下 Spark Pagerank 的性能。当内存只有 32GB DRAM 时，和 128GB 的 DRAM 相比，执行时间将整体增加 106%。当程序执行于 128 GB NVM 时，其性能比使用等量 DRAM 降低了近 55%。当使用 96 GB NVM 来扩展原有的 32GB DRAM 后，在没有应用恰当的数据布局管理时，程序性能虽然增加，但仍旧和全部使用 128GB DRAM 时有 32% 的差距。这是由于，虽然大内存缓解了磁盘 I/O 开销、Spark 无效计算、GC 开销，但却由于 NVM 和 DRAM 的性能有一定差距，热数据运行于 NVM 时会造成一定的性能瓶颈。

而本章通过对内存计算框架 Spark 进行分析，发现其迭代计算应用具有简洁、清晰的计算行为, 而且 Spark 应用的主要数据，RDD，亦具有清晰可辨的生命周期和冷、热属性，如图 4.2 所示为 Spark PageRank 的计算过程。Spark 以“Stage”作为计算的基本单元，一个 Stage 的最终计算结果会被输出到磁盘中，来进行 Shuffle 计算。因此，如果一个 RDD 没有被开发人员进行显式的持久化，RDD 对应的“内存数据”将会在 Stage 结束后死亡，并被 GC 释放，并在下次使用时被重新构建。因此，对于一些迭代计算的应用，为了加速计算会

将一些在每个 Stage 都频繁使用的 RDD 持久化到内存中，以减少重复计算，如图中的 RDD *links*；亦或是将一些迭代中间的计算结果进行持久化，来加速错误恢复，如图中的 RDD *contribs*。

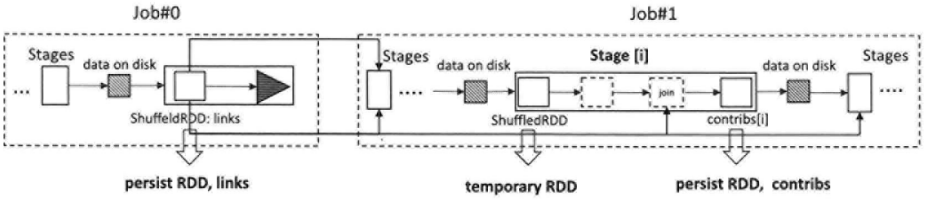


图 4.2　Spark 迭代计算应用的计算流程

Figure 4.2　The computation behavior of Spark applications

针对该现象，本章提出可以将 Spark 应用数据粗粒度的划分为以下几类来进行数据布局：

I **短生命周期 RDD 与其他临时数据。**快速死亡的数据，会被 GC 快速回收，可以高效复用较少的 DRAM 空间来进行处理。然而，Nguyen 等人在 [24] 提出大数据应用的数据生命周期会跟随计算的迭代行为呈现出规则性，不再遵循“多数数据快速死亡”的假设。本章将对内存计算的该问题进行细致分析。

II **长生命周期 RDD 数据 (persist RDD)。**被开应用开发人员显式持久化的 RDD 数据，可以由开发人员根据其计算行为手动指定在异质内存中的位置。

i **被频繁访问的 RDD (Hot persist RDD)。**参与大量 Stage 的计算。

ii **访问不频繁的 RDD (Warm persist RDD)。**参与少量 Stage 的计算。

iii **减少访问、容错等 RDD (Cold persist RDD)。**参与一个或极少数 Stage 的计算，迭代过程中的中间结果，被持久化到内存来进行快速恢复。

本章在接下来的小节中对以上粗粒度的数据划分、布局策略进行了分析，并通过实验进行了验证。

4.2　内存不足时造成的性能瓶颈分析

由于 DRAM 的工艺限制，其在功耗、容量密度等方面已经很难满足应用快速的增长的需求。本节将在接下来的内容中分析，当主存不足时 Spark 将会

在磁盘 I/O、计算、GC 等方面面临怎样的性能瓶颈，以及这些性能瓶颈对应的解决策略。

4.2.1 磁盘 I/O 开销分析及应对方法

随着芯片计算能力的快速发展，其与传统磁盘之间的性能差距被逐渐拉大，为了弥补磁盘导致的性能问题，科研人员从硬件、操作系统、文件系统、上层软件等多个层次进行了针对磁盘的优化工作。

硬件方面提出了增加多级中间缓存、开发新型磁盘介质等方式来减少磁盘 I/O 开销，如作为磁盘缓存的 Intel Optane[78]，亦或是 SSD 等新型磁盘介质。但这些新产品价格较昂贵，而磁盘容量较大，部署这些新型磁盘会将服务器整机的成本显著拉高。除此之外，这些产品仍旧和内存 (DRAM) 有一个数量级的延迟、带宽差距。此外，诸如操作系统、文件系统等软件层次，则是希望通过直接利用服务器中的空闲内存来作为磁盘缓存，来尽量避免对磁盘的访问。该种策略依赖于大量空闲的主存，当应用程序将主存耗尽时，这些软件策略往往就失去了效果。

分布式计算框架 Spark 的计算行为仍以 Map-Reduce 为基础，而 Shuffle 计算为 Reduce 计算过程中的一步，其将 Map 计算的结果写入磁盘以供各个 Reduce 计算进行使用。Spark 为了保持良好的容错性和效率，仍旧使用了该种写磁盘的 Shuffle 机制。另一方面，Spark 的内存模块由 Spark Execution Memory、Spark Storage Memory 两部分组成，而当 Spark Storage Memory 不足时，持久化的 RDD 无法被完整置于其中时，便会将一些先前持久化的 RDD 替换到磁盘，而该过程会进一步造成磁盘读写。本节接下来的部分将针对 Spark 应用分析内存不足时的磁盘 I/O 开销。

内存不足时的磁盘 I/O 开销。

表 4.1 以 Spark Pagerank 为例，展示了服务器内存不足时带来的磁盘 I/O 等性能影响。其中，第一列展示了只有 32GB DRAM 内存时的性能。为了避免发生 Swap 操作，我们将 Spark Executor 的内存容量限制在了 30GB。此时，操作系统最多有 2GB 空闲内存可以用来磁盘缓存。和有 128GB DRAM 内存相比，此时的程序执行时间变慢了 106% 。由于运行时堆 (Java Heap) 的容量不足，原本置于内存中的 RDD ，有将近 26GB 的数据被替换到了磁盘中。除此之外，Map-Reduce 框架中的 Shuffle 操作需要将计算结果写入磁盘，以便在多

表 4.1 内存不足时带来的性能影响

Table 4.1 The performance degradation caused by not enough memory

Configuration & Results	SMALL MEMORY +DISK	INFINITY DRAM	INFINITY DRAM
Physical Memory(GB)	32	128	128
Executor Memory(GB)	30	30	120
Disk Cache(GB)	<2	Enough	Enough
Shuffle Write To Disk(GB)	20	20	20
RDD To Disk(GB)	25.9	25.9	0
Elapsed Time (S)	2373	1591	1151
Normalized Elapsed Time	2.06	1.39	1.00

进程之间传递数据。在该例子中，Shuffle 写入磁盘的数据也高达 20 GB。而此时的操作系统缓存不足 2GB，因此引发了显著的磁盘读写操作。

被替换到磁盘的 RDD 和 Shuffle 操作两项合计将 46 GB 的数据写入了磁盘，而此时的系统缓存不足 2 GB 可用，自然会造成严重的磁盘 I/O 开销。图 4.3 展示了只有 32GB DRAM 时的磁盘读请求和 CPU I/O 等待时间的变化曲线。该图的左侧纵轴为每 10s 进行一次采样，统计内读取磁盘的频度，右侧纵轴展示了因为 CPU 磁盘 I/O wait 时间占总 CPU 时间的百分比。从该图中可以看到，此时磁盘距有明显的访问请求，而 CPU I/O 等待时间的变化趋势和磁盘读请求的变化基本一致。频繁的磁盘访存拖慢了整个程序的性能。

系统缓存显著减少磁盘 I/O 操作。 当将服务器的内存增加到 128GB，但是维持 Spark Executor 的可用内存仍旧为 30GB 时，配置如表 4.1 第二列所示，虽然被替换出内存的 RDD 和 Shuffle 操作仍旧将 46GB 的数据写入了磁盘，但是程序的性能发生了显著的提高。这是因为，此时操作系统和文件系统有了充足的空闲内存可以作为磁盘缓存来使用。这些写到磁盘的数据会被暂存于内存中，当再次读取这些数据时，将会直接命中磁盘缓存，而不是需要从磁盘中读取，因此并未造成显著的磁盘读写操作，如图 4.4 所示。此时的系统缓存 (Cached Memory) 的使用量高达 60GB，显著的加速了磁盘的读写操作。

在服务器有充足 DRAM (128GB) 时，我们增大了运行时堆的大小到 120 GB，此时不但提供了充足的磁盘缓存，从内存写入磁盘的 RDD 亦减少了 26GB，进一步减少了磁盘 I/O 开销。至此，可以看到只需增大内存大小和运行时堆大

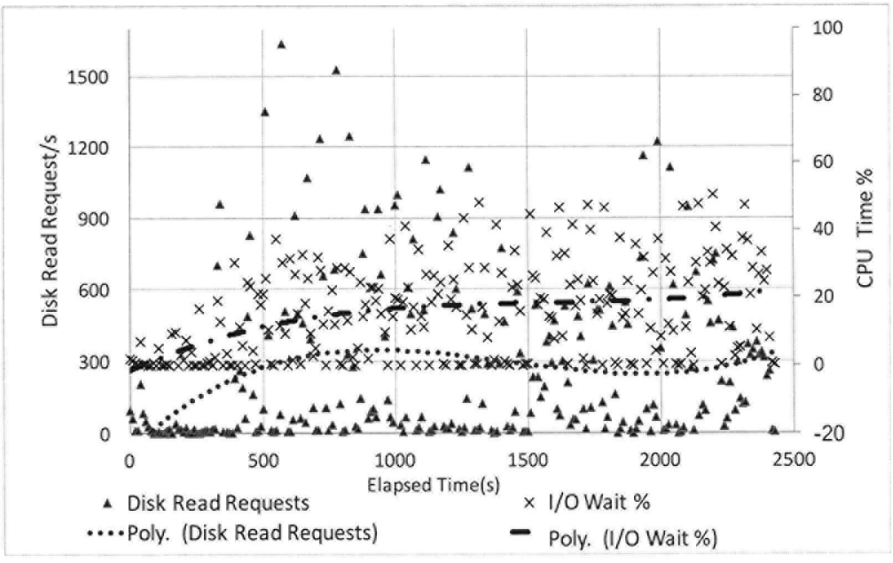


图 4.3　32 GB 内存时，磁盘读请求、CPU I/O 等待时间的变化趋势

Figure 4.3　Trend of Disk Read Request number and CPU I/O wait under 32GB DRAM

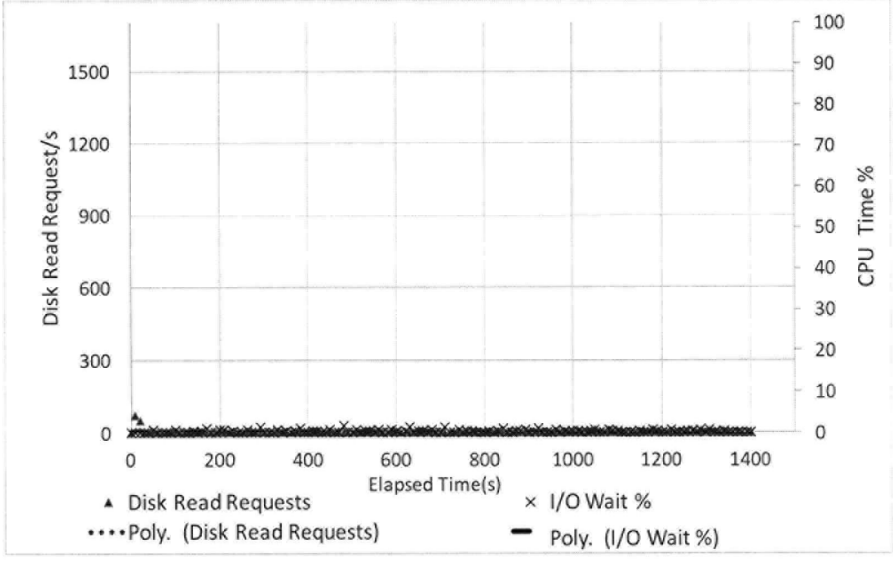


图 4.4　128GB 内存时，磁盘读请求、CPU I/O 等待时间的变化趋势

Figure 4.4　Trend of Disk Read Request number and CPU I/O wait under 128GB DRAM

小，在现有 Spark 机制、操作系统磁盘缓存机制的处理下，便可显著的减少磁盘 I/O 带来的性能开销。研究 [79] 认为，目前各个层次，如操作系统、文件系统、分布式系统等，都针对磁盘进行了不同的优化操作，因此，磁盘对最新大数据应用的性能影响一般在 19% 左右。我们将在接下来在的小节中利用 NVM 扩展主存来减少磁盘 I/O 开销，并通过合理的数据布局来避免 NVM 性能不足

导致的性能瓶颈。

4.2.2 内存不足导致的无效计算开销

当内存不足时 Spark 框架将会导致两个方面的额外计算开销。首先，正如第 4.2.1 节中分析，当运行时堆内存不足时，一部分 RDD 将会被替换到磁盘，而该过程将会对替换到磁盘的 RDD 进行序列化压缩操作，这会显著增加所在 Stage 的计算开销。而如果 RDD 本身便以序列化的形式存在于运行时堆，那么该处额外的计算开销将会减少。

另一方面，经过分析发现，由于 Spark 有自己的内存管理模型，其会根据自身的内存使用统计来做出控制 RDD 的决策。如某个 RDD 的大小为 5GB 时，此时剩余的 Spark Storage Memory 空间仅为 3GB, 而 Spark 无法预先判断此时是否可以容纳整个 RDD，其会首先尝试 RDD 的计算和持久化工作，当 Spark Storage Memory 无剩余空间提供时，其会依次尝试：1）首先向 Spark Execution Memory 借用空间；2）若借用失败，其尝试将一些 RDD 替换出 Spark Storage Memory；3）如仍无足够空间，其会取消当前的计算过程，并尝试将 RDD 写入磁盘。

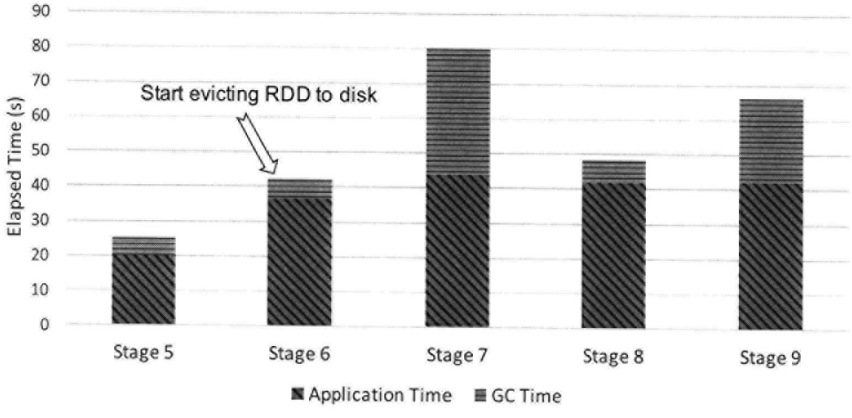


图 4.5 Spark PageRank 每次迭代的执行时间分解

Figure 4.5 The execution time decomposition of each Stage for Spark PageRank

当内存不足，而 RDD 较大时，将会快速发生上述问题，造成额外的计算开销。如图 4.5 展示了 Spark PageRank 应用在 32 GB 运行时堆 (Java Heap)，675MB 输入集下，计算量相同的五次迭代 (Stage) 的计算时间分解。一个程序的执行时间可以分为单纯的“应用计算时间 (Application Time)”和“GC 时间

(GC Time)”两部分，仅从应用计算时间 (Application Time) 来看，从 Stage 6 开始，其应用计算时间 (Application Time) 便开始显著增加，并稳定在原计算时间的 200% 左右。这是由于从 Stage 6 开始，Spark Storage Memory 开始发生数据被替换到磁盘的现象。

而如果运行时堆 (Java heap) 的大小相对于输入集显著增加，那么将会大大延缓 RDD 被替换到磁盘情况的发生时机，减少无效计算的发生。对此，我们测试了相同输入下，运行时堆 (Java heap) 大小对应用计算时间的影响，图 4.6 展示运行时堆 (Java heap) 从 16GB 扩展到 120GB 的过程中，程序的应用执行时间 (Application Time) 和 GC 时间的变化趋势。需要强调的是，此时系统的内存充足，有足够的磁盘缓存来减少磁盘 I/O 开销，整个测试过程中并没有检测到显著的磁盘读请求操作。从该图中可以看到，随着运行时堆 (Java heap) 大小的增加，程序的运算时间 (Application Time) 快速减少，在 120GB 大小时，并没有 RDD 被替换到磁盘，也即几乎没有无效的额外计算。

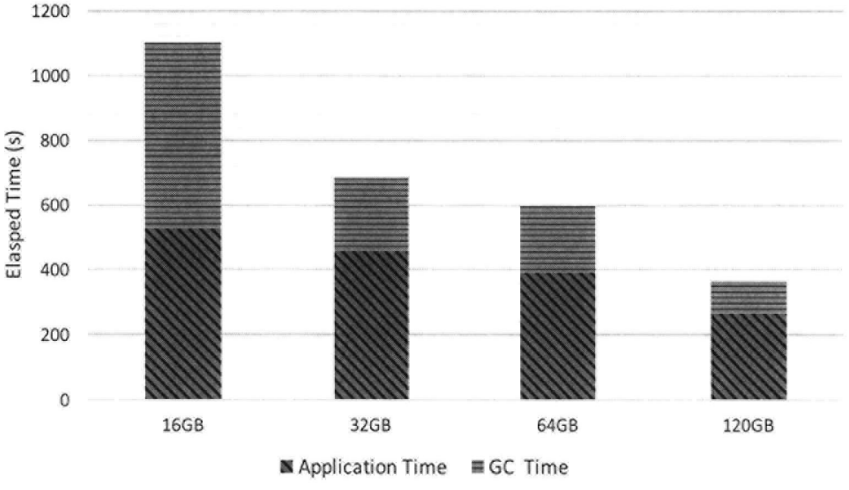


图 4.6　Spark PageRank 执行时间的分解

Figure 4.6　The execution time decomposition of Spark PageRank

至此，可以看到当内存不足时，将会给程序带来显著的无效计算开销，而通过扩大内存大小，便可以高效的消除这些额外计算。根据 Ousterhout 在 [79] 中的分析，CPU 逐渐成为了制约大数据应用的性能瓶颈，而通过扩展内存便可减少将近一倍的无效计算，因此高存储密度的 NVM 给加速大数据应用计算带来了全新的机遇。

4.2.3 GC 开销分析及处理方式

为了易编程性和跨平台的通用性，Spark 选择了基于托管式语言进行开发。而由于其为了加速计算将大量的数据缓存到内存中，导致 GC 的开销成为了限制 Spark 应用性能的一个显著因素。针对该问题，科研人员都给与了高度重视，如 Tungsten 计划[46] 提出的将部分数据置于 Off-Heap 来减轻 GC 压力，以及华中科技大学[21] 提出的，重新构建 Spark 的基本计算单元，RDD，将其压缩为单一的 Byte Array 以减少堆中数据对象 (Object) 个数的策略，其目的均为降低 GC 的开销。本节将对 Spark 应用中导致 GC 开销的原因、解决方案进行讨论。

Spark 系统带来的垃圾回收 (GC) 开销有两个方面：1）DRAM 容量无法满足内存计算规模, 限制了运行时堆 (Java Heap) 大小，导致 GC 被频繁的触发。2）大量长期存活的数据被缓存到运行时堆 (Java Heap) , 导致 GC 单次扫描的时间过长。综上所述，如果想减少 GC 需要从两个方面来进行：减少 GC 的频度；减少单次 GC 的开销。

内存容量对 GC 的影响。

当内存不足时，为了避免 Swap 现象的发生，运行时堆 (Java heap) 的大小将受到限制，此时，GC 会被频繁的触发。图 4.7 展示了，在相同输入下，运行时堆 (Java heap) 从 16GB 增长到 120GB 时各项 GC 行为的变化。从该图中可以看到，当内存、运行时堆 (Java heap) 同时增大时，GC 开销并非成比例的进行减少。GC 行为受到了多方面的因素影响，运行时堆 (Java heap) 仅为影响 GC 频度的 (GC count) 的关键因素。

从图 4.7 中可以看到，随着运行时堆 (Java heap) 以 2 倍的形式增大，GC 频度 (GC count) 成比例的进行降低。然而，单次 GC 时间的不规律变化，GC 的总开销并没有和预期一样成比例减少，而 GC 时间占程序执行时间的比例虽然整体成下降趋势，但和运行时堆 (Java heap) 容量之间亦没有相应的比例关系。根据本节分析，这是由于当运行时堆 (Java heap) 较小时，有大量的 RDD 会被替换到磁盘，而这会减少运行时堆 (Java heap) 中数据对象的数量，也即变相降低了单次 GC 的开销。从图中可以看到，随着运行时堆 (Java heap) 的增加，单次 GC 的平均开销亦逐渐增大。本节的后半部分将分析，如何通过更改 RDD 状态来减少存活数据对象 (Object) 数目来减少单次 GC 开销。

图 4.7 中展现的另一个趋势为，虽然随着运行时堆 (Java heap) 增大，GC 的

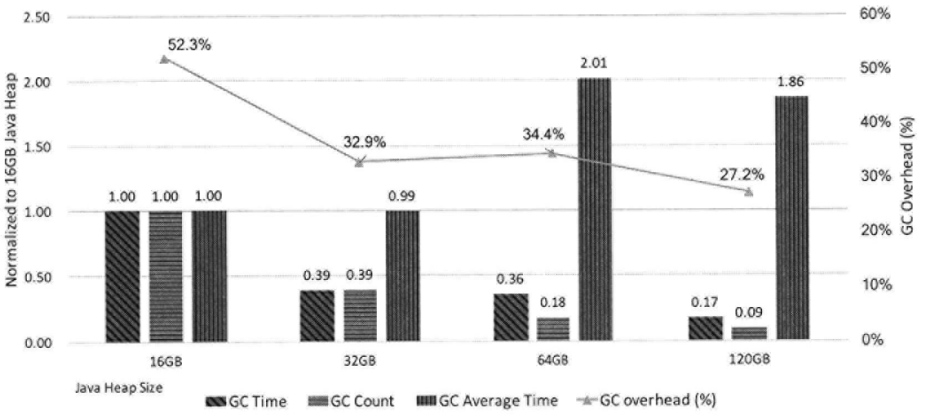


图 4.7　内存大小对 GC 行为的影响

Figure 4.7　The affect of Java heap size on GC behavior

时间显著降低，但是其在整个程序中所占的比例并没有显著降低。这是因为，随着运行时堆 (Java heap) 增大，其会减少 Spark 框架中的无效计算，第 4.2.2 节已对此进行了详细分析。综上所述，内存容量会有效减少 GC 频度，然而单次 GC 的开销同时取决于运行时堆大小以及其内部的数据形态。

RDD 持久化状态对 GC、无效计算性能的影响。

本节探讨运行时堆中的数据形态 (RDD storage level) 对 GC 开销、无效计算开销的影响。减少单次 GC 开销需要减少运行时堆 (Java Heap) 中的存活数据对象的数量，或者重新设计针对大数据的 GC 算法[24]。就 Spark 层次而言，由于其将大量的数据显式的缓存到主存中，显著的增加了行时堆 (Java Heap) 中数据对象 (Object) 的数量。此外，一个在 Storage Memory 中长期存活的持久化 RDD，可能由上百万独立的数据对象 (Object) 构成。每一次 GC 都要扫描 RDD 中每一个存活的数据对象，会导致巨大的开销。而被缓存到 Spark Storage Memory 中的部分 RDD 仅仅是为了快速容错，在计算过程中，并不会频繁的对其进行访问。因此，针对该种容错型 RDD，本文建议将其序列化 (Serialization) 后存储在内存中。经过序列化，一个原本由众多独立数据对象构成的 RDD，会转化为一个独立的 Byte Array。所有数据经过压缩后，均被储存在这个独立的数据对象中。同时，其所占据的内存空间也会被极大的压缩，因此也会降低 GC 被触发的频率。

另一方面，经过序列化的数据对象大小会被显著压缩，如 Spark PageRank 中的 RDD *contribs* 序列化后，占用的大小被压缩了 6.8 倍。此时会显著延缓内

Algorithm 1 The Spark code for PageRank[8]

```
1
2     val lines = ctx.textFile(args(0), slices)
3     val links = lines.map { s =>
4       val parts = s.split("\\s+")
5       (parts(0), parts(1))
6     }.distinct().groupByKey()
7      .persist(StorageLevel.MEMORY_ONLY)
8
9     var ranks = links.mapValues(v => 1.0)
10    for (i <- 1 to iters) {
11      val contribs = links.join(ranks).values.flatMap {
12        case (urls, rank) =>
13          val size = urls.size
14          urls.map(url => (url, rank / size))
15              .persist(StorageLevel.MEMORY_AND_DISK_SER)
16      }
17
18      ranks = contribs.reduceByKey(_ + _)
19              .mapValues(0.15 + 0.85 * _)
20    }
```

存空间耗尽发生的时刻，也即会减少第 4.2.2 节中所述的无效计算的发生。

然而，在 Spark 架构中，如果想在一个序列化的 RDD 上进行计算，需要首先将其反序列化。这将耗费大量的 CPU 计算量。而在现代大数据系统中，CPU 的计算能力已经逐渐成为了限制其性能主要的瓶颈[79]，因此需要仅将访问不频繁的 RDD 进行序列化以平衡 CPU 计算开销和 GC 开销。本文认为，应该对 Spark Storage Memory 中的持久化 RDD 进行分类，将其中用于容错的 RDD 进行序列化存储，而将其中频繁计算的 RDD 仍旧保持非序列化的状态。如代码 1 中列出的 Spark Pagerank 算法。通过该简洁的算法可以清晰的看到其中 RDD 的冷、热属性，RDD *links* 将会参与每一次迭代中的计算，但 RDD *contribs* 仅仅参加一次迭代中的计算，然而，如果内存空间足够，开发人员倾向于将其持久化用于容错。因此，为了将 GC 开销控制到最小，这里仅将频繁参与计算的 RDD *links* 设置为 MEMORY_ONLY 级别，但是序列化主要用于容错的 RDD *contribs* 为 MEMORY_AND_DISK_SER。

经过以上变化后，和将 RDD 全部以非序列化形式存储对比，在 120GB 堆大小的配置下，Spark Pagerank 的性能大幅提高了 66%，如表 4.2 所示，其中，GC 开销降低了 87%，消除大量无效计算后，应用自身的执行被加速了 60%。

在大量用于容错的 RDD 以非序列化形式存在时，一次 MajorGC (Full GC) 便可以消耗 128 秒的时间。因此，如无特殊说明，本文中实验部分所使用的测试用例都经过了该优化：只将频繁参与计算的 RDD 以非序列化形式存储于行时堆 (Java Heap) 中，而计算不频繁、或者用于容错的 RDD，将以序列化的形式存储于行时堆 (Java Heap) 。

表 4.2 优化 Spark Pagerank 的 GC 开销

Table 4.2 Optimize the GC overhead for Spark Pagerank

配置 & 结果	原程序	优化后
Physical Memory(GB)	128	128
Executor Memory(GB)	120	120
迭代次数	30	30
RDD links	MEMORY_ONLY	MEMORY_ONLY
RDD contribs	MEMORY_AND_DISK	MEMORY_AND_DISK_SER
执行时间 (s)	1547	933
执行时间加速比	1	1.66
应用执行时间加速比	1	1.60
GC 加速比	1	1.87

4.2.4 本节小结

通过以上分析，可以看到由于内存不足在各个方面带来显著的开销。首先，由于内存不足，数据被内存计算框架 (Spark) 显式写到磁盘，而导致的磁盘 I/O 开销。而此时磁盘缓存的不足亦加剧了磁盘读写开销。其次，由于可用内存空间不够而导致部分 Spark 计算无法完成，从而造成的无效计算开销。以及运行时堆受限于内存容量，导致 GC 被频繁触发所带来的开销。

通过将存储密度更高、性能远强于磁盘的 NVM 整合到服务器，可以直接解决其中的部分问题。如充足的内存不但可以避免 Spark 将内存中的数据替换到磁盘，亦为磁盘提供了足够的缓存，有效的减少了磁盘 I/O 操作。而另一方面，扩大的内存和运行时堆 (Java heap) 亦可减少 GC 的频率与无效计算的发生。

然而，更大的主存和运行时堆 (Java heap) 意味着更多数据对象的存在，在减少磁盘 I/O 开销的同时确会显著增加单次 GC 的开销，如图 4.7 所示，当运行时堆容量从 32 GB 增长到 64GB 时，程序的 GC 开销甚至几乎没有减少。另

一方面，通过以上分析可知，NVM 的性能无法满足一些热数据对内存性能的要求。数据在异质内存中的随机布局，将会造成显著的内存性能瓶颈。

本章通过对典型内存计算迭代应用 (Spark 应用) 的分析可知，内存计算应用具有清晰、简洁的计算流程，开发人员可以快速从程序源码中识别出程序分析难以发现的程序语义。因此，本文针对该问题提出了合理的异质内存编程接口，来允许编程人员将一些程序语义传递到内存计算框架和运行时系统，以改善不同数据在运行时堆中的形态和不同冷、热数据在异质内存中的布局。

4.3 用户控制的粗粒度的数据布局策略

相比于 DRAM，NVM 可以做到十倍于其的存储密度[10]，因此，其给解决内存不足带来的各种问题带来了机遇。但如第 4.1 节所述，当直接使用 NVM 扩展主存时，如果不加任何的管理，相对于扩大的容量来看，程序的性能提升并不显著。如图 4.1 所示，在使用 NVM 将 32GB 增加到 128 GB，并设置 120GB Spark Executor Memory 的情况下，性能增加了 56%，但是仍旧和使用 128 GB DRAM 有将近 32% 的差距，然而此时的内存容量已经达到了原来的 4 倍。

如第 4.2.2节所述，扩大内存容量可以减少 GC 开销，并消除大部分磁盘 I/O 开销和无效计算，但是 NVM 却无法满足一些数据对内存性能的需求。图 4.8 展示了某段时间内，Spark Pagerank 在 DRAM 上的读带宽趋势，可以看到，内存的读带宽已经达到了 15.4 GB/s，超过了 NVM 12.8GB/s 的理论峰值。同时，NVM 的延迟也为 DRAM 2-3 倍，在其上的频繁访存同样会造成性能下降。因此，只有通过合理布局数据在异质内存上的分布，避免非易失性内存的性能成为整机的瓶颈。

为了进一步提升性能，需要对运行时堆 (Java heap) 中的数据进行划分，并将其合理布局到 DRAM 和 NVM 中。而本节通过分析 Spark 迭代计算应用的计算行为，发现其数据使用行为简洁、规则，因此提出可以直接由开发人员进行粗粒度 (RDD 粒度) 的数据布局。并将数据分为以下几类：

I 短生命周期 RDD 与其他临时数据 (Non-persist RDD and Temporary data)

II 被持久化的，长生命周期 RDD 数据 (Long life-time RDD / persist RDD)

i 被频繁访问的持久化 RDD (Hot persist RDD)

ii 访问不频繁的持久化 RDD (Warm persist RDD)

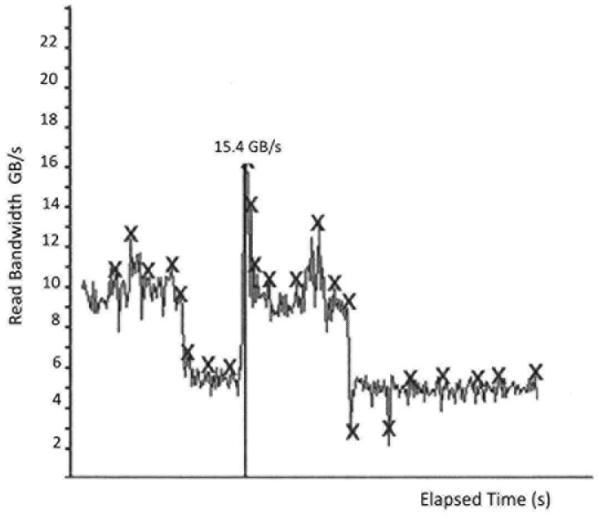


图 4.8　Spark Pagerank 运行时于 DRAM 上时的读带宽

Figure 4.8　The read bandwidth on DRAM of Spark Pagerank

iii 极少访问的、用于容错的持久化 RDD (Cold persist RDD)

通过分析，我们认为一个自定义数据结构, 如 Spark RDD，中的数据具有类似的访存行为和一致的生命周期。因此，本文将以 RDD 粒度来划分数据，并将其布局到异质内存中，布局策略如图 4.9 所示。

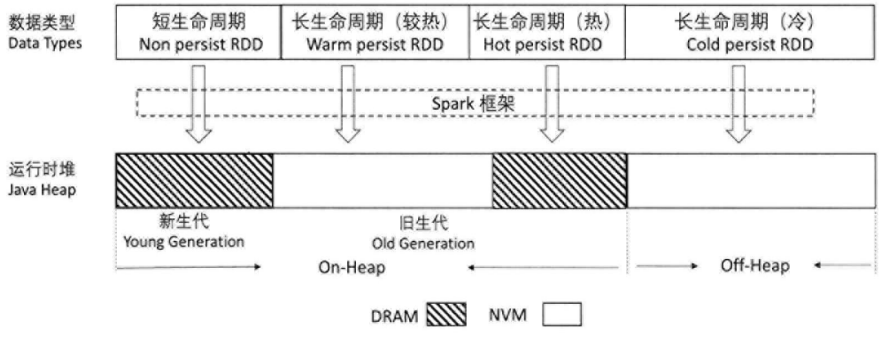


图 4.9　Spark 应用数据的粗粒度划分和布局

Figure 4.9　The coarse grained data layout of Spark data types

本章将在接下来的小节中，分别介绍各种数据在运行时堆中的布局策略与依据，并在第 4.4 节中对该布局策略进行了真机实验分析。需要强调的是，本章以分析应用计算行为、阐述一体化异质内存管理框架的设计为主，通过本章对 Spark 应用的深入分析，了解其应用的计算特性和数据使用特性后，才提出了开发人员控制的 RDD 粒度数据布局策略。然而，由于整个一体化异质内存

管理框架的细节较多，一些问题的解决策略将在第5章中进行进一步阐述。

4.3.1 短生命周期数据的布局策略

第3章中，针对单机应用测试集Dacapo和SPECjbb，提出了利用运行时堆(Java heap)的组织形式和以函数为粒度的方式对数据进行冷、热划分。其中提出，由于单机应用中的多数数据快速死亡，这些数据会在新生代中被快速回收，导致新生代区域有非常高效的使用效率，因此可以将其直接映射到DRAM。

而Nguyen、Xu等人却在[24]中提出，大数据应用的数据生命周期以不再遵循传统的，“多数数据将快速死亡”的生命周期假设，并因此提出了“世代(epoch)”假设，即数据在仅会在迭代结束时大量死亡，在迭代结束之前的GC行为只能释放极少量的无用数据，非常低效，如图2.1所示。大数据中的一次迭代往往会触发多次GC，如果其中大部分的数据只会在迭代结束时死去，那么这些一次迭代内部的临时数据将会在计算过程中被GC迁移到旧生代区域，新生代区域的内存使用效率将十分低效。本节讲针对内存计算迭代应用(Spark应用)的该特性进行深入分析，分析结果决定了临时数据的处理策略。

对于Spark应用来说，其计算流程会被以Stage为单元进行切分，正如4.1节中所述，一个Stage内部的临时数据均会在其内部死亡，只有被用户显式持久化的数据才会长期存活。本节对GC行为和Stage之间的相关性进行了分析，以观察这些临时数据的生命周期规律，如图4.10所示。其中，每个方框表示了一个Stage的时间跨度，而曲线表示每次GC前运行时堆(Java heap)中的数据量，柱子表示了每次GC释放的数据量。

首先，从该图中可以观察到，一个Stage内部确实触发了数据众多的GC行为。其次，并无法观察到“世代(epoch)”现象，即并不存在“Stage结束时的GC释放的数据，远远大于其他GC”的现象。每一次MinorGC均释放了大量的数据，而其中三次数据释放最多的GC均为Full GC (MajorGC)，而经过分析，Full GC释放的大量数据多为被替换到磁盘的RDD死亡数据，亦或是一个独立任务(Job)完成后释放的无用数据，亦和Stage的结束无关。

通过图4.10可以看到，其每一次GC均会释放数GB的数据，在一个Stage内部可以释放数十GB的数据，并不会将数据留待Stage最后释放。该GC特征是由Spark的pipeline、以及“copy on write”的计算行为导致的。一个Stage的计算行为如图4.11所示，从数据角度，可以将Stage内部的计算视为

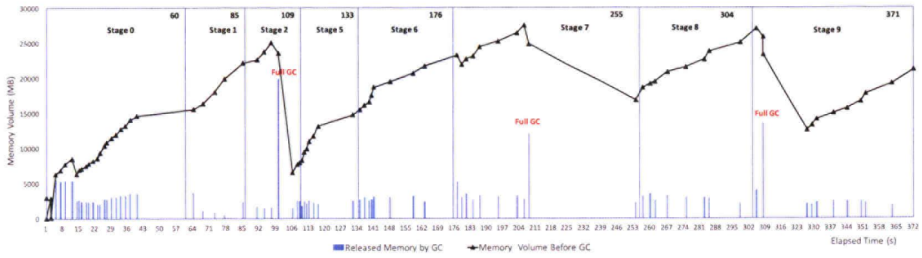


图 4.10　GC 行为和 Stage 计算单元之间的相关性

Figure 4.10　The correlation between GC and Stage

起始“实例化 RDD”到末尾“实例化 RDD”的并发计算，而中间的“非实例化 RDD”均不会去保持其对应的实际数据，仅是记录了计算行为和数据之间的依赖性。对应图 4.11 中的 RDD 编号，Stage 内部的 RDD 计算过程可以由公式：$f_9(f_8(f_7(f_6(Input\_from\_disk))))$ 来表示。其中 MapPartitionsRDD[9] 的每一个元素都是按照以上公式，从 ShuffledRDD[6] 计算得来的。

根据 Spark RDD 的“copy on write”实现特征，计算过程中如果对前面 RDD 的元素进行了修改操作，那么将会创建 (new) 一个新的数据对象，并对其赋以计算后的新值，而并非是修改原有 RDD 元素的值。另一方面，一旦一个新值被计算出后，其所依赖的旧值将会失去意义，可以被 GC 回收。在多线程并发的按照以上“copy on write”的形式并行计算时，自然会在每一步计算均产生大量的死亡数据。也因此导致，一个 Stage 内部中触发的每一次 GC 均会释放大量的数据，而不是等到 Stage 结束。

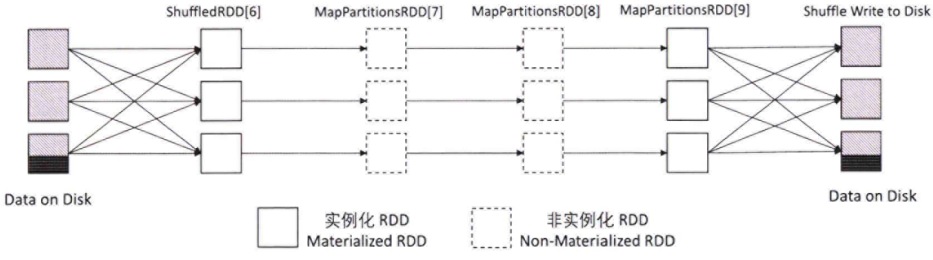


图 4.11　GC 行为和 Stage 计算单元之间的相关性

Figure 4.11　The correlation between GC and Stage

综上所述，对于 Spark 应用来说，其数据对象的生命周期和传统的大数据应用，如 Hadoop 等，有一定的区别，同时仍旧具有单机应用数据生命周期的特征。根据 Spark 应用的该特点，仍旧可以将其“临时数据及短生命周期 RDD”

交予运行时系统的新生代负责，而不必对其进行单独的数据划分。对于存活过了数次 MinorGC 被迁移到旧生代区域的该类短生命周期数据，由于其仍旧会快速死亡，而不会被频繁访问，本章将其置于了旧生代的 NVM 部分。我们在第 4.4 章对该策略进行了详细的实验分析，并取得了良好的效果。

表 4.3 控制运行时堆中的区域在异质内存上的映射

Table 4.3 Control the mapping of Young and Old generation on heterogeneous memory

Command line	Description
-XyoungGenOnNode	Position of Young Generation : 0 for NVM; 1 for DRAM
-XoldGenOnNode	Position of Old Generation: 0 for NVM; 1 for DRAM
-XDRAMPercentageOfOld	The percentage of Old Generation is mapped to DRAM

针对短生命周期数据的布局策略，本节基于 OpenJDK 开发了表 4.3 中所示的命令来控制各个生代在异质内存上的映射，以及旧生代中的 DRAM、NVM 比例控制。本文基于 3.2 节中提出的异质内存仿真平台，以及 NUMA 库函数实现了让操作系统暴露 DRAM、NVM 物理空间给运行时系统的方法。大数据应用开发人员仅需通过单一命令行，便可以将运行时堆的任何生代映射到 DRAM 或 NVM。

4.3.2 长生命周期冷数据的布局策略

一些数据被置于内存中的目的并不是因为其会被频繁访问，而是为了程序崩溃时进行快速的错误恢复。如一些迭代计算的中间结果等，亦或是一些使用不频繁，但重新计算开销巨大的 RDD。而根据第 4.2.1节中对磁盘 I/O 的讨论可知，由于其和 CPU、内存之间巨大的性能差距，将这些冷数据置于磁盘时，会带来显著的磁盘 I/O 开销。然而，这些冷数据确实无法充分利用 DRAM 的超高性能，NVM 的性能便可满足其性能需求。

根据第 4.2.3 节中的分析可知，一个 RDD 数据结构由上百万独立的数据对象 (Object)，而持久化的 RDD 将会长期存活于运行时堆 (Java heap)，直至被替换到磁盘中，而在此之前，每一次 GC 都会因遍历这些巨量的存活 RDD 数据而造成显著的 GC 开销。因此，可以将这些访问不频繁的，但重新计算开销巨大的 RDD 序列化后储存于 NVM 中。如图 4.12 所示，将约 40GB 的冷数据写入 NVM Off-Heap 时, 和全部使用 DRAM 相比，仅会带来 5% 的性能差，和内存不足，只有 32 GB DRAM 时相比性能仍旧提升了 96%。图 4.18 展示了此

时的 NVM 端内存带宽，其读带宽峰值仅在 500MB/s 左右，和 12.8GB/s 的峰值有相当差距。

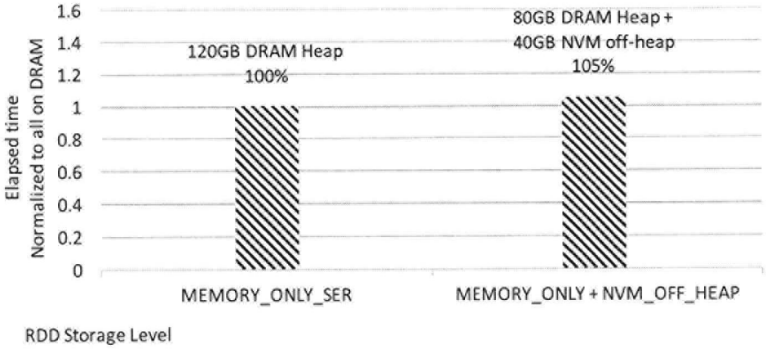


图 4.12　将容错 RDD 置于非易失性内存构成的 Off-Heap

Figure 4.12　Put fault tolerance RDD into NVM Off-Heap

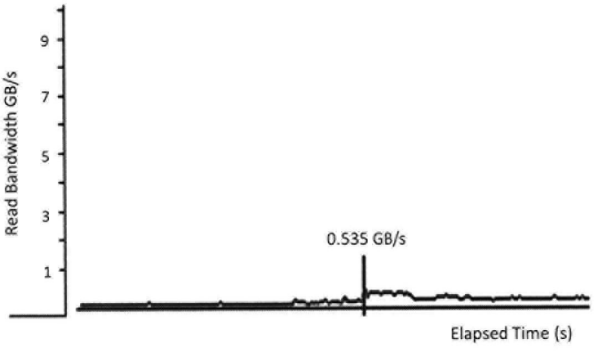


图 4.13　NVM Off-Heap 上的读带宽

Figure 4.13　The read bandwidth of NVM Off-Heap

由于内存计算应用的计算行为简洁、清晰，应用开发人员可以准确的在自定义数据结构粒度 (RDD 粒度) 识别出这些重新计算开销巨大的冷数据。如 Spark PageRank 源代码 1 所示，开发人员可以快速的判断出 RDD 变量 *links* 对应的数据几乎参加了每一次迭代的计算，而中间计算数据 RDD *contribs* 则仅仅参加了一次迭代。对于一个迭代数十次、数百次的计算过程来说，一旦分布式系统中的某个节点崩溃，在没有进行容错存储的处理下，重新计算丢失数据将会带来十分巨大的开销，而 RDD *contribs* 恰恰是这样的中间结果。另一方面，将其以非序列化形式置于内存中，又会带来明显的 GC 开销。因此，在内

存容量足够的情况，将其缓存于 NVM Off-Heap 是一个恰当的选择。

本文扩展了原有的 RDD Off-Heap 编程接口，如表 4.4 所示，以使其 Off-heap 部分可以同时支持 DRAM、NVM 部分，此处操作系统的支持工作使用了和 4.3.1节相同的机制。仅需对现有的应用，替换其冷数据的 Storage Level 便使其在异质内存中合理布局。

表 4.4 扩展后的 Off-Heap RDD 存储等级

Table 4.4 The extended Off-Heap RDD Storage Level

Storage Level	Description
DRAM_OFF_HEAP	Store the data of RDD to DRAM Off-Heap
NVM_OFF_HEAP	Store the data of RDD to NVM Off-Heap

4.3.3 持久化的热、较热数 RDD 的布局策略

对于频繁使用的 RDD 数据，以及访存“较频繁”的 Warm 状态的 RDD 数据，为了加速计算，用户往往对其进行持久化操作。对于该类计算较频繁数据，本节推荐将其置于运行时堆 (Java heap)，并保持在非序列化形态，以减少（反序列化）计算开销。然而，由于二者的访问频度差别，本节提供了相应的编程接口，来允许开发者在自定义数据结构粒度 (RDD 粒度) 将二者的数据分别置于运行时堆的 DRAM、NVM 区域。由于一个非序列化状态的 RDD，穿透三层数据抽象 (Spark RDD、Scala var/val、Java Object Model) 后，对应了运行时系统中上百万独立的数据对象，因此本文设计了一套“大数据框架-运行时系统协同机制”来对其数据数据进行管理。

为了提高编程接口的兼容性，本文通过扩展现有的 RDD Storage Level 来添加控制其位置 (DRAM、NVM) 的选项，如表 4.5 为扩展后的 MEMORY_ONLY 接口。完整的接口，以及具体使用、设计、实验分析，本文将在随后的第 5 章进行详细阐述。另外，本章的实验测试并不包含 Hot persist RDD、Warm persist RDD 的在异质内存上布局控制效果部分，仅以短生命周期 RDD、临时数据、冷持久化 RDD 布局为主进行性能测试。

另一方面，在自定义数据结构粒度（如 RDD 粒度）进行数据布局的策略被本文实现在运行时系统 (OpenJDK) 之中，并未依赖任何语言库，因此原则上该机制可以扩展到其他基于 Java 运行时系统 (OpenJDK) 执行的大数据框架中。本文将在以后的工作中对此进一步的讨论。

表 4.5 扩展后的 On-Heap RDD 存储等级示例

Table 4.5 The example of extended On-Heap RDD Storage Level

Storage Level	Description
MEMORY_ONLY_DRAM	Store the data of RDD to DRAM On-Heap
MEMORY_ONLY_NVM	Store the data of RDD to NVM On-Heap

4.4 实验和分析

4.4.1 实验平台

本章仍使用第3 章中提出的 NVM 仿真平台，然而，由于此次使用的为大数据应用，如 Spark PageRank, Spark Graphx 等，其内存使用量巨大，因此没有必要再压缩 CPU 的 LLC (Last Level Cache)。同时，该次实验中没有添加任何干扰程序来进一步压缩 NVM 端的带宽，因此 NVM 的带宽由 QPI (Intel QuickPath Interconnect) 限制，读、写带宽可以同时达到 12.8GB/s，最终的平台参数如表 4.6 所示。

表 4.6 非易失性内存的仿真参数

Table 4.6 The techniques of the NVM emulator

	DRAM	NVM
Read Latency(ns)	120	300
Bandwidth (GB/s)	29	Read 12.8 Write 12.8

本节使用两个节点来进行计算，一个用于任务调度、控制的 Spark Master 节点，一个实际用于计算的 Spark Slave 节点。服务器的配置信息如表 4.7 所示。在 Spark Slave 节点中，仅利用一个 CPU 的 16 个核心进行计算，使用其他节点的内存来模拟 NVM。

4.4.2 测试集和输入数据

该处的测试程序选用 Spark 自带的 Spark PageRank, Spark K-Means, 以及基于 Spark 开发的图计算框架 GraphX，来验证本文提出的编程框架。本文使用 Big Data Bench[80] 生成的数据集作为程序的输入集。

表 4.7 服务器信息

Table 4.7 The techniques of the servers

Spark Master			
CPU	X5650	2.67GHz	x2
Memory	DDR3	1333MHz	Dual Channel
Spark Slave			
CPU	E7-4809 v3	2.00GHz	x4
Memory	DDR 4	1867MHz	Dual Channel

4.4.3 实验测试与分析

Spark PageRank

首先进行实验测试的是 Spark PageRank, 该测试用例用于网页排序，是一个经典的离线分析程序，其计算类型为迭代计算。Spark PageRank 的核心算法如第 4章代码 1所示。由于测试中，使用 NVM 将主存扩展为了 128 GB，并将 Spark Executor 设置为了 120 GB，因此有足够的内存空间来缓存用于容错的 RDD。应用的 RDD 持久化类型如表 4.2 优化后的配置所示。

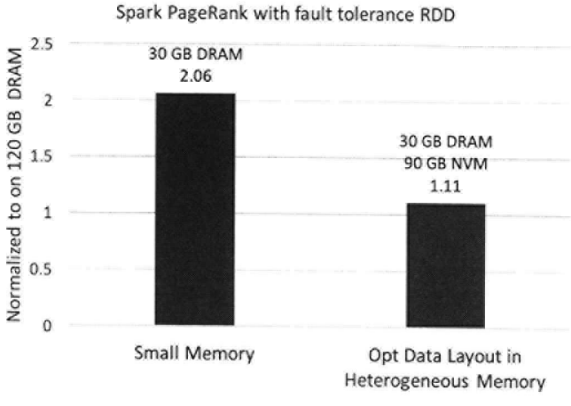


图 4.14 Spark PageRank 使用框架后的性能

Figure 4.14 The performance of Spark PageRank after applying own approach

Spark PageRank 在异质内存中的性能如图 4.14 所示。其纵坐标为程序的运行时间和 Spark Executor 配置为 120 GB DRAM 时的比例。图中标注的容量均为 Spark Executor 使用的容量，而非整机容量。如蓝色柱子表示了整机 32GB DRAM, Spark Executor 被配置为 30 GB 时的情况。从图中可以看到，当只有 30GB DRAM 时，程序执行时间比 120GB 时慢了 106%。而使用 90GB NVM 扩

展 Spark Executor，并使用本节提出的框架合理在异质内存上布置数据后，和使用 120 GB DRAM 只有 11% 的性能差距。

Spark K-Means

Spark K-Means 为通过离线分析进行聚类的程序，其计算行为亦为迭代计算。但是其内存使用的类型和 Spark PageRank 不同，其主要的数据由一个巨大的 RDD 构成，计算以迭代的形式在其上进行。而 Spark PageRank 的主要数据由数量众多、大小相近的 RDD 构成。因此，并没有足够的 DRAM 空间将 Spark K-Means 最主要的 RDD 置于 DRAM, 同时，其需要置于 Off-Heap 进行容错的数据也明显少于 Spark PageRank。

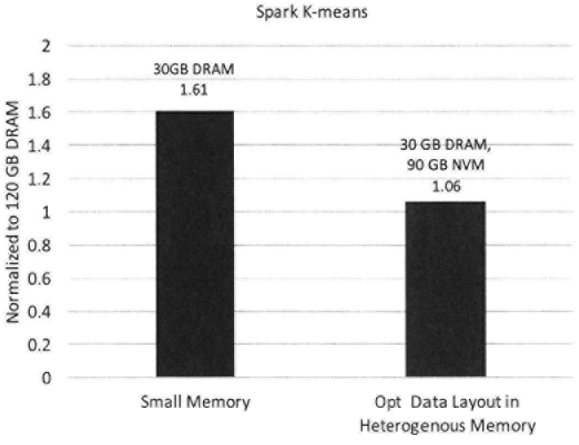


图 4.15　Spark K-Means 使用框架后的性能

Figure 4.15　The performance of Spark K-Means after applying own approach

在使用本章的编程框架进行数据布局后，Spark K-Means 的性能如图 4.15 所示。和内存充足 (120 GB DRAM) 时相比，30GB 堆大小的配置下 (整机 32GB DRAM)，程序性能下降了 61%，而当使用 90 GB NVM 扩展内存，并使用本章提出的框架布局 RDD 后，整体性能上升了 51.8%, 和全部使用 120 GB DRAM 只有 6% 的差距。由于 Spark K-Means 的主要 RDD 巨大，其会被很快的移动到旧生代区域，而该实验配置中，旧生代区域几乎全部在 NVM 中。这说明，Spark K-Means 在其主要 RDD 上的计算并不是延迟敏感性，同时 NVM 的带宽也基本可以满足其需求。经过分析发现，Spark K-Means 的计算需求量很大，每次取一个数据后（图顶点），都需要计算该顶点和各个中心点之间的距离，并选出距离最小的中心点，将其归入对应的类。

Spark GraphX - ConnectedComponents

除了一些基本的应用外，还有一些基于 Spark 开发的框架，如用于图计算的 GraphX 。为了验证该研究提出的编程框架的适用范围，本文测试了其对 GraphX 的支持。由于 GraphX 上应用的计算均为直接调用 GraphX 提供的接口，其计算行为类似。因此，这里选取了其中的一个应用，ConnectedComponents (CC)，来进行测试。测试配置和 Spark PageRank，Spark K-Means 类似。

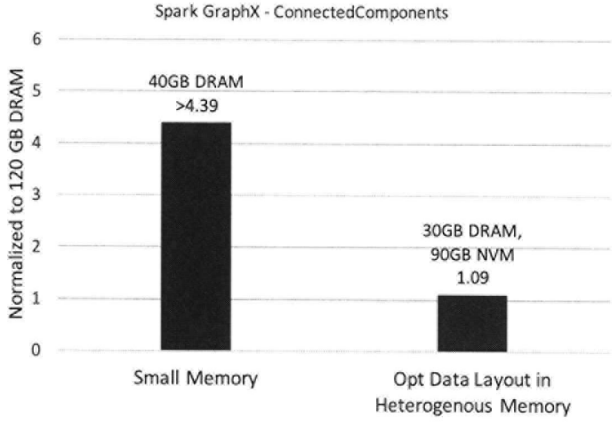


图 4.16　Spark GraphX - CC 使用框架后的性能

Figure 4.16　The performance of Spark GraphX - CC after applying own approach

其性能如图 4.16 所示，当整机内存只有 32GB DRAM，并将 Spark Executor 配置为 30GB 时，GraphX CC 无法完成计算。随后，整机内存增加到 64GB，并将运行时堆 (Java heap) 大小扩展为 40GB 后，GraphX - CC 仍旧需要大量时间来完成计算。相比于整机 128GB DRAM，120GB 堆大小的配置，其执行时间至少增长了 339%，可以认为几乎无法完成计算过程。当使用 90 GB NVM 对运行时堆进行扩展，并应用本章的框架进行数据布局后，和内存不足时相比，程序加速了 300% 以上, 和使用 128 GB DRAM，120 GB 堆大小配置相比，仅有 9% 的性能差距。

本研究提出的布局策略虽然较为初级，但从实验来看，其成功使用了 NVM 来扩展主存，由于 NVM 的存储容量更大，功耗更低，使用其扩展服务器的内存后，大数据框架可以处理更大规模的运算，如 GraphX-CC 的测试过程。在使用异质内存时，应用本章提出的编程框架后，在异质内存中的比例仅有 25-30% 的情况下，便可以达到和使用等量 DRAM 相近的性能。

4.4.4　进一步的测试与分析

加州大学伯克利分校的 Ousterhout 等人认为，目前大数据计算框架的性能瓶颈仍旧在处理器、内存等计算能力上，网络传输造成的性能下降一般只在 2% 左右，针对大数据优化的重点还是应该在处理器、内存、磁盘等计算方面[79]。因此，本节的探讨仍旧是单机节点的内存性能对处理器计算行为的影响，以及内存性能和计算规模之间的关系上。本节关注的核心问题为，如果将应用的计算规模进一步增大，如 NUMA 结构中单节点的内存容量增长到 TB 级别，那么性能不及 DRAM 的异质内存会不会成为新的性能瓶颈，本节提出的框架是否会继续有效。针对该问题，本小节主要从 NVM 的延迟、带宽两方面进行分析。

非易失性内存高访存延迟的影响

当程序的 LLC Miss 数目一定时，内存的物理访存延迟越大，处理器暂停的时间 (CPU Stalls) 便会成比例的增加[34]，该理论也是一些 NVM 延迟模拟的理论基础。因此，为了探讨当计算规模增大时，NVM 高访存延是否会带来严重的性能瓶颈，本节监测在固定的硬件配置下，以及相同的程序中，NVM LLC Miss 是否会随着计算规模的增大迅速增加。

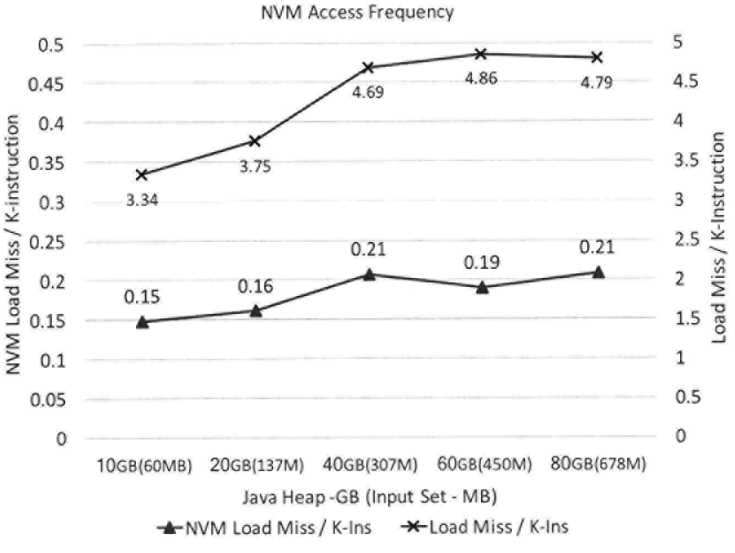


图 4.17　NVM 访存频率和计算规模之间的关系

Figure 4.17　The correlation between calculating scale and NVM access frequency

图 4.17 展示了应用 Spark PageRank 中 NVM 访存频率和计算规模之间的

相关性。其中，左侧纵坐标表示了每千条指令中的 NVM 读访存 (NVM LLC Load Miss) 次数，右侧纵坐标表示了每千条指令中的总的读访存 (LLC Load Miss) 次数，而横坐标表示了堆大小和输入集逐渐增大的过程。从图中可以看到，在运行时堆 (Java Heap) 和数据集以指数级增长的过程中，NVM 访存和总访存之间的比例稳定在 4% - 5% 左右，并没有急剧增加。也就是说，NVM 的访存比例并不会随着计算规模的增加而快速的增长，因此 NVM 的高访存延迟对程序的影响，仍旧是在一个稳定的比例下。

另一方面，每千条指令中，NVM 读访存数目和总读访存数目只有在运行时堆 (Java Heap) 从 20 GB 扩展为 40GB 时有比较显著的增长，分别增长了 31.25% 和 25.1%，而在其他情况下，二者的波动都很小。经过分析认为，这是由于当运行时堆 (Java Heap) 从 20GB 扩展为 40GB, 输入集大小也翻倍后，该服务器的 Cache 无法再应对该规模的计算，因此 LLC Miss 会显著增加。而在此后的计算规模增大的过程中，Cache 的影响也趋于稳定。经过以上分析，可以初步认为对于特定的应用，如 Spark PageRank, 和特定的数据布局下，NVM 访存延迟对程序性能的影响不会随着计算规模的快速增长而迅速增加。

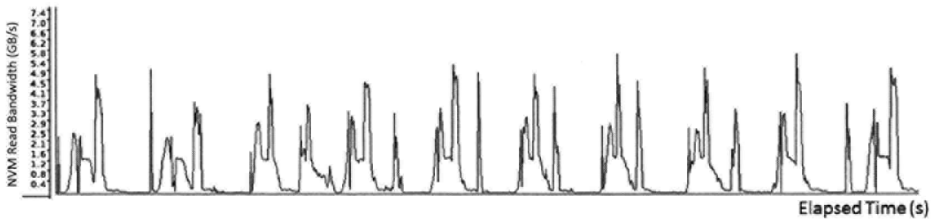
非易失性内存低带宽的影响

根据目前的资料显示，NVM 的访存带宽较 DRAM 较小。但有一些资料认为，NVM 的带宽可以通过工艺水平提高到和 DRAM 相似的水平。本文着眼于当前的可用资料，认为 NVM 的访存带宽仅为 DRAM 的 1/2 - 1/6 左右，因此在这里探讨，使用本节提出的框架进行合理的数据布局后，随着计算规模的增大，NVM 的带宽是否会再次成为整个程序的性能瓶颈。

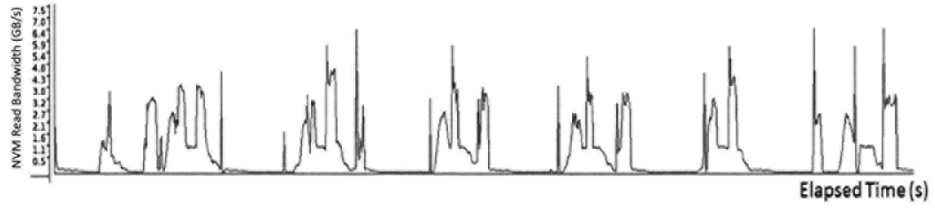
图 4.18 展示了 Spark PageRank 在应用了本文的编程框架后，NVM 读带宽和计算规模之间的关系。Java Heap 从 40 GB 增长到 80GB，输入集也等比放大的过程中，NVM 之上的带宽仅轻微增加，和其 12.8GB/s 的读写带宽值有相当大的差距。因此可以认为，在合理的数据布局下，NVM 部分的带宽可以应对近期大数据计算规模的增长。如果大数据应用的规模快速增长，确实可能带来新的性能问题。届时，我们将会研究这些大规模计算的行为，并提出新的解决方案来应对该问题。

平衡序列化开销和 GC 开销

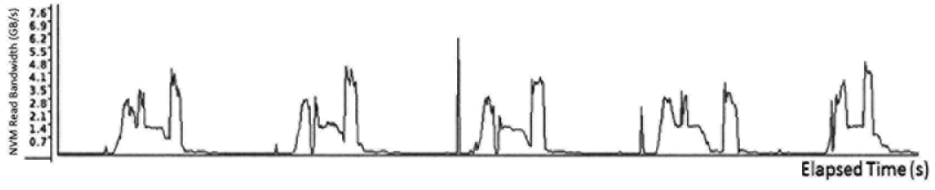
正如前文所述，当将一些容错 RDD 置于 NVM Off-Heap 时，由于需要对



(a) 40GB Java Heap, input set 307MB



(b) 60GB Java Heap, input set 450MB



(c) 80GB Java Heap, input set 670MB

图 4.18　NVM 读请求访存带宽在不同计算规模下的变化

Figure 4.18　NVM memory read bandwidth variation on different calculating scale

这些数据进行序列化处理，因此会带来一定的计算开销。根据 [79] 的分析，当今大数据的性能瓶颈也主要在 CPU 的计算性能上。然而，根据在 4.2.3 中的分析可知，如果将这些容错 RDD 直接以非序列的形式置于运行时堆 (Java Heap) 将会带来接近 30% - 50% 的 GC 开销。对于 Spark PageRank 应用来说，将这些容错数据序列化后，带来了 81% 的性能提升。因此，开发人员可以首先通过，OpenJDK 输出的日志、Spark 日志等快速判断一个程序中 GC 的开销。如果 GC 时间仅仅占据了运行时间很小的比例，那么确实没有必要将一些 RDD 置于 NVM Off-Heap，即使有加速比，也会很小。该判断过程并不需要借助于专业的分析工具，使用 OpenJDK，Spark 自带的日志输出信息便可判断。因此，该问题不会导致本文框架的使用复杂性。

4.5　本章总结

本章经过对以 Spark 为代表的内存计算框架的分析，深入了解了该类应用的计算行为、数据使用特点，并通过分析上层应用和运行时之间的交互，分析

了内存不足时造成性能显著下降的原因。在第3章结论的基础下，结合本章进行的分析，提出了一个基于 OpenJDK 和 Spark 的异质内存管理策略。通过给应用开发者提供简洁、易用的编程接口，便可让应用开发者合理在 DRAM 和 NVM 之间以自定义数据结构粒度 (RD) 进行数据布局。同时，该接口的使用无需重新撰写原有应用，而仅仅是对持久化 RDD 的 Storage Level 进行一些简单的替换，并通过命令行操作运行时的生代布局。虽然本章节提出的数据布局策略较为初步，但是可以在 Spark PageRank, Spark K-Means 和 Spark GraphX 框架上取得良好的效果。本文将在第5章中，针对内存计算应用提出更加精准的异质内存管理策略。

第 5 章 Spark - JVM 协同的一体化异质内存编程框架

随着数据中心、大数据计算等应用的发展，其对内存容量的需求快速增长。而为了进一步加速计算、容错过程，将大量数据缓存到内存中的内存计算框架应运而生，如 Spark，这进一步加大了应用对内存的需求。由于 DRAM 工艺的限制，其容量、功耗难以满足应用持续增长的需求，使用由 DRAM 和 NVM 构成的异质内存来解决内存计算对大容量内存的需求是一个研究重点。但由于大数据应用的内存使用量巨大，传统的单一粒度的数据分类、布局策略可能会带来显著的管理开销。因此，本文尝试利用程序语义，在多层次进行数据的划分、布局。

为了追求跨平台特性，这些分布式的内存计算框架多为基于托管式语言开发，如 Scala、Java 等。托管式运行时具有独立的内存管理机制，这给异质内存数据布局管理带来了新的挑战，如 GC 对操作系统、硬件数据布局的干扰，如图 1.2 所示。本章在前述章节的基础上，进一步探讨如何利用运行时系统的特点在更加恰当的粒度进行数据的冷热划分，以及低开销的将数据布局到异质内存。

第 3章探索了利用运行时系统对异质内存进行管理的可能性。而第 4章探讨了内存计算应用 (Spark 应用) 的计算行为和内存使用特点，并提出了为开发者提供必要的编程接口，使其可以在自定义数据结构粒度 (RDD 粒度) 进行数据布局的策略。本章基于前述研究的成果，提出了一套 Spark - JVM (Java Virtual Machine) 协同合作的策略来解决 RDD 粒度数据布局的问题：将上层应用 (Spark 应用) 的语义信息传递到底层运行时系统 (OpenJDK)，以便运行时系统可以在复杂粒度 (RDD 粒度) 进行数据的冷热划分、布局，并可根据大数据应用的特点定向的优化运行时系统的 GC 机制。最后，本章整合了前述的数据布局管理策略，提出了一体化的异质内存管理机制，并将其实现在了内存计算框架 Spark 与运行时系统 OpenJDK 之中。

5.1 研究动机与概要

以 Spark 为主的内存计算应用，具有高并发性和良好的容错性。同时，其数据操作过程也十分简洁，应用开发人员可以根据源码快速的判断出数据 (RDD) 的冷热与生命周期。如第 4 章中的代码 1 展示了 Spark PageRank 的源码，而图 1.3 展示了 Spark PageRank 的计算过程。从中可以很清晰的看到 RDD *links* 具有贯穿整个应用的始终的生命周期，同时参与了几乎每一次迭代 (Stage) 中的计算。而 RDD *contribs* 在内存足够时，会被持久化存储到内存中。直到内存不足时，这些用于容错的 RDD 可能会被替换到磁盘，或者直接删除掉，结束其生命周期。除此之外，中间计算过程中，还有大量的临时 RDD 存在。

本文在第 4 章中提出了，由应用开发人员在自定义数据结构粒度 (RDD 粒度) 进行数据布局的思想，并提供了部分编程接口的设计和布局策略。本章针对其中的关键问题–“如何将上层应用程序语义传递到底层运行时系统中的孤立数据对象”的解决方案进行了详细的阐述和分析，并完成了所有相应编程接口的设计。

```
Handler object : array, [Lscala/Tuple2;
  depth[0]:obj scala/Tuple2
    depth[1]:obj java/lang/String                          // key
      depth[2]:array [C
    depth[1]:obj spark/util/collection/CompactBuffer       // value
      depth[2]:array, [Ljava/lang/String;
        depth[3]:obj java/lang/String
          depth[4]:array [C
        depth[3]:obj java/lang/String
          depth[4]:array [C
    .......
  depth[0]:obj scala/Tuple2
    .......
```

图 5.1 Spark RDD 的构成

Figure 5.1 The organization of Spark RDD

RDD 为 Spark 层次的基本计算和容错单元，但在运行时系统层次，其为由众多独立数据对象 (Object) 构成的群体，如图 5.1 所示，为一个处于非序列化

形式的 Spark RDD 的结构图。RDD 的数据对象成员结构关系类似于一个子图，其顶点为一个数组，Object Array。该 Object Array 的长度由应用的输入集决定，在输入集为 700MB 时，其长度可以达到几百万。同时，该 Object Array 的每一个元素都将引用其他的数据对象，这种引用关系将会依次传递数层。如，该 RDD 从最顶层的 Object Array 到最底层的 Char 数组 ([C array)，一共有 6 层之多。这便会造成 Spark 层次的一个 RDD 数据单元，在运行时系统中对应上百万的独立数据对象 (Object)。而运行时系统在处理这些数据时并不会感知到这些数据对象 (Object) 之间的关系。因此，也就难以将 RDD 的冷、热属性，生命周期属性传递到这些数据对象 (Object) 中，也就难以利用 GC 对这些数据对象 (Object) 进行识别和布局。

然而，经过前述的研究发现，GC 本身就是一个并发的图遍历过程。因此，本章提出可以利用 GC 去识别出一个 RDD 所属的所有数据对象 (Object)，并按照 RDD 的属性来划分、处理这些数据。首先，这就需要一种信息传递机制，允许开发人员将重要的信息从大数据应用传递到运行时系统。然后，由 GC 获取这些标注信息，并在扫描过程中将标注信息传递到 RDD 所属的所有数据对象。最后，在 GC 移动数据的阶段，根据每个数据对象的分类，将数据布局到对应的 DRAM、NVM 区域，或者进行特定的机制优化。

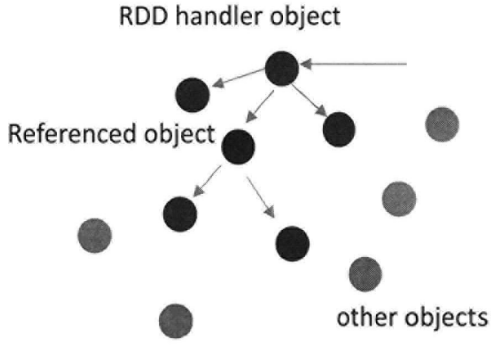


图 5.2　利用 GC 识别出构成 RDD 的所有数据对象

Figure 5.2　Utilize GC to recognize all the related objects belonging to a RDD

图 5.2 展示了利用 GC 识别一个 RDD 所属数据对象 (Object) 的过程。RDD 包含的数据对象 (Object)，通过引用关系 (Object Reference) 构成了一个子图。当 GC 获知该子图的根节点 (RDD handler object) 后，便可以在扫描 (Trace) 过程中，通过引用关系 (Object Reference) 来识别出其包含的的所有数据对象 (Object)。

同时，每当发现一个对应的数据对象 (Object) 时，可以将子图顶点 (RDD handler object) 所具有的数据属性传递给该数据对象 (Object)。然后，GC 在移动数据的过程 (Object Promotion、Compact) 中便可以根据已经划分好的分类进行数据的布局。

最终，基于前述的所有研究，本章提出了一个 Spark 和 JVM (Java Virtual Machine) 协同合作的编程框架。本文使用的 JVM 为 OpenJDK 内部的 HotSpot，在本章后续的表述中，JVM 均指 HotSpot[17]。此外，运行时堆 (Java Heap) 的新生代区域按照第 4 章中的结论，依然被映射到 DRAM，而其旧生代区域被划分为 DRAM 区和 NVM 区两部分，分别被称为 Object Space 和 Dram Space。利用本文开发的信息传递通道机制，Spark 应用开发者可以将上层应用的数据使用信息传递到底层运行时系统，此时，改良的运行时系统便可根据获得的信息进行自定义结构粒度 (RDD 粒度) 的数据划分、布局、应用定向优化。

5.2 Spark - JVM 协同编程框架的设计概要

Spark 应用开发人员看到的数据单元是一个个独立的 RDD，其无法感知 RDD 在 JVM 中对应的众多数据对象 (Object)。而且，多数数据对象 (Object) 的创建过程均在 Scala 库中进行的，同时由于数据对象 (Object) 数量众多，所以不可能由开发人员去手动标注所有的这些数据对象 (Object)。

为了解决该问题，本文设计了一套 Spark - JVM 协同工作的数据信息标注和传递策略。Spark 应用开发人员只需要针对 RDD 进行单一的属性标注，如将其置于 DRAM 或 NVM，Spark 框架、OpenJDK 系统便会自动传递标注属性，并按照属性划分、布局数据。本文将一个 RDD 划分为两部分：作为根节点的 RDD handler object (array)；以及其引用的所有的 Data object。如图 5.2 所示，红色的根节点为 RDD handler object，深灰色的节点为引用的 Data object。Spark 应用开发人员对一个 RDD 的属性进行指定后，修改后的 Spark 框架会将该属性写入对应的 RDD handler object。而后在 OpenJDK 层次，GC 会识别出具有特殊属性的 RDD handler object, 并在 GC 扫描 (Trace) 过程中将该属性传递到其引用的 Data object。

整个 Spark - JVM 的架构如图 5.3 所示，从中可以看到，Spark 和 JVM 之间通过实例化的 RDD (Materialized RDD) 连接。这是因为，RDD 是一个 Spark 层次的抽象的数据概念，只有特定环境下，一个 RDD 对应的数据才会被实际

计算出来并被保存在内存中。很多情况下，RDD 仅仅是 Spark RDD 谱系图中的一个数据结构抽象，用来维护容错性和计算流程。

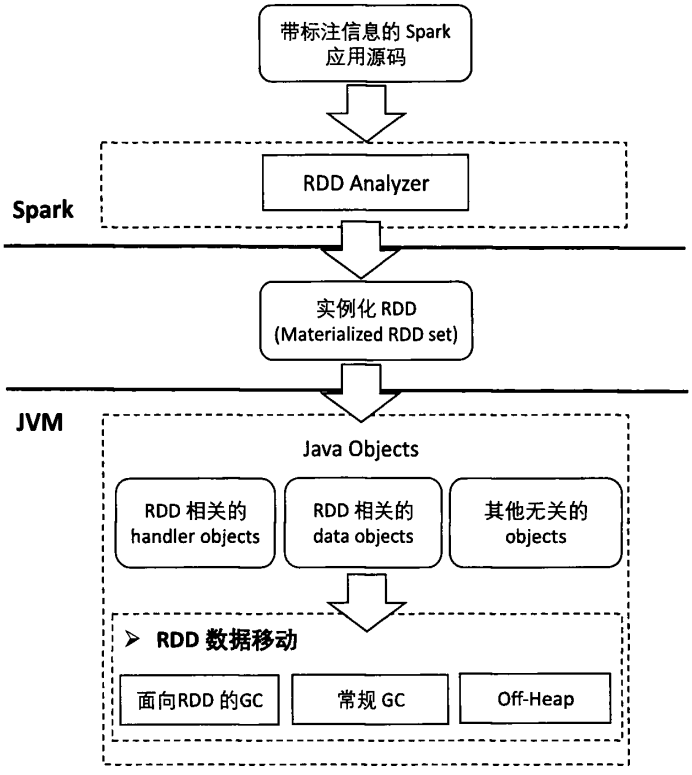


图 5.3 Spark - JVM 异质内存编程框架

Figure 5.3 Overview for the Spark - JVM framework for heterogeneous memory

当 Spark 应用开发人员利用本文提供的接口写好程序后，Spark 会读取该应用的源程序，分析出被开发人员标注的 RDD，并调用本文开发的 JVM 接口，将属性信息系写入相关实例化 RDD 对应的 Handler object，该信息指示了 RDD 的位置属性，DRAM 或 NVM。当 GC 被触发后，扫描 (Trace) 过程中便会将所有存活的数据对象分成三类，被标记的 RDD handler object、其引用的数量众多的 RDD data object、以及其他的无关的 object。同时，在数据移动阶段 (MinorGC object promotion)，具有 DRAM 属性的数据对象就会被移动到旧生代对应的 DRAM 区域，而无属性或具有 NVM 属性的 RDD 则会被迁移到对应的 NVM 区域。此外，该框架仍旧支持第 4 章中提出的 NVM Off-Heap 接口。操作系统需要向运行时系统暴露异质内存信息，以便运行时系统可以灵活的申请 DRAM、NVM 空间。本章仍旧利用第 3章、第 4 章中所用的 mbind 机制，

以及基于 NUMA 的异质内存仿真平台来完成 DRAM、NVM 空间的申请。

最终，本章提出一套 Spark - JVM 协同工作机制，在应用自定义的数据结构粒度 (RDD 粒度) 管理异质内存中的数据布局。为了使本章的工作具有更广的适用范围，主要的机制被实现在了 JVM (OpenJDK) 中，并根据 Spark 的计算特点在其内部调用了相应的 JVM 接口。该机制并不依赖于 Scala 等任何一种语言，可以直接应用于可以运行在 JVM 上的大数据编程框中。

5.3 异质内存编程接口的设计和使用

为了易用性，本文提出的编程框架并没有提供新的 RDD 标记方式，而是将 RDD 的 DRAM、NVM 信息扩展到原有的 Storage Level 上。当开发人员认为一个 RDD 为热数据，需要将其持久化时，可以给予其 DRAM 标注。亦或是，当开发人员认为需要将一个计算不太频繁的“Warm RDD”持久化时，可以赋予其 NVM 标志。

表 5.1 为本文提供的编程接口，其为对一些现有的 RDD Storage Level 添加了对应的 DRAM、NVM 信息。Spark 应用开发人员可以根据对程序的理解，赋予对应 RDD 以位置信息。该节以常用的 Spark PageRank 和 Spark K-Means 为例来解释带有位置信息 (DRAM、NVM) 的编程接口用法。

图 5.4 展示了修改原有 Spark PageRank 源码，使其 RDD 具有位置信息的过程。该例子仅对 Spark PageRank 的源码进行了两处修改，而且仅仅是给源码中原有的 Storage Level 添加了 DRAM 或者 NVM 位置信息。原有代码中，RDD *links* 因为参加了每一次 Stage 的计算，因此被以非序列化形式，持久化储存到运行时堆中。而在修改后的新版代码中，开发者应该赋以其 DRAM 标注信息。而 RDD *contribs* 仅参加了一次 Stage 内部的计算，但是由于其为迭代的中间计算过程，因此被序列化后置于 Off-Heap 用于容错。根据第 4.3.3 节中的分析，这类容错 RDD 被置于 NVM 便可，并不会带来明显的性能波动。经过两处修改，原本对异质内存没有任何支持的 Spark PageRank 便可以合理的将数据布局到异质内存中。本节将在 5.7 节中对其性能进行测试。

图 5.5 展示了将本文提供的异质内存编程接口应用于 Spark K-Means 源码的用例。从该图中可以看到，仅对 Spark K-Means 源码进行了一处修改，其中，旧版代码中原有的“cache()”对应了“persist(StorageLevel.MEMORY_ONLY)”状态，因此，我们仅仅是给其增加了 DRAM 位置信息。此时，RDD *data* 为

表 5.1 经过扩展后带位置信息的 RDD Storage Level

Table 5.1 The extended RDD Storage Level with layout information

Storage Level	Description
MEMORY_ONLY	将 RDD 以非序列化形式存储到运行时堆，GC 时优先移动到 NVM 区。
MEMORY_ONLY_NVM	和 MEMORY_ONLY 相同。
MEMORY_ONLY_DRAM	将 RDD 以非序列化形式存储到运行时堆，GC 时优先移动到 DRAM 区。
MEMORY_AND_DISK	将 RDD 以非序列化形式存储到运行时堆，GC 时优先移动到 NVM 区。堆空间不够时，替换到磁盘。
MEMORY_AND_DISK_NVM	和 MEMORY_AND_DISK 相同。
MEMORY_AND_DISK_DRAM	将 RDD 以非序列化形式存储到运行时堆，GC 时优先移动到 DRAM 区。堆空间不够时，替换到磁盘。
MEMORY_ONLY_SER	将 RDD 以序列化形式存储到运行时堆 (每个 Partition 为单一 Byte Array)，GC 时优先移动到 NVM 区。
MEMORY_ONLY_SER_NVM	和 MEMORY_ONLY_SER 相同。
MEMORY_ONLY_SER_DRAM	将 RDD 以序列化形式存储到运行时堆 (每个 Partition 为单一 Byte Array)，GC 时优先移动到 DRAM 区。
MEMORY_AND_DISK_SER	和 MEMORY_AND_DISK, 但是以序列化形式存储于运行时堆。
MEMORY_AND_DISK_SER_NVM	和 MEMORY_AND_DISK_SER 相同
MEMORY_AND_DISK_SER_DRAM	和 MEMORY_AND_DISK_DRAM 相同, 但是以序列化形式存储于运行时堆。
DRAM_OFF_HEAP	将数据存储于 Off-Heap 区，优先使用 DRAM。
NVM_OFF_HEAP	将数据存储于 Off-Heap 区，优先使用 NVM。

Spark K-Means 的主要数据结构，不但多数计算均在其上进行，而且其容量十分巨大。因此，当运行时堆 (Java heap) 中的 Dram Space 无法容纳整个 RDD *data* 时，剩余的数据将会以非序列化状态存储于运行时堆中的 NVM 部分。

至此，可以看到，仅需对现有 Spark 程序中的 Storage Level 进行 DRAM、NVM 标注便可，并不需要进行复杂的分析。每个 RDD 均可以被指定 DRAM 或者 NVM 位置信息，随后该信息会被传递到对应的 RDD handler object 以及 RDD data object 之上。首先在 Spark 层次，标注信息将会在一个 Stage 内部的，具有数据依赖关系的实例化 RDD (Materialized RDD) 之间进行传播，传播的具体策略将在 5.4节中进行详细介绍。而标注信息在 JVM 层传播时，会由于共享数据对象 (RDD shared objects) 的存在产生一些冲突，而本章在 5.5节中给出了

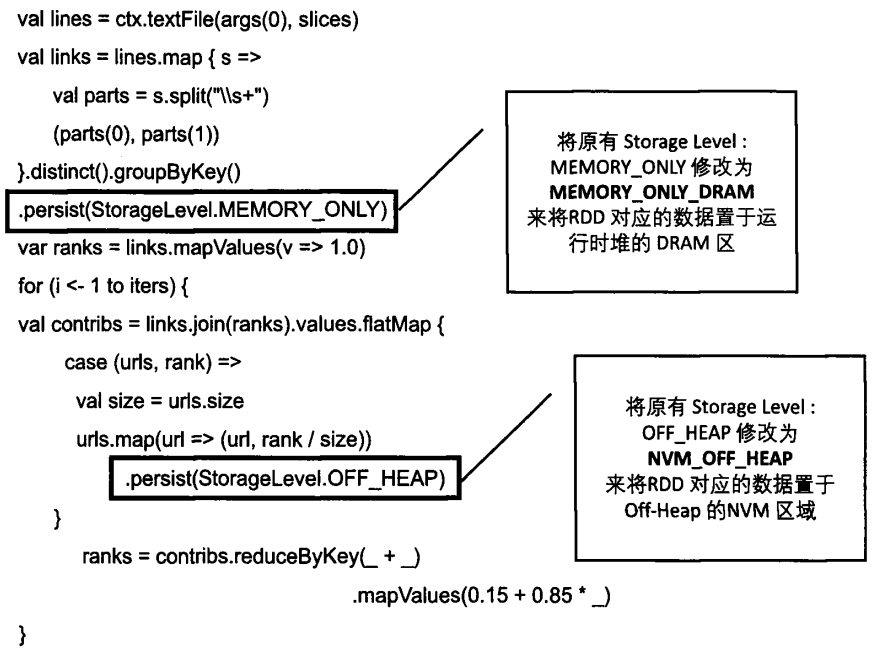


图 5.4 Spark PageRank 使用编程接口的示例

Figure 5.4 The programming interface usage example for Spark PageRank

以“DRAM first”策略为核心的的冲突解决方案。

5.4 标注信息在 Spark 层次的传播

大数据系统均为基于多层软件栈开发。如 Spark 基于 Scala 语言开发，而 Scala 语言下层还有运行时 (Runtime system), 而运行时 (Runtime system) 基于操作系统运行。这种多层软件栈的设计虽然带来了编程的简易型、代码的可读性，然而却带来了数据管理的复杂性。在 Spark 层次，其基本的数据单位为 RDD，为只读的单一数据抽象。而在 JVM 层次，其不但对应了众多独立的数据对象 (Object)，而且，由于 RDD 的“copy on write”特性，不同 RDD 甚至可能共用不同的数据对象 (Data object)。这便给 RDD 粒度的数据布局带来了复杂性。本文在 Spark、JVM 两个层次设计了协同机制来解决该问题，分别为 Spark 层次的 RDD handler object 标记，和 JVM 层次的 RDD data object 标记。并仅对 Spark 应用开发人员暴露出了简洁、易用的编程接口。首先，本节将介绍 Spark 层次的设计方案，也即标注信息在 RDD handler object 上的传递过程。

在一个 RDD 变量被 Spark 应用开发人员标注后，需要在 Spark 框架内部分析、标注所有相关的 RDD handler object. 该分析过程需要借助于 Spark 的

```
val lines = spark.read.textFile(args(0)).rdd
val data = lines.map(parseVector _).cache()
val K = args(1).toInt
val convergeDist = args(2).toDouble
val kPoints = data.takeSample(withReplacement = false, K, 42)
var tempDist = 1.0

while(tempDist > convergeDist) {
 val closest = data.map (p => (closestPoint(p, kPoints), (p, 1)))
 val pointStats = closest.reduceByKey{case ((p1, c1), (p2, c2)) => (p1 + p2, c1 + c2)}
 val newPoints = pointStats.map {pair =>
  (pair._1, pair._2._1 * (1.0 / pair._2._2))}.collectAsMap()
 tempDist = 0.0
 for (i <- 0 until K) {
  tempDist += squaredDistance(kPoints(i), newPoints(i))
 }
 for (newP <- newPoints) {
  kPoints(newP._1) = newP._2
 }
 println("Finished iteration (delta = " + tempDist + ")")
}
```

将原有 Storage Level：cache()/persist() 修改为 **MEMORY_ONLY_DRAM** 来将RDD 对应的数据置于运行时堆的 DRAM 区

图 5.5 Spark K-Means 使用编程接口的示例

Figure 5.5 The programming interface usage example for Spark K-Means

Stage 计算行为和 RDD 谱系图 (Lineage) 来完成标注。本文在 Spark 内添加了一个名为“RDD Analyzer”的模块来完成该分析过程。如图 5.6 是一个 RDD lineage 的例子。Spark 的计算过程会被划分为一些独立的 Stage，划分的依据为 RDD 之间的依赖关系。当一个 RDD 和其父 RDD 之间是宽依赖时 (Wide/Shuffle dependency)，那么此为一个 Stage 的开始。这些宽依赖会导致 Shuffle 计算，其中间结果需要写入磁盘来进行多进程/线程间的数据共享、容错。该图中，每一个方块表示一个 RDD partition, 而多个临近的方块表示一个完整的 RDD。灰色的方块表示该 RDD 被开发人员持久化 (persist) 置于内存中，而蓝色的 RDD 表示没有被应用开发人员显著标注的 RDD。

在“RDD Analyzer”确定哪些 RDD 对应的 handler object 需要被标记后，其便会通过调用本文在 OpenJDK 8 中添加的 API 来进行该过程。API 的使用方式如图 5.7 所示。经过标记的 RDD handler object 将会被 JVM 识别，并在数据分配、GC 时进行对应的特殊处理，第 5.5节将对此进行详细介绍。该标注函数基于 OpenJDK 开发，可以应用于任何基于 JVM，即 OpenJDK 中的 Oracle

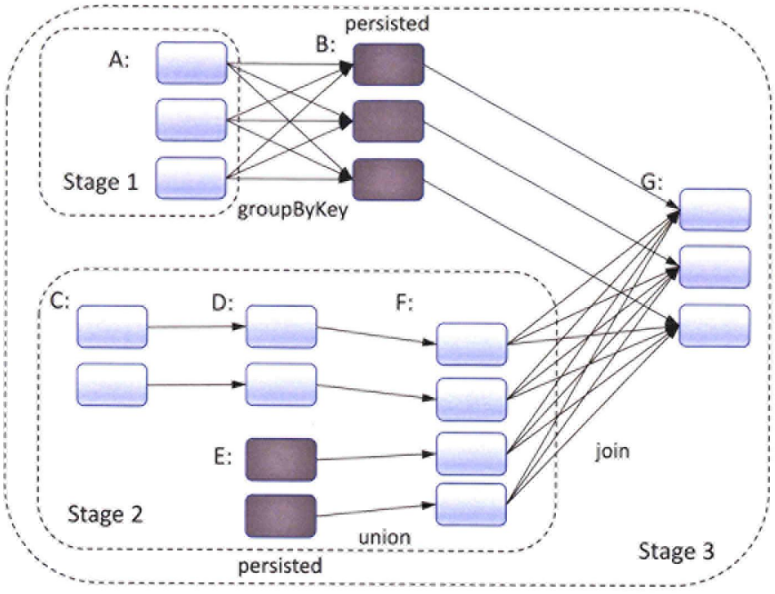


图 5.6 一个 RDD 谱系图

Figure 5.6 The example for a RDD lineage graph

HotSpot，运行的框架中。

```
// change Java thread  to DRAM state.
Platform.enter_rdd_region()
_array = new Array[V](initialSize)
// change Java Thread to normal state.
Platform.leave_rdd_region()
```

图 5.7 运行时提供的 RDD handler object 标记函数

Figure 5.7 The JVM interface used for marking RDD handler object

本节将在接下来的内容中介绍，“RDD Analyzer”模块识别到用户对 RDD 添加的位置信息 (DRAM、NVM) 后，如何确定哪些相关的 RDD handler object 需要被进行标记。

5.4.1 RDD Analyzer 的信息传递策略

本节以图 5.6为例，阐述“RDD Analyzer”将标注信息在 Stage 内部的 RDD 之间的传播方式。当开发人员将 RDD *F* 标注为 DRAM 时，如 MEMORY_ONLY_DRAM，根据其“copy on write”的计算特征，其数据可能来自于其任何一个父 RDD，抑或是全部经过计算生成的新数据，而我们很难通过 Spark 框架精确的分析出这些信息。为了保证 RDD *F* 所关联的数据能被 GC 顺

利的迁移到旧生代对应的 DRAM 区域，这里采取的方案是将其所有依赖的实例化 RDD (Materialized RDD Set)，均标记为和其具有相同的位置信息（DRAM or NVM ）。在该例中，对应的 RDD *C*，RDD *E* 也会被标注为相应的 DRAM 标志。需要注意的是，RDD 的标注信息仅会在一个 Stage 内部传递，且“RDD Analyzer”标注策略的核心思想为 - 让带有 DRAM 标志的 RDD 的后续计算均在 DRAM 之上进行。

这里并没有处理 RDD *D* 是因为，在 Spark 的实际 Pipeline 计算过程中，RDD *D* 并没有被实例化。其仅仅是 RDD *C* 到 RDD *F* 并行计算过程中的中间一步计算，该计算过程可以用公式 $f_F(f_D(f_C(Input\_from\_disk))+RDD\ E)$ 中 f_D 表示。此时，即使 RDD *D* 产生一个新的数据对象 (Data object)，$RDD_D.item$，其会产生在位于 DRAM 的新生代区域 (Young Generation)，而后便会立刻基于该数据对象 (Data object) $RDD_D.item$，进行 f_F 计算生成新的数据对象 $RDD_F.item$。数据 $RDD_D.item$ 会很快死亡，并且被正常的回收，而数据对象 $RDD_F.item$ 则会跟随 RDD *F* handler object 被 GC 正确的标注。如果 f_F 并没有产生新的数据对象，而是直接将原有的 $RDD_D.item$，传递到 RDD *F*，那么，这些数据对象也会跟随 RDD *F* handler object 被 GC 正确的标注，仍然无需通过 Spark 去处理 RDD *D* handler object 。

被实例化的 RDD 的计算规则与上述“Pipeline”方式不同，在生成其所有的数据后，才会开始进行下一个 RDD 的计算。如实例化的 RDD *E*，需要生成整个 RDD *E* 的 Data objects 后，才会根据其去计算 RDD *F* 对应的数据。而在计算 RDD *E* 的过程中，一旦多次产生 GC (MinorGC)，其数据对象 (Data object) 便可能会被移动到旧生代区域的 NVM 部分。此时，如果 RDD *F* 继承了这些数据对象，即使对这些继承的共享数据对象 (Shared data object) 正确标注，也无法再次利用高频的 MinorGC 将其布局到 DRAM 区域。因此，“RDD Analyzer”会去标注被实例化的 RDD，也即“Materialized RDD set”中的 RDD handler object，来保证涉及的数据对象 (Data objects) 可以被正常迁移到 DRAM 区域。

由于 RDD *C* 为第一个 RDD，其会从磁盘中读取数据，构建一个被称为 ShuffledRDD 的实例化 RDD。此时会计算出其所对应的 RDD 数据，也即其所属的 RDD data object 。因此 RDD *C* 对应的 handler object 也会被标注为 DRAM 标志，保证后续的计算发生在 DRAM 中。此外，中间计算过程中的临时数据

都会生成在新生代区域 (Young Generation)，其亦在 DRAM 区域。随后的第 5.5 节将会在 JVM 层次分析如何处理 RDD Data object 的标记问题。

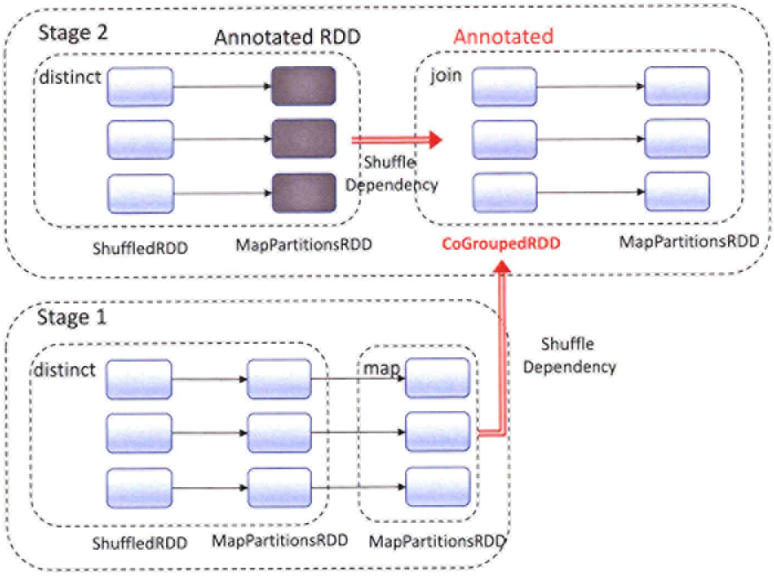
然而，如果 RDD *E* 已经被指定了不同的位置信息，如 NVM 标志。那么其将无视 RDD *F* 传递来的信息，保持其 RDD handler object 仍旧为 NVM 信息。我们称这种 RDD handler object 的标注冲突解决方式为“Erliest First”策略。由于 Spark 的 Stage 计算特征，每一个 Stage 的输入数据都是来自于磁盘或者某个被开发人员显示持久化的 RDD，因此无需对前面 Stage 中的 RDD 进行任何处理。Spark 的这种 Map - Reduce 计算特性简化了 RDD handler object 的标注策略。

5.4.2 CoGroupedRDD 计算的特殊处理

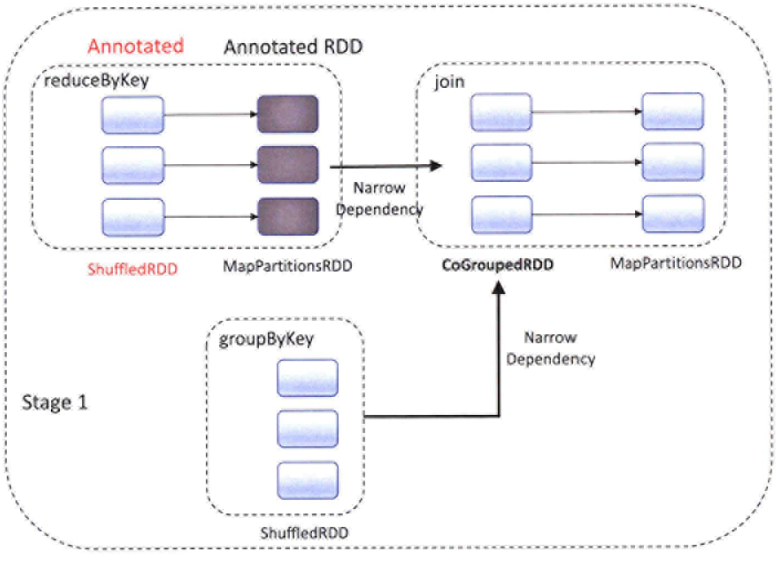
一般情况下，一个 Stage 内部的 RDD 依赖关系均为窄依赖 (Narrow dependency)，并基于此进行 Pipeline 方式的并发计算。然而，当 Stage 内部有 join、CoGroup 等类型的计算时，Stage 内部会有 CoGroupedRDD 类型的 RDD 。其于父 RDD 之间的关系，可能为窄依赖或者宽依赖。如图 5.8 所示的两种情况。

图5.8(a) 展示了，当 Transformation join 导致 Stage 内部有 CoGroupedRDD，同时，其于两个父 RDD 之间为宽依赖 (wide/shuffle dependency) 时的情况。当开发人员利用带位置信息的 Storage Level 进行 DRAM 标注时，如图中灰色 RDD 所示，为了保证 CoGroupedRDD 随后的计算过程都在 DRAM 上进行，“RDD Analyzer”模块会标记对应的 CoGroupedRDD 为 DRAM，不会再去标记 Stage 的起始 RDD。因为该种宽依赖情况下，会导致 CoGroupedRDD 的数据对象被实际计算出来，也就是说，其会被实例化。后续的计算，将会在 CoGroupedRDD 上发生，而并不会再去访问 Stage 头部的 ShuffledRDD。

而对于图 5.8(b) 中所展示的标注情况，灰色 RDD 为开发者标注。虽然 Stage 内部有 CoGroupedRDD 存在，但其和父 RDD 之间都是窄依赖。这种情况下，CoGroupedRDD 并不会被实例化，仍旧是按照标准的 Pipeline 方式，$f_1(f_2(input))$，进行计算。此时，Stage 内计算所以来的数据来自于 Stage 的首个 RDD (ShuffledRDD) 之上，或者某些中间生成的临时数据之上，而新生成的临时数据位于新生代区域，也即 DRAM 之上。因此，只需要将被标注的 RDD 所依赖的 Stage 内部的首个 RDD (ShuffledRD) 标记为 DRAM 便可让后续计算按照开发者的意愿，全部在 DRAM 上进行。



(a) CoGroupedRDD with Shuffle Dependency



(b) CoGroupedRDD with Narrow Dependency

图 5.8　当 stage 内部有 CoGroupedRDD 时的 RDD handler object 标记策略

Figure 5.8　RDD handler object mark policy for a CoGroupedRDD in stage

5.4.3 Spark 层次标注信息传递总结

至此，本节阐述了当开发人员对一个 RDD 变量进行位置标注 (DRAM、NVM) 后，Spark 内部的“RDD Analyzer”模块对相关 RDD 的 handler object 之间传递标注信息的策略。当开发人员对一个 RDD 进行 DRAM 标注后，可以认为其希望将该 RDD 对应的所有数据布局到 DRAM，并希望后续基于该 RDD 的计算亦在 DRAM 上进行。如果开发者将 RDD 标记为 NVM 时，亦是希望其数据被布局到 NVM 区域。然而，由于 Spark 层次和 JVM 层次的数据管理差异性，会因为数据共享而带来布局冲突的问题。本文将在 Spark 、JVM 两个层次，分别针对 RDD handler object ，RDD data object 进行了相关的冲突处理。其中，本节介绍了 Spark 层次的 RDD handler object 的解决策略。

综上，RDD Analyzer 根据用户标注，对 RDD handler object 的标记策略总结如下：

- 当 Spark 应用开发人员标记一个 RDD 变量为 DRAM、NVM 时，其所在 Stage 的 Materialized RDD set（实例化 RDD 集合）中的 RDD 将会被进行以下处理：其所在 Stage 的第一个 RDD，ShuffledRDD 或者从 HDFS 中读取数据构成的 FiledRDD ，其对应的 handler object 将会被标记相同的位置信息，以保证被开发人员标记的 RDD 相关数据可以移动到对应的 DRAM、NVM 区域。
- 如果被用户标记的 RDD 所依赖的某个 Stage 内部的实例化 RDD，已经具有不同的位置信息（DRAM、NVM），那么“RDD Analyzer”将会保持其对应的 handler object 原有的位置标识。我们称此为”Earlier First“标识策略。但在 JVM 层次，并不保证该 RDD 对应的 data object 位置信息不被改变。
- 如果 Stage 内部有 CoGroupedRDD 存在，那么当 CoGroupedRDD 和其父 RDD 之间是宽依赖 (wide/shuffle dependency) 时，由于 CogroupedRDD 也会被实例化，属于 Materialized RDD Set, 因此，其对应的 handler object 会被标记为和用户标注 (DRAM、NVM) 一样，而不再去标注 Stage 的输入 RDD，ShuffledRDD 或 FiledRDD。而如果 CoGroupedRDD 和父 RDD 之间的依赖关系为窄依赖 (Narrow dependency)，那么仍旧按照前述规则进行对应 RDD handler object 的标注。

5.5 JVM 层次的数据划分和移动

当 Spark 内部的“RDD Analyzer”根据 Spark 应用开发人员的位置标注信息，利用 JVM 提供的接口，按照第 5.4 节中所述的策略，对相关的 RDD 的 handler object 进行标注后，JVM 便可以识别到这些标注信息，并展开 JVM 层次的数据划分、迁移工作。本节介绍了 JVM 如何根据 RDD handler object 利用 GC 来对 RDD data object 进行划分和迁移。本节首先阐述了 JVM 层次的架构设计，然后详细分析了，RDD 之间数据共享的成因、带来的问题、以及本文提出的 Spark - JVM 协同数据布局策略。

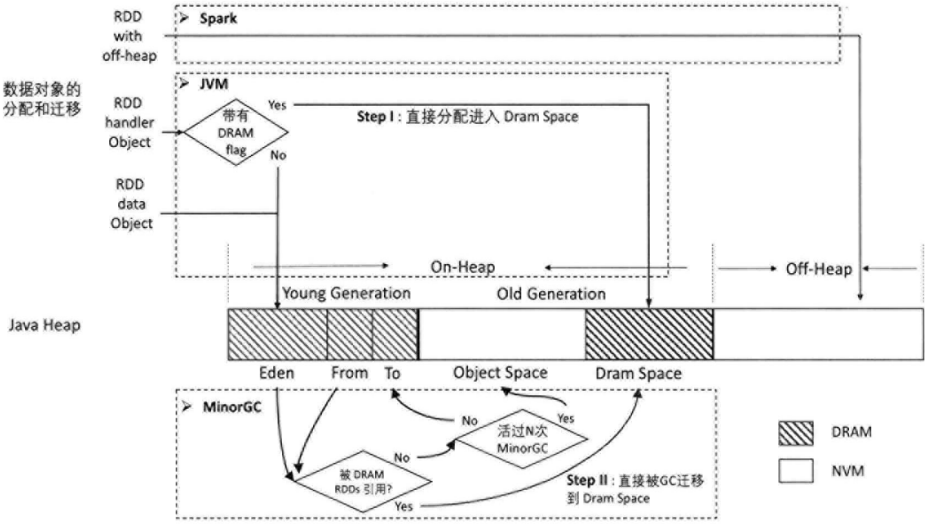


图 5.9 RDD 粒度数据迁移策略

Figure 5.9 之

上的标注信息， Migration-driven RDD allocation

5.5.1 协同编程框架在 JVM 层次的设计

编程框架在 JVM 层次的设计如图 5.9 所示。本节首先介绍了面向异质内存的运行时堆 (Java Heap) 设计，然后介绍了 JVM 如何根据 Spark 的标注信息进行数据的分类和移动。

面向异质内存的运行时堆设计

首先，运行时的整个内存空间被划分为了 On-Heap、Off-Heap 两大部分。其中，Off-Heap 被映射到了 DRAM、NVM 两部分，被开发人员标记为 NVM_OFF_HEAP、DRAM_OFF_HEAP 的 RDD 数据将直接进入相关区域，不

再由运行时堆管理。因为经过第 4 章的讨论，可以看到对 Off-Heap 中数据的读写，都需要经过反序列化、序列化的过程，该过程中处理器计算占据了主要的性能开销，访存带宽、延迟对整体的性能影响有限，因此本文建议将所有置于 Off-Heap 的数据均置于 NVM 区域。

对 On-Heap 部分的内存空间，这里仍旧沿用了第 4 章的研究结论，将整个新生代区域 (Young Generation) 映射到了 DRAM 来处理 Spark 应用中大量的临时数据，而对于旧生代区域 (Old Generation)，本文将其划分为了 DRAM、NVM 两部分，分别被称为 Object Space 和 Dram Space。其中，DRAM 部分用于存储被开发人员标注 DRAM 信息的 RDD，而 NVM 部分作为默认的空间，存储所有的其他数据以及被标注为 NVM 的 RDD 数据。同时，本文仍旧使用表 4.3 中所示的命令行来控制旧生代区域 (Old Generation) 中的 DRAM、NVM 比例。

迁移制导的 RDD 创建策略 (Migration-Driven RDD Allocation)

在 Spark 层次，根据开发人员的的标注，RDD 可以分为 On-Heap、Off-Heap 两大类。其中，Off-Heap 部分的地址空间不再受 JVM 管理。而 On-Heap 部分的 RDD 根据用户的信息标注，又被划分为 DRAM、NVM 两部分，其中无标注的默认形式亦被认为具有 NVM 标志。当 Spark 中的“RDD Analyzer”模块根据这些标注，对 RDD handler object 进行相应的标注后，JVM 便可以在数据创建、GC 等过程中识别出这些标记。

如图 5.9 所示，在进行数据创建时，一旦 JVM 识别到一个被标注为 DRAM 的 RDD handler object，便可感知其为一个生命周期很长的巨大数组。便将其直接分配进入旧生代区域 (Old Generation) 中的 Dram Space，如图中“Step I”所示。否则，其需要经过数次 MinorGC, 并在 From Space 和 To Space 之间数次移动，才能最终被迁移到旧生代区域 (Old Generation)。在随后被触发的 MinorGC 过程中，所有被旧生代区域 (Old Generation) 中 Dram Space 部分引用的数据对象 (Object)，均为一个 RDD 的 data object 部分。其具有和 RDD handler object 相似的长生命周期和访存模式，因此直接将其从新生代区域 (Young Generation) 迁移到旧生代 (Old Generation) 的 Dram Space，如图中“Step II”部分所示。此便为 JVM 根据 Spark 对 RDD handler object 的标注信息，进行数据分类、迁移的总体过程。本文将该过程称为 –“迁移制导的 RDD 创建策略 (Migration-Driven RDD Allocation) ”。

5.5.2 DRAM First 数据共享解决策略

“RDD 引用数据冲突”问题即由于 Spark 和 JVM 之间的数据操作行为、计算行为之间的差距，会造成在 Spark 层次来看不同的 RDD，在 JVM 层次来看却有共享数据对象 (Object) 的现象。对此，第 5.4 节给出了在 Spark 层次的解决方案：对于一个被用户显示标记为 DRAM (NVM) 的 RDD，首先，通过“RDD Analyzer”寻找其所在 Stage 内部依赖的实例化 RDD (Materialized RDD)，然后，将这些 RDD 对应的 Handler object 同样标注为 DRAM(NVM)，以此来保证对应的数据 (RDD data object) 会被 GC 顺利的迁移到 DRAM (NVM)。并提出了“Earlier First”的 Spark 层次的 handler object 标注冲突解决策略。而在 JVM 层次，为了遵循 –“让带有 DRAM 标志的 RDD 的后续计算均在 DRAM 之上进行”的设计思想，我们将优先将同时具有 DRAM、NVM 属性的 data object 优先迁移到 DRAM 区域，以保证程序的性能。本文称此策略为 JVM 层次的“DRAM First”策略。

此外，由于 OpenJDK 8 本身具有的多线程 GC 行为，会进一步加剧处理该问题的复杂性。因此，需要在 Spark、JVM 两个层次的联合策略来解决该问题。下面，本节将以 OpenJDK 8 的 GC 策略为例来解释，如何通过 GC 读取 Spark 的标记信息，以及如何保证将数据正确迁移到对应的 DRAM、NVM 区域。

添加 DRAM To Young Scavenge Task 遍历过程

为了支持实际的内存计算分布式框架 (Spark)，本章采用的运行时系统为 OpenJDK 8[41]，其由 Oracle 开发，在业界被广泛部署。OpenJDK 内部具有多样的性能优化和复杂的策略，轻微的修改也可能带来显著的性能下降，而其说明文档并不完整，为本文工作带来了一定的工程实现挑战。OpenJDK 8 的默认 GC 方式为，分生代的并行扫描模式 (Generation Based Parallel Scavenge GC)。同时，为了提高多线程回收的效率，其采用了”Work steal” 方式的线程负载平衡策略[81][82]，即当各个 GC 线程的工作量不匹配时，优先完成工作的 GC 线程可以去从任务较多的 GC 线程中“偷取”任务，以达到线程间负载平衡的目的。该多线程 GC 的形式给处理信息传递、移动带来了很多挑战。

图 5.10 展示了一个 RDD 所引用的数据对象 (Data object) 同时被其他数据引用，从而造成 RDD 数据共享 (shared RDD data objects) 的情况。红色的点表示 Spark 标注的 RDD handler object，深灰色的点表示其引用的 RDD 数据对象

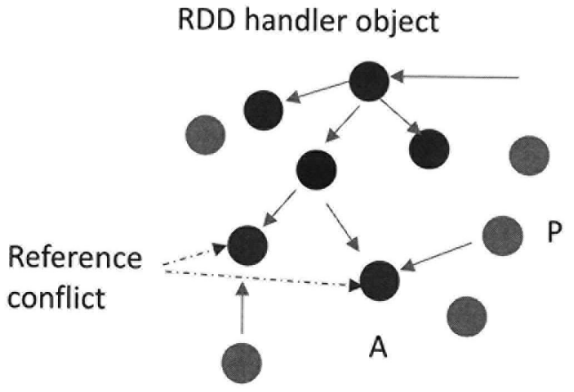


图 5.10　RDD 数据共享

Figure 5.10　shared RDD data objects

(RDD associated data objects)，浅灰色的点表示其他的数据对象，而蓝色点表示该数据对象 (Object) 同时被该 RDD handler object 和其他的数据对象引用。接下来将以此为例阐述 RDD 数据共享带来的问题和解决办法。

本章使用高频的 MinorGC 来进行数据移动，同时修改 MajorGC (Full-GC) 来维护良好的数据布局。OpenJDK 8 中的 MinorGC 由三类任务构成：Old to Young Scavenge Task、Root Scavenge Task、Work Steal Task。一旦 MinorGC 被触发，任务调度中心 (Task Scheduler) 会按照以上顺序分阶段构建三类 GC 任务，然后由空闲的 GC 线程获取相关的任务并执行，每个任务都是由多线程并发完成的。

一个 GC 遍历任务由存活数据扫描 (Trace)、数据迁移 (Object promotion) 两步完成。本节以 Old to Young Scavenge Task 为例来阐述该过程，MinorGC 认为新生代区域中所有被旧生代引用的数据都是存活的，因此，GC 线程需要并发扫描从旧生代区域指向新生代区域中的引用。每当其在新生代区域的 Eden Space 或 From Space 发现一个被旧生代数据引用的数据对象时，便将其认定为存活，此便为 MinorGC 的扫描 (Trace) 过程。当在 MinorGC 发现一个存活的数据后，便如图 5.9 所示检查该数据对象 (Object) 的生命周期，如果其达到了一定要求，该数据对象便会被迁移到旧生代区域，否则将其迁移到 To Space，该过程便为数据迁移 (Object promotion)。一旦该数据对象 (Object) 完成迁移后，其所引用的所有新生代区域中的数据对象亦将被认定为存活，并递归的经历

以上的处理过程。

因此，在 Old To Young Scavenge Task 的执行过程中，只有当 GC 线程从红色的点，RDD handler object，开始，顺着其引用 (Reference) 达到蓝色的点，如 *A* 时，这些共享的 RDD 数据才会获取 RDD handler object 所具有的属性，并被正常的迁移到旧生代区域的 DRAM 区域中。如果是其他的并发 GC 线程抢先一步，通过其他的引用路径，如从 *P* 访问到了数据对象 *A*, 那么 *A* 便会按照正常的流程被迁移到 To Space 或者旧生代区域的 NVM 区域，此时变无法再利用 MinorGC 将其迁移到 DRAM 区域。

然而，我们认为，一旦某个 RDD 被用户标注为 DRAM，其所包含（引用）的数据应该优先按照用户指定的 DRAM 位置进行布局，即使该数据可能被其他具有 NVM 属性的 RDD handler object 引用，亦或被某些临时的数据对象 (Object) 引用。此便为迁移制导的 RDD 创建策略 (Migration-Driven RDD Allocation) 中的“DRAM First”迁移原则。因此，本文在 MinorGC 过程中添加了一个全新的任务：DRAM To Young Scavenge Task，由该任务来将被旧生代区域 (Old Generation) 中 Dram Space 所引用的数据对象快速迁移到其中。比起已经存在的另外三种 MinorGC 任务，该扫描任务具有最高的执行优先级，其会传递 DRAM 标志信息到所有被引用的 RDD data object 中。在此之后，即使该对象是被 Work Steal 机制中的某个空闲进程进行移动的，同样会按照该 DRAM 标志位迁移到对应的旧生代区域 (Old Generation) 中的 DRAM 部分。经过修改过的 MinorGC 策略如算法 2 所示。

Algorithm 2 Modified MinorGC policy

```
1  Key Structures:
2
3  Field : Object field, points to a referenced object.
4  Item  : &field, field address.
5  Queue : GC thread local queue, store the items to be handled.
6  Flag  : DRAM;  NVM.
7  dram_space      : DRAM region of Old Generation.
8  object_space    : NVM region of Old Generation.
9
10 Procedures:
11
12 Triger minorGC & build GC tasks:
13   build DRAM To Young Scavenge tasks;
14   build Old To Young Scavenge tasks;
15   build Roots Scavenge tasks
16   build Work Steal tasks.
```

```
17
18 Schedule a GC task to run:
19   DRAM To Young Scavenge Task(){
20     Scavenge the dirty card  of dram_space
21     Find objects within it
22     Invoke  Push_object_contents(DRAM)
23     Invoke  Handle_items_within_Queue()
24   }
25
26   Old To Young Scavenge Task(){
27     Scavenge the dirty card of object_space
28     Find objects within it
29     Invoke  Push_object_contents()
30     Invoke  Handle_items_within_Queue()
31   }
32
33   Roots Scavenge(){
34     Invoke  Push_object_contents() of the root objects
35     Invoke  Handle_items_within_Queue()
36   }
37
38    Work Steal Task(){
39       if a gcThread is idle then
40         Steal an  Item from other gcThread->Queue
41         Invoke  Handle_items_within_Queue()
42       end if
43   }
44
45
46 Basic functions:
47
48 Push_object_contents( Flag = NVM ){
49   foreach Field in an object do
50     if object in eden_space/from_space then
51       Push  &Field into Queue with Flag
52     end if
53   end for
54 }
55
56 Promote_obj(){
57     if promotion_count < threshold then
58       Copy it to to_space
59     else
60       Copy it to old_gen : object_space
61     end if
62     Set a forwarding pointer in original address
63 }
64
65 Handle_items_within_Queue(){
66   while Queue not empty{
67     Pop a Item
68     if not promoted then
69       if Item->Flag is DRAM then
70         Promote to dram_space
71         Invoke  Push_object_contents(DRAM)
```

```
72        else
73          Invoke  Promote_obj()
74          Invoke  Push_object_contents()
75        end if
76      end if
77    }
78  }
```

经过对 MinorGC 算法的修改，可以保证所有在 Eden Space、From Space 中的共享 RDD 数据，会被优先移动到旧生代的 DRAM 区域。在 MinorGC 的扫描过程中，如果有引用关系，*ObjA Reference ObjB*, 那么 GC 将会传递 *ObjA->Flag* 到 *ObjB*。特殊的，*DRAM Flag* 具有更高的优先级，将会覆盖默认的 *NVM Flag*。但是，如果 *ObjB* 已经按照 *NVM Flag* 被迁移到了 To Space 或者旧生代的 NVM 区，那么 *DRAM Flag* 便无法再对 *ObjB* 造成任何影响。接下来的章节将会对该问题进行详细的阐述。

DRAM First 策略在“RDD Analyzer”信息传递中的体现

对于具有一个 *DRAM Flag* 的 RDD *A* 的共享数据对象 (Shared RDD data objects)，算法 2 只能保证当其在 Eden Space、From Space 时才可被 GC 正确迁移到旧生代的 DRAM 区域。然而，在 RDD *A* 实际创建前，这些数据可能已经被 MinorGC 迁移到了 To Space、旧生代的 NVM 区域中，如图 5.11 所示。DRAM Space 中的红色顶点，表示一个具有 *DRAM Flag* 的 RDD handler object。图中的深灰色、蓝色顶点表示其所引用的 RDD data objects。而 To Space、Object Space 中的蓝色顶点在该 RDD handler object 创建前，已经被 MinorGC 移动到了 To Space、Object Space，而导致无法再被 MinorGC 移动。

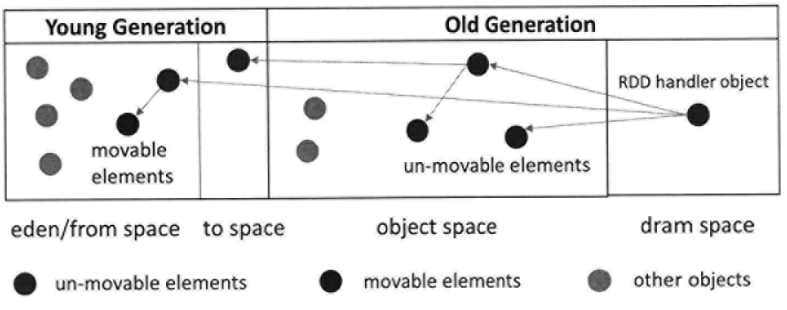


图 5.11 共享的 RDD 数据已被 MinorGC 迁移到 To Space 或旧生代

Figure 5.11 Shared RDD data objects are already moved to To Space or Old Gen

因此，为了保证一个具有 *DRAM Flag* 的 RDD 的数据都能被顺利移动到

DRAM 区域，除了本章节提出的 MinorGC 算法，仍旧需要借助于第 5.4 节中所描述的，“RDD Analyzer”模块对相关 RDD 的分析，以及对 handler object 的标注策略。即将被用户标注的 RDD 所依赖的，并在 Stage 内部的实例化 RDD 亦进行相同的信息标注。此时虽有一定概率将已经死亡的数据迁移到旧生代的 DRAM 区域 (Dram Space)，造成一定的 DRAM 空间浪费。但是，任何一次 MajorGC (Full-GC) 便可以将这些数据清除，释放旧生代的 DRAM 空间。

然而，经过对一些 Spark 应用的分析发现，多数被开发人员显示持久化 RDD 为一个 Stage 的起始 RDD，即 ShuffledRDD，该种标注策略会大大减少无用数据迁移到 Dram Space 的比例。这是因为，Shuffle 计算为 Spark 计算中耗时最多，对性能影响最大的计算。因此，如果一个 RDD 为 Shuffle 计算的结果，并且会反复参与多次迭代中的运算，那么将其持久化后，就会显著的节省重复计算的开销。此外，每个 Stage 内部的 ShuffledRDD 在计算过程中本身就会被 Spark 实例化，将其显示的持久化 (persist) 仅仅是将实例化的输出转存到 Spark Storage Memory 中。

如图 5.12 所示，为 Spark 应用开发人员持久化并标注一个 ShuffledRDD 为 DRAM 后的行为。在一个 ShuffledRDD 构建过程中，其首先会创建其数据，然后才会将结果储存到 Spark Storage Memory 中。在该过程中，“RDD Analyzer”需要将一开始的临时数组 (Temporary object array) 作为 handler object 进行标注，否则，会造成图 5.11 所示的情况。此时，ShuffledRDD 的数据 (Data objects) 直接来自于“RDD Analyzer”标注的 Temporary object array，几乎没有多余的数据被迁移到旧生代的 DRAM 区 (Dram Space)。

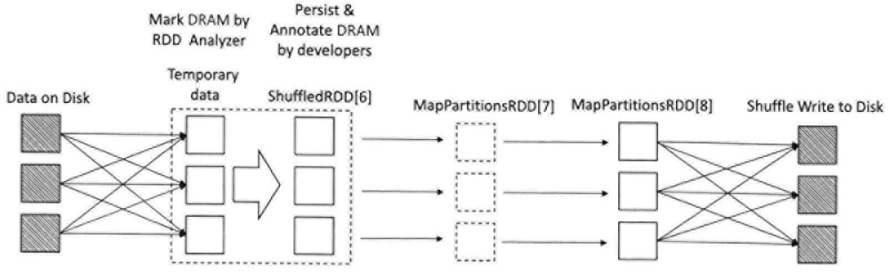


图 5.12　应用开发人员标记 ShuffledRDD 为 DRAM

Figure 5.12　Developers annotate a ShuffledRDD with DRAM Flag

而当 Spark 应用开发人员持久化、并标注 Stage 内部的一些临时 RDD 时，

确实有一定可能造成多余的数据被迁移到旧生代的 DRAM 区。我们将在随后的工作中，努力减少被迁移的无用数据。在第 5.7 节进行的实验通过良好的性能收益，验证了本文提出的 Spark - JVM 协同框架的高效性。

5.6 利用信息传递机制进行定向优化

当 Spark 应用的开发人员显式的对持久化的 RDD 做标注信息 (DRAM、NVM) 时，本文提出的框架可以在 Spark 和 JVM 层直接获取对应数据的冷、热信息，以及数据的生命周期信息。针对这些获取的信息，我们重新设计了 RDD handler object 的创建过程，以及对应 RDD 创建的 MinorGC 过程，如图 5.9 中的“Step I”、“Step II”所示。此外，在获取到这些从上层传递下来的信息后，本文在 Spark、JVM 层次分别做了一针对应用的定向优化。

5.6.1 Spark 层次的优化

Scala 语言是一种基于托管式运行时开发的函数式语言，其本身具有很好的易用性，包括无需声明对象类型、将函数作为参数等特性。Spark 在使用 Scala 开发时，关注点为良好的可移植性、计算的高并发性，并未过多关注代码的效率。如在构建一个 RDD 时，其直接使用了 *class PrimitiveVector* 这个自定义类来管理 RDD 的实际数据，其将数组存储到一个 *Array*[*V*] 中，此便为 JVM 可见的 RDD handler object 。然而，这是一个固定长度的数组，Spark 在构建 RDD 的过程中，并不知道每个 RDD partition 的实际长度有多少。因此其默认构建了一个长度为 64 的 *Array*[*V*] ，并向其中添加数据，一旦其被数据充满，便会重新构建一个为原长度两倍的新数组，并将旧的数据拷贝到其中，整个过程如图 5.13 所示。

对于 Spark PageRank 这个应用来说，当输入集只有 960MB，RDD 被分为 16 个 Partition 时，每个 RDD Partition 的长度甚至可以达到上百万，即 2^{20} 以上。此时，一个 RDD 的构建过程中，会导致数十次的 RDD handler object 创建，以及数据拷贝。因为，异质内存的空间很大，因此此处采用了“空间换时间”的策略。对于一个具有 *DRAM Flag* 或者显示被赋予了 *NVM Flag* 的 Array 构建过程，将其初始值设定为 64K，也即 2^{16}。此时虽然可能造成一定的空间浪费（几十 MB），但是却可明显的减少计算。在后面的实验中，我们使用 Off-Line Profiling 的形式，根据数据集的输入，将每个 RDD handler object 的初始长度

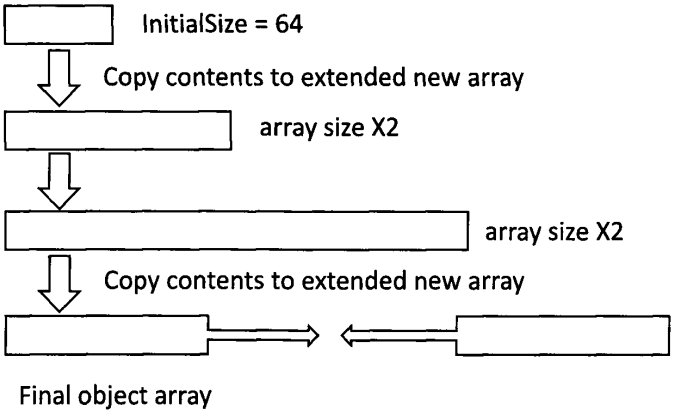


图 5.13 RDD handler arrary 的构建过程

Figure 5.13 The expansion of handler array during RDD building process

调到了最佳值。

5.6.2 JVM 层次的优化

Spark K-Means、Spark Logistic Regression based Classification 是两个离线的聚类、分类应用。其数据使用行为类似，Java Heap 内均具有一个占支配地位的超大 RDD，而且计算过程中，Java Heap 中的数据会趋于稳定，总容量由新生代大小和位于旧生代的该超大 RDD 构成。这两个应用均为迭代计算，其主要计算亦是发生在该 RDD 上。其触发的 GC 主要由 MinorGC 构成，而且频率很高，旧生代区域中的数据增长十分缓慢。在该种计算模式下，获取 RDD Analyzer 传递下来的信息后，有机会提高 MinorGC 的效率。

为了理解本节提出的优化，需要首先介绍为了加速 MinorGC，OpenJDK 8 所使用的 Card Table 机制和 Write Barrier 机制。该机制被广泛的应用于 Generation GC 设计中。

Card Table 机制

MinorGC 只需要处理新生代 (Young Generation) 中的数据，无需关心旧生代 (Old Generation) 中的数据。为了判定新生代中哪些数据是存活的，其需要从一些变量作为扫描存活数据的起始点 (Root Set)。正如在 5.5.2节中所介绍的 Scavenge Task，其中 MinorGC 的两个起始点为：堆栈中的变量、旧生代中的数据。这两个 Root Set 中的数据均被认为是存活的数据，因此，其所引用的数据对象 (Object) 自然也被认为是存活的。针对不同的 Root Set，OpenJDk 8 分别设计了 Root Scavenge Task 和 Old To Young Scavenge Task 两种并发遍历任务。

由于旧生代 (Old Generation) 中的数据对象 (Object) 数量众多，OpenJDK 设计了 Card Table 机制来快速识别出哪些旧生代中的数据对象会引用新生代中的数据对象，该机制便为 Card Table 机制。首先，OpenJDK 将整个 Java Heap 划分为固定大小的 Card，512 Bytes 。然后，其规定，只有当一个 Card 的状态为 Dirty 时，该 Card 中才可能有指向新生代的数据对象。因此，MinorGC 过程中，不同的 GC 线程只需要遍历旧生代中的 Dirty Cards, 并从中找出确实有指向新生代的数据对象 (Object)，然后将其作为起始点开始进行 GC 扫描。这便直接过滤掉了旧生代中大量的数据对象。

Write Barrier 机制

对于 Card Table 机制，需要在 Spark 应用运行和 GC 过程中两个阶段维护 card 的值。其中，在 Spark 应用运行过程中，需要动态监测每一次对数据对象 (Object) 的 Reference 赋值操作，如 $ObjA.field = ObjB$，此时 $ObjA$ 所在的 Card 便会被设置为 Dirty。维护该过程的机制，便是 Write Barrier 。同时，为了减少 Write Barrier 的监测开销，这里并没有进一步区分 $ObjB$ 是否在新生代，该识别过程留给了 GC 迁移过程进行。而且该监测过程的代码是由 JIT 直接以汇编形式生成的，具有很高的效率。而在 GC 执行过程中，当从通过某 Dirty Card 发现了从旧生代指向新生代的引用后，并将对应的数据迁移到了旧生代时，该 Card 对应的 Dirty 值将会被清理。

Card Table 机制存在的问题以及优化机遇

Card Table 机制存在一些效率问题，本节在此介绍一个在 Spark 应用中比较突出的性能问题，并利用 Spark - JVM 消息传递加以解决。

为了加速旧生代中 Dirty Card 的扫描效率，OpenJDK 8 亦采用了多线程来对其进行分割的并发扫描。其将整个旧生代切分为许多块 (Slice)，每次由多个线程扫描 Slice 中的一部分 (Stripe)。有时一些巨大的数据对象会横跨多个 Card, 并结束在某个 Card 的内部，如图 5.14 所示。

这样，在一次 MinorGC 的 Old To Young Scavenge 的过程中，处理前面 Slice 时，无法清除红色的 Dirty Card 为 Clean 状态，因为其无法判断该 Card 后半部分的情况；同样，当处理下一个 Slice 时，其同样无法清除该红色的 Dirty Card，因为其无法判断在处理前一个 Slice 的过程中，红色 Card 的前半部分中是否还有指向新生代的引用。此时，当下一次 MinorGC 发生时，该红色的 Dirty Card

会导致在其上的大数组被再次扫描。虽然，重复处理不一定导致数据移动，但是对于长度在数百万的数组和高频的 MinorGC 来说，这里仍旧会带来的显著的额外开销。该种情况只有留待 MajorGC 来来处理，当 MajorGC 将整个新生代的数据都迁移到旧生代后，那么所有的 Dirty Card 都可以被安全的清除。

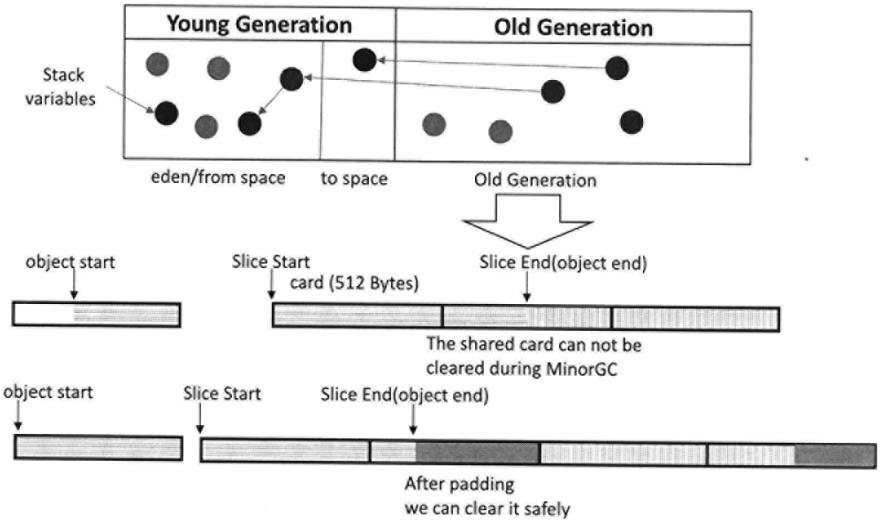


图 5.14　一些 Dirty Card 无法在 MinorGC 中被重置

Figure 5.14　Shared dirty cards can not be cleaned during MinorGC

Spark 应用的计算单元为数量众多的 RDD，每一个 RDD Partition 在构建过程中，都会产生大量的巨大数组，同时，对于 Spark K-Means、Spark Logistic Regression based Classification 这类以 MinorGC 主导，MajorGC 很少的应用会进一步加重图 5.14 所展示的情况。

然而，当 JVM 感知到 RDD Analyzer 对 RDD handler 的标注信息后，便带来了减少这种开销的机会。如图 5.9 所示，当 JVM 识别到一个 RDD Handler 的分配时，可以控制其使用的内存空间大小和位置。因此，本文通过 padding 操作，来确保其在旧生代的 DRAM Space 中占据数个完整的 Card，不再和其他的大型数组共享 Card。此外，本文也修改了 MajorGC 来保持 DRAM Space 中的这种数据布局。在这种情况下，MinorGC 便可以顺利清除这些带有特殊信息的大数组所占局的 Dirty Card，从而避免连续触发的 MinorGC 对这类数组的重复扫描。

这里使用 Spark K-Means 来验证该优化效果，其会显著的减少 MinorGC 的开销。首先，为了验证该优化对 MinorGC 开销的影响，该处选择了两种配置

来进行对比：所有数据全部运行在 DRAM 作为性能最优的基准 (baseline)；将新生代映射到 DRAM，然而将整个旧生代 (DRAM、NVM 两块区域) 都固定在了 NVM 之上。该配置是为了展示，在异质内存中使用恰当的布局 (新生代在 DRAM，旧生代在 NVM)，并辅以本章节提到的定向后所能达到的最优性能。此时，即使开发者将 RDD 标注为 DRAM 后，其虽然会被置于旧生代的 DRAM Space，但其仍旧是运行时 NVM 之上，只不过应用了本节所提到的优化策略。

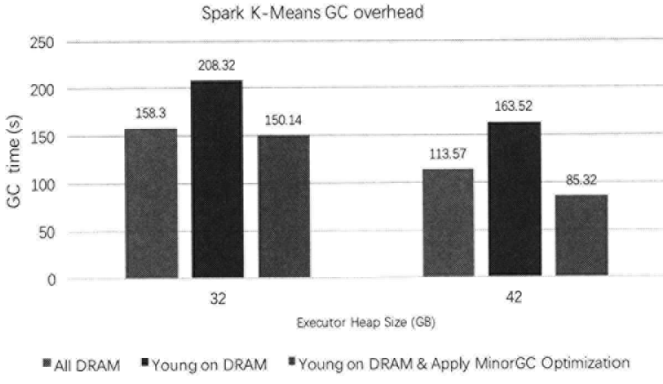


图 5.15　利用 Spark 传递的信息定向优化 MinorGC 开销

Figure 5.15　Optimize MinorGC overhead via the message delivered by Spark

实验结果如图 5.15 所示，灰色柱子表示了程序全部运行在 DRAM 上的 GC 性能。而蓝色柱子表示只有新生代在 DRAM 之上，整个旧生代在 NVM 上时的 GC 性能。橘色柱子表示了，在蓝色配置之上，应用本节的优化之后的效果。而横轴表示了两种不同的 Java Heap 大小。通过对比配置相同的蓝色柱子和橘色柱子可以看到，应用优化之后，GC 的开销分别减少了 38.7% 和 91.7%。此时的 GC 开销甚至比使用等量 DRAM 的开销还要小。

因为 GC 的计算行为类似于图遍历，是一个以 Pointer Chasing 为主的访存行为，根据文章[10] 的结论，该种类型的访存行为对内存延迟的性能很敏感。因此，NVM 之上的 GC 性能会显著弱于 DRAM 之上。然而，经过使用本节提出的优化，可以看到，橘黄色柱子几乎和灰色柱子具有一样、甚至更低的时间。程序的运行时间由 GC 时间和应用运行时间两部分构成，经过本节对 GC 开销的优化后，仅仅将新生代 (25% of total heap) 映射到 DRAM 的 Spark K-Means 和全部使用 DRAM 具有了几乎相同的性能，如图 5.16 所示。

该节提出的 Card Table 优化仅仅实现在了旧生代中的 DRAM Space 中，为

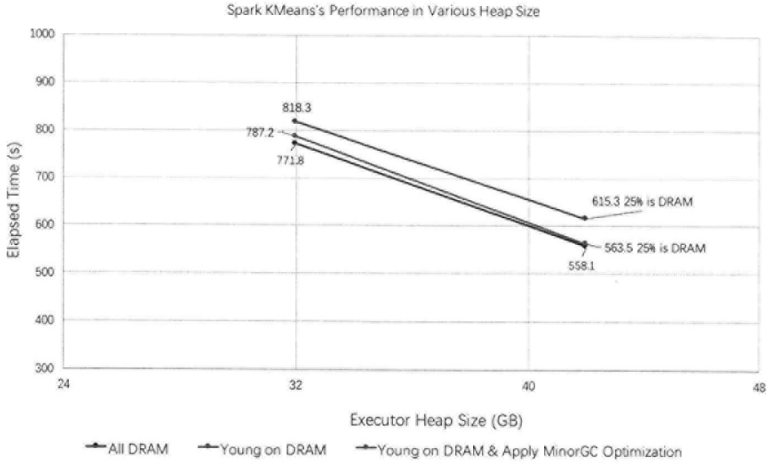


图 5.16　采用 GC 优化后的 Spark K-Means 性能

Figure 5.16　The Elapsed time of Spark K-Means after applying the GC optimization

了使用本节提出的 Card Table 优化，需要对 RDD 进行 *DRAMFlag* 标注，使其全部处于旧生代的 DRAM Space。而在随后的实验中，评价的主要标准是异质内存中的数据布局带来的收益，因此，实验中整个异质内存仅仅有 1/4，1/3 比例的 DRAM 给新生代和旧生代 DRAM Space 使用。旧生代的 DRAM Space 并无空间容纳众多 RDD，所以几乎没有应用 Card Table 优化。

5.7　性能测试与分析

5.7.1　测试平台

该处实验所使用的模拟平台和第 4章一致，也是基于同样的服务器和方法模拟出的异质内存平台，其参数见表 4.6 。此外，服务器的配置亦同表 4.7 所示。

5.7.2　测试用例分析

本节将使用 Spark 中自带的测试用例来展示 Spark - JVM 联合编程框架的效果。这里根据测试用例的计算行为，将其划分为了三类，如表 5.2 所示。其中，第一类 Spark PageRank 的计算流程如图 1.3 所示。其为一个迭代计算，参与计算的 RDD 数量较多，且可以被清晰的划分：参与每一次迭代计算的 RDD links；在内存空间允许的情况下，可以通过持久化来进行容错的 RDD contribs; 以及其他大量临时生成的 RDD。对于该种类型的应用，开发人员可以很好的识别各种 RDD 的冷、热行为，并给予恰当的信息标注。

第二类应用由 Spark K-Means, Spark LR 构成。正如 5.6.2 中的分析，其数据主要由一个占据支配地位的 RDD 和新生代中的临时变量构成。其每次迭代的计算主要是遍历、处理该主要 RDD 上的数据。新生成的数据几乎全部在新生代被回收，旧生代增长十分缓慢。对于该种类型的应用，Java Heap 中并没有足够的 DRAM 空间来容纳涵盖了大部分计算的“支配 RDD (Dominant RDD)”。但是，仍建议用户将其标注 DRAM，从而将一部分计算分流到 DRAM 上。

第三类应用以 Spark TC 代表。该应用的目的是计算一个图中任意两个点之间的连通性，并将其结果存储到一个 RDD 中，因此，其 Java Heap 内会有一个极速增长的 RDD 数据。例如，对于一个 15MB 的输入集，构建的第一个 RDD 大小只有 172 MB。而在第二次迭代计算的过程中，其大小增长了近十倍，达到了 1.9GB。在第三次迭代后，该 RDD 的大小达到了 20GB。为了防止 Out of Memory Error, 我们根据 Java Heap Size 限制了程序的迭代次数。

表 5.2 测试用例及其数据使用特点

Table 5.2 The benchmarks and their data usage characters

Benchmark	Description	Characters
Spark PageRank	Websites ranking algorithm	A small and Hot RDD with lots of temporary RDDs
Spark K-Means	K-means clustering	Access a single dominant RDD
Spark LR	Logistic regression based classification	
Spark TC	Transitive closure on a graph	A rapidly expanding dominant RDD

5.7.3 Spark PageRank 的性能分析

首先对 Spark PageRank 进行性能分析，如图 5.17 展示了其在 DRAM 和异质内存上的性能。横轴表示 Java Heap 也即 Spark Executor 的内存大小，分别为 64GB 和 120GB。纵轴表示程序执行时间比上整个 Java Heap 在 DRAM 上的时间。此外，对于异质内存中的比例，选择了 1/3、1/4 两种配置来进行实验展示。

应用框架后的性能展示

蓝色柱子表示程序运行在 DRAM 上的时间，将其作为基准，正则化后为 1。黄色柱子和绿色柱子展示了在异质内存有 1/4 为 DRAM 时，没有任何管理、使用本文提出的编程框架后的性能对比。从中可以看到，在 64GB 和 120GB 两

种配置下，在无管理时，异质内存的上的性能比全部使用 DRAM 下降了 34% 和 42% 。而使用本节提出的框架后，性能提升了 9.8% 和 14.5% , 和全部使用 DRAM 仅有 22% 和 24% 的性能差。此时，全部 25% 的空间都被新生代占据了。

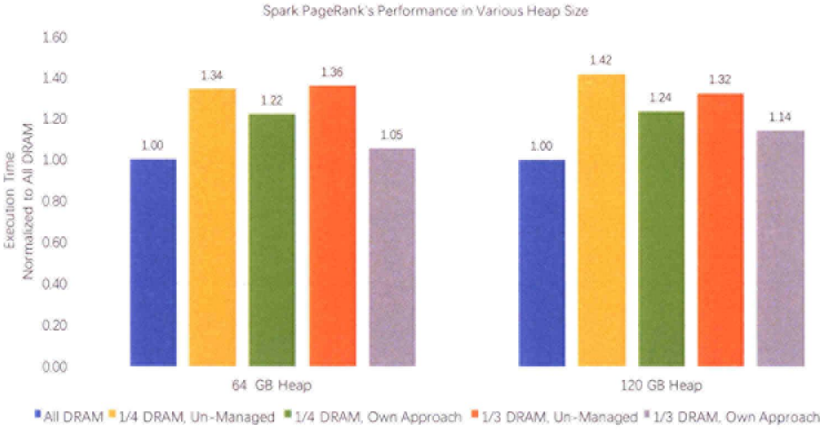


图 5.17　Spark PageRank 应用框架后的性能分析

Figure 5.17　Spark PageRank's performance after applying own framework

当 DRAM 比例达到 1/3 时，旧生代中的 DRAM Space 也会分到一部分 DRAM 空间，此时用户标注的，带有 *DRAM Flag* 的 RDD 可以被置于其中。对于 Spark PageRank, 为了充分利用所有内存空间，该处实验并没有将 RDD *contribs* 进行持久化，此亦为 Spark 所给例子的默认配置。虽然此时的容错性有一定的降低，但内存的使用效率会大幅升高。同时，RDD contribs 进行持久的配置，在第 4章中的策略下，已经取得了良好的性能。在本节的实验中，性能对比如橘黄色柱子和灰色柱子所示，在 64GB 、120GB 两种配置下，无管理时的性能比使用 DRAM 下降了 36% 和 32% 。然而，在应用本文框架以 RDD 进行布局后，性能显著提升，和全部使用 DRAM 相比，仅分别有 5% 和 14% 的性能差。

性能优化分析

本文从带宽角度来对应用框架后的性能提升进行分析。如图 5.18 为 1/3 DRAM 比例下，Spark PageRank 在三种配置下的平均访存带宽。此时，Java Heap 和 Spark Executor 的大小被固定在了 64 GB 。蓝色柱子表示了 DRAM 端的总带宽，橘色柱子表示了 NVM 端的总带宽，包含了读带宽和写带宽。可以

看到，在全部使用 DRAM 的情况下，即使是平均带宽也达到了将近 11GB/s，而读带宽的峰值，达到了 25GB/s，如图 5.19(a) 所示。而在 1/3 DRAM 的异质内存中，无管理模式下，NVM 端的平均访存带宽是 DRAM 端的 2 倍以上，其瞬时带宽也常常逼近 12.8 GB/s 的限制，如图 5.19(b) 所示，因此，造成了性能的显著下降。

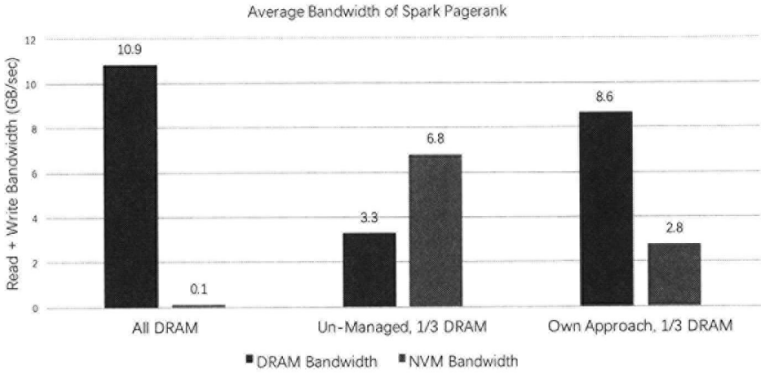


图 5.18 Spark PageRank 的平均带宽分析

Figure 5.18 Spark PageRank's average total bandwidth

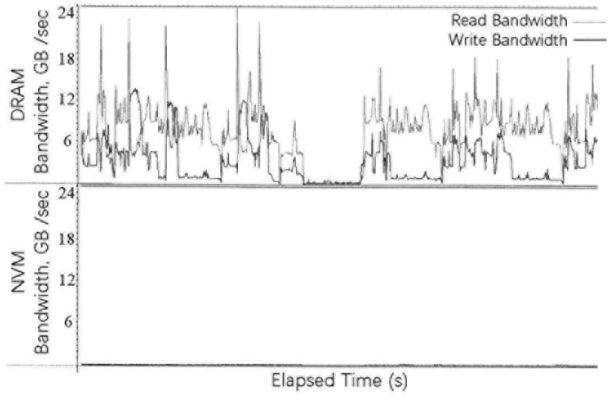
而在应用本文提出的框架后，按照编程人员的指定将带有 *DRAM Flag* 的数据布局到 DRAM，将其他访问不频繁的 RDD 布局到 NVM 后，重新平衡了 DRAM、NVM 两侧的访存量。其平均带宽如图 5.18所示，瞬时峰值带宽如图 5.19(c) 所示。从中可以看到，经过合理布局后，大部分的读访存请求命中在了 DRAM 一侧，其峰值可以达到 16GB/s 以上。同时，DRAM 端的写访存请求，也明显多于 NVM 端。

通过这里的性能分析和带宽分析，可以看到，对于 Spark PageRank 这一类迭代内存计算应用来说，通过开发人员以 RDD 粒度进行数据布局，配合生代级别 (Generation) 的粗粒度数据布局，确实可以在异质内存上得到较好的性能。和全部使用 DRAM 相比，使用 DRAM 比例仅有 1/3 的异质内存时，通过合理的数据布局便可以将性能差减少到 10% 左右。

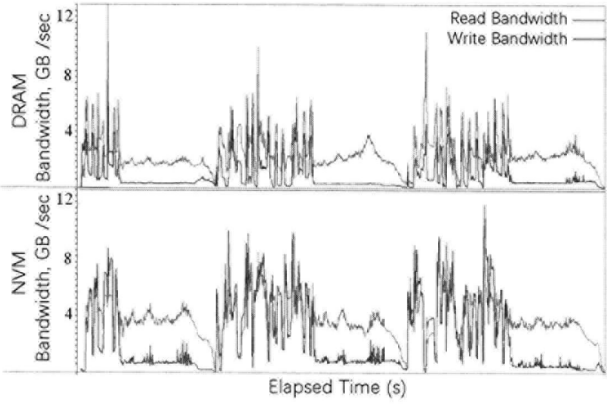
5.7.4 Spark K-Means 和 Spark LR 的性能分析

应用框架后的性能展示

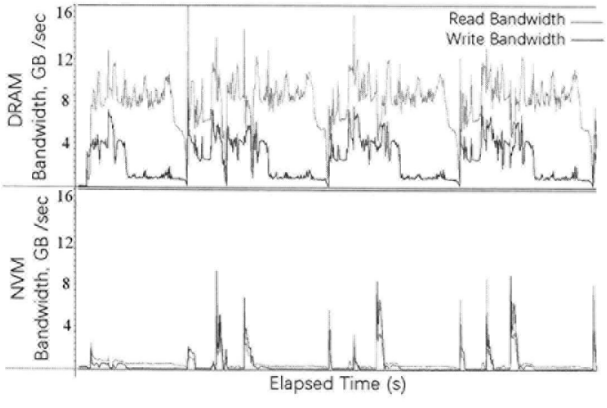
本节对第二类 Spark 应用，Spark K-Means 和 Spark LR 在应用本节框架后的性能进行了分析，第 5.7.2节对该类应用的数据使用特点、计算特点进行了



(a) All DRAM



(b) Un-Managed, 1/3 DRAM



(c) Own framework, 1/3 DRAM

图 5.19　Spark PageRank 的访存带宽

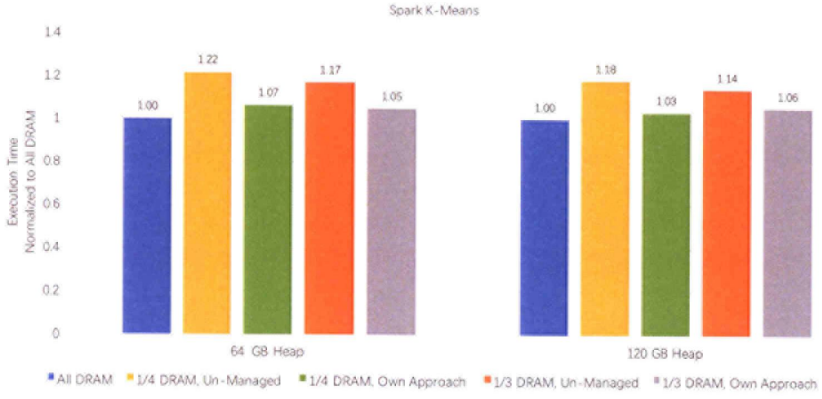
Figure 5.19　Spark PageRank's memory access bandwidth

介绍。在本节的分析过程中，将处于支配地位的 RDD 简称为“支配 RDD”。图 5.20 展示了二者在异质内存中的性能，虽然二者在 Spark 引用源码层次展现出了相似的迭代计算特征，并且主要计算数据均为一个占支配地位的 RDD，GC 行为也类似，但由于每次迭代内的计算量不同，导致了二者访存细节的差异性。相比于 LR，K-Means 具有更加密集的计算；而 LR 具有更高的访存峰值。

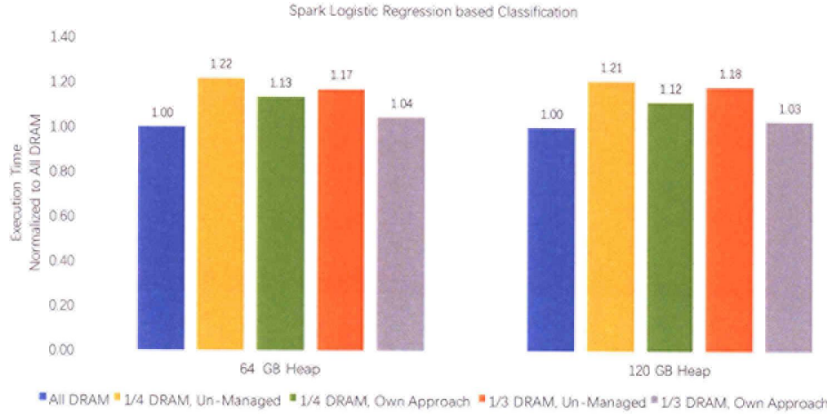
图 5.20(a) 展示了 Spark K-Means 在应用异质内存框架后的性能，同样的，黄色和绿色柱子表示了当 DRAM 比例只有 1/4 时的异质内存平台，而橘色和灰色柱子展示了整个异质内存平台中有 1/3 内存时的运行情况。首先，通过黄色柱子和绿色柱子对比可以看到，经过应用本节提出的编程框架后，在 64 GB 和 120 GB 两种配置下，整体性能分别显著提升了 14% 和 14.5%，此时和全部运行于 DRAM 只有 7% 到 3% 的性能差距。此时，异质内存中的 DRAM 比例仅有 1/4，只够将新生代映射到 DRAM，旧生代的 DRAM Space 没有空间来容纳用户标注 *DRAM Flag* 的 RDD。

橘色柱子和灰色柱子展示了当整机有 1/3 DRAM 时的情况，可以看到，虽然 Un-Managed 配置下，性能小幅提升，而在应用我们的框架后，对比绿色柱子和灰色柱子，Spark K-Means 性能几乎没有变化。此时旧生代 DRAM Space 的空间，不足以容纳整个“支配 RDD”。

图 5.20(b) 展示了 Spark LR 的性能情况。虽然最终，在整机有 1/3 DRAM 时，应用本文的框架后，Spark LR 的性能也达到了和 All DRAM 仅有不到 5% 的性能差距。但是，在整机平台仅有 1/4 DRAM 时，应用本文的框架后，和无管理状态 (Un-Managed) 相比，性能小幅提升 8% 左右。Spark LR 的性能提升和 Spark K-Means 展现出了完全不同的特点，这是由于二者的计算差异导致的。下文将结合带宽对此问题进行阐述。



(a) Spark K-Means



(b) Spark LR

图 5.20　Spark K-Means 和 LR 应用框架后的性能

Figure 5.20　Spark K-Means and LR's performance after applying own framework

性能优化分析

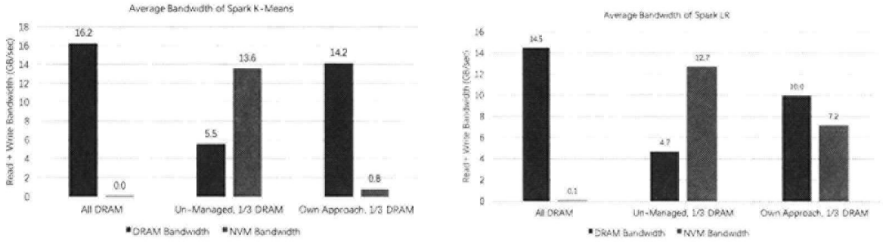
图 5.21 展示了异质内存平台具有 1/3 DRAM 时，应用本文框架进行数据移动后的性能平均带宽展示。其中依旧同时包含了读、写两部分带宽。

通过对比图 5.21(a) 和图 5.21(b) 中的第三部分，也即应用框架进行数据迁移后的平均带宽可以看到，在完成数据布局后，Spark K-Means 中大部分的访存行为均被迁移到了 DRAM 端。而对于 Spark LR 来说，其 NVM 端仍旧保留了显著的访存行为，但是完全没有达到 NVM 的带宽瓶颈。

经过分析，造成这样情况的原因在于，在每次迭代中，获取一个数据 (点) 后 Spark K-Means 需要进行的计算显著的多于 Spark LR。Spark K-Means 需要

将这个获取的数据和 N 个中心点进行浮点计算，来判断其属于哪个范围。该过程的计算量较大，因此导致对“支配 RDD”的访存带宽并不大。而其计算过程中，仍旧会产生大量的临时数据 (RDD)，但是这些数据均被新生代回收了。因此，当将新生代映射到 DRAM 后，Spark K-Means 的主要访存也被迁移到 DRAM 端。而对于 Spark LR 来说，其 NVM 端仍旧保留了大量的访存。

对于 Spark K-Means 来说，当将新生代映射到 DRAM 后，和无管理状态 (Un-Managed) 相比，其性能便会显著提升，见图 5.20(a) 中黄色、绿色柱子对比。然而，当将用户标注的 RDD 部分迁移到 DRAM 后，几乎并没有迁移多少访存到 DRAM，所以性能几乎没有变动，如图 5.20(a) 绿色、灰色柱子对比。而 Spark LR 则展现了不同的行为，其“支配 RDD”之上有较大的访存带宽。当将新生代迁移到 DRAM 后，和无管理状态 (Un-Managed) 相比, 其性能提升有限，如图 5.20(b) 黄色、绿色柱子对比。而当，将部分“支配 RDD”也迁移到 DRAM 后，一部分的访存也被分流到了 DRAM，进一步减少了 NVM 端的访存压力，此时 NVM 端的带宽不再是限制 Spark LR 整体性能的瓶颈。因此，其整体性能会进一步提升，如图 5.20(b) 中绿色、灰色柱子所示。



(a) Spark K-Means **(b) Spark LR**

图 5.21 Spark K-Means 和 LR 的平均带宽

Figure 5.21 Spark K-Means and LR's average total bandwidth

该节仍旧抓取了 Spark K-Means 和 Spark LR 的一段访存带宽来展示数据迁移的影响。图 5.22 和图 5.23 展示了在 64GB Java Heap (Spark Executor Memory) 配置下，Spark K-Means 和 Spark LR 的瞬时访存带宽特征。通过对比二者在 DRAM 之上运行时的带宽，如图 5.22(a) 和图 5.23(a) 所示，可以看到，Spark LR 拥有比 Spark K-Means 更高的瞬时带宽，Spark LR 的瞬时读带宽可以达到 20GB/s ，而 Spark K-Means 均在 16GB/s 以下。这和我们前述的分析，每次迭代中 Spark K-Means 进行更多的计算相符。

此外，在使用本文的框架合理部署数据后，Spark LR 仍旧有大量的内存读、写命中在了 NVM 端，如图 5.23(c) 所示。其内存读带宽远远小于 12.8GB/s 的瓶颈，而其内存写带宽却逼近了 12.8GB/s 的瓶颈。由于在本文中使用 NUMA 的远端 DRAM 来对 NVM 进行仿真，而对于 DRAM 来说，内存写并不会严重的影响程序性能，除非读、写之间有严重的依赖而导致内存读无法满足处理器的计算。虽然目前一般认为 NVM，尤其是 PCM (Phase Change Memory) 具有读写不对称性[6]，写性能较差，但是本文仍旧认为其可以达到 12.8GB/s 的写带宽，此值仅为目前服务器 DRAM 服务器的 1/8 到 1/4 的左右，符合主流研究的认知[10][6]，如频率为 2400MHz，4 通道的 DRAM 服务器，其内存理论带宽可以达到 76.8GB/s 。

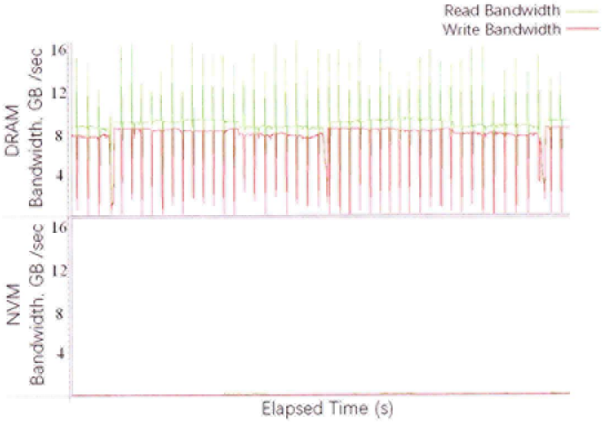
此外，通过使用带宽分析辅以运行时间分析，可以确定，以 RDD 为粒度的数据布局，确实可以通过利用异质内存中的少量 DRAM 来容纳大量的访存行为。同时，将大量的访存不频繁的数据留在了 NVM 中。即使针对 Spark K-Means, Spark LR 这类无法将整个 RDD 均迁移到 DRAM 的情况，仍旧可以通过分散热数据，来减少 NVM 端的压力，提升整体性能。

5.7.5 Spark Transitive Closure 的性能分析

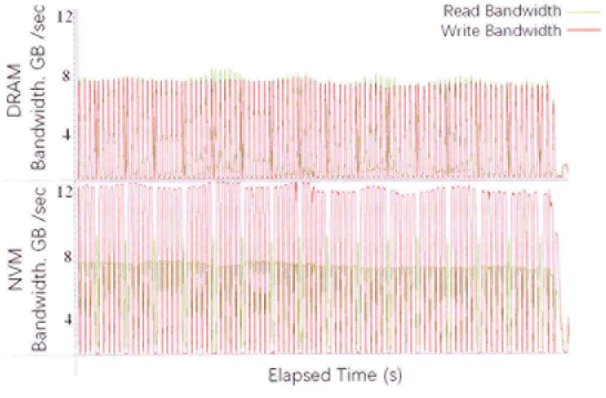
应用框架后的性能展示

Spark TC (Transitive Closure) 是一个图遍历应用，其目的是找出一个图中所有的有向边，并将其储存于一个 RDD 中。其与 Spark K-Means 和 LR 的数据使用区别在于，虽然 Spark TC 中也有一个占据支配地位的 RDD，然而，其大小却不是恒定的，而是在前期以指数级别增长，随后趋于稳定。正如 5.7.2 节中的描述，在一个很小的输入集下，在初始的迭代计算过程中，该“支配 RDD”会以十倍左右的速度快速增长，直至导致 Out Of Memory 错误。因此，本节的实验限制了应用的迭代次数，以便 Java Heap 在 64GB 和 120GB 下不会崩溃。

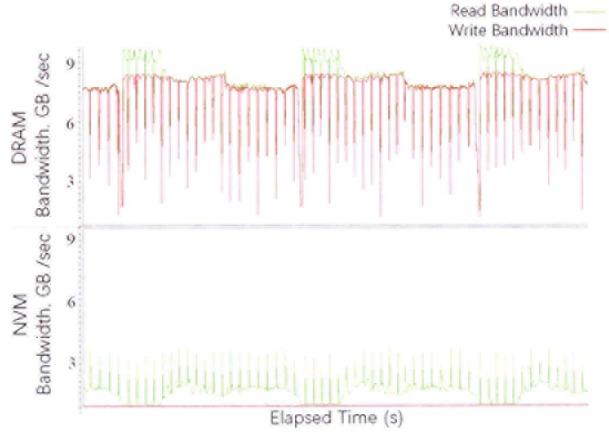
图 5.24 展示了 Spark TC 在 64GB、120GB 两种 Java Heap(Spark Executor Memory) 配置下的运行时间。在异质内存中的 DRAM 仅有 1/4 时，仍旧沿用第 4 章的结论，将新生代映射到 DRAM。性能提升如图 5.24 中黄色、绿色柱子所示，在两种 Java Heap 配置下分别为 5.2% 和 16.7% 。在不同的 Java Heap 大小中，性能的加速差异十分明显。经过分析，认为其原因在于在固定的输入集，固定的迭代次数下，120 GB Java Heap 的配置时，有大量的热数据没有替换到



(a) All DRAM



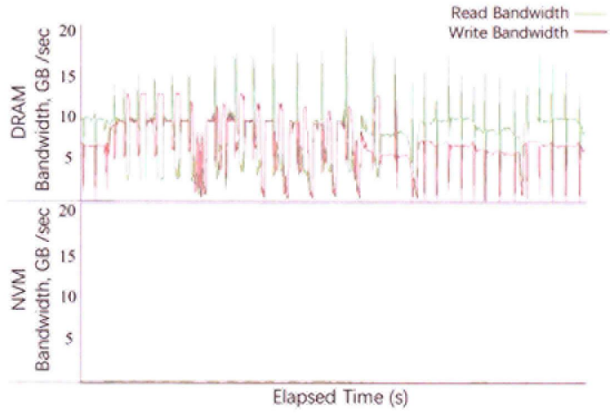
(b) Un-Managed, 1/3 DRAM



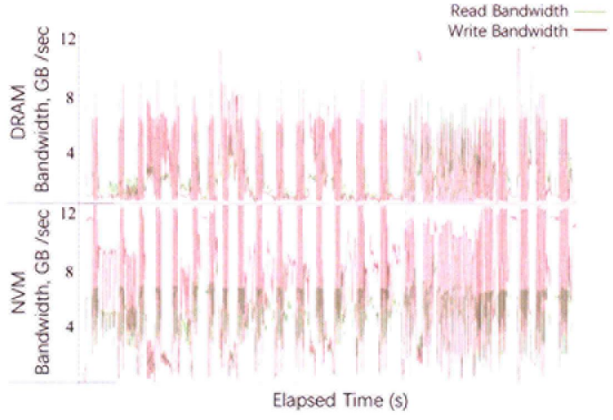
(c) Own framework, 1/3 DRAM

图 5.22　Spark K-Means 的访存带宽

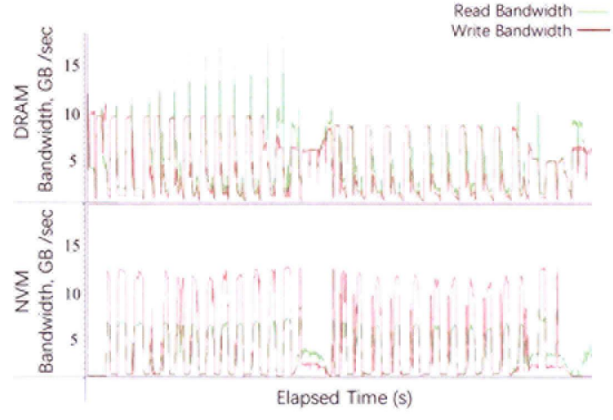
Figure 5.22　Spark K-Means's memory access bandwidth



(a) All DRAM



(b) Un-Managed, 1/3 DRAM



(c) Own framework, 1/3 DRAM

图 5.23　Spark LR 的访存带宽

Figure 5.23　Spark LR's memory access bandwidth

旧生代 (此时均为 NVM) 时，计算便结束，此时大量热数据处于位于 DRAM 的新生代。

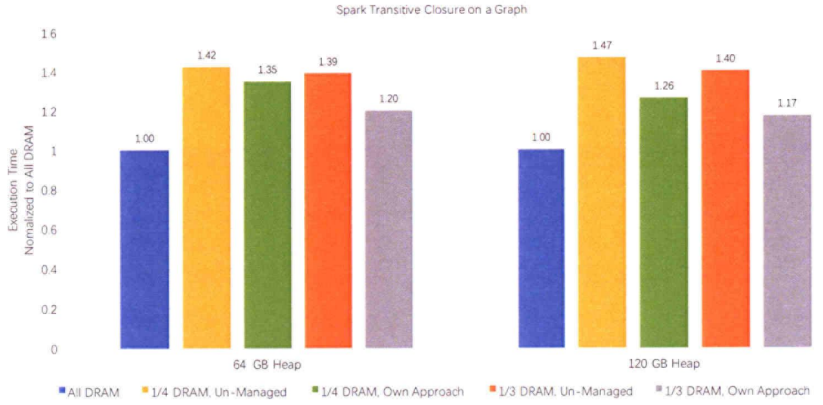


图 5.24 Spark TC 应用框架后的性能分析

Figure 5.24 Spark TC's performance after applying own framework

当异质内存中的 DRAM 比例达到 1/3 时，旧生代的 DRAM Space 有一定空间来容纳热数据。Spark-JVM 协同框架按照 Spark 应用开发人员的 RDD 标注，将对应的热数据迁移到旧生代的 DRAM Space 后，两种 Java Heap 配置下的加速比趋于一致，均和全部使用 DRAM 存在 20% 左右的性能差。这是由于，即使在 64 GB Java Heap 配置下，热数据虽然迅速的被迁出新生代，但是仍旧被置于了旧生代的 DRAM Space, 所以并没有带来性能的下降。另外值得一提的是，Spark TC 中 RDD Analyzer 的标注方式如图 5.8(a) 所示，因为 CoGroupedRDD 和父 RDD 之间的宽依赖关系，其标注了 CoGroupedRDD 对应的数据为 DRAM，而并非对应 Stage 起始的 ShuffledRDD 。

性能优化分析

这里仍旧使用平均带宽和带宽趋势图来阐述数据布局带来的变化。图 5.26 是结合了内存读、写的总体平均带宽图。此时的 Java Heap 和 Spark Executor Memory 仍旧固定在 64GB, 而异质内存中的 DRAM 比例为 1/3 。从图中可以看到，处于无管理 (Un-Managed) 状态下，NVM 端的平均带宽远远高于 DRAM。而从 5.26(b) 来看，Un-Manged 状态下，NVM 端的瞬时读、写带宽都可逼近 12.8GB/s 的瓶颈。因此，往往会造成性能下降。除此之外，正如 Intel 研究院学者提出的分析，即使没有达到带宽瓶颈，仍有一些 Pointer Chasing 类型的访存模式，对内存的延迟十分敏感，造成程序性能下降[10]。因此，尽可能的充分

利用 DRAM，将访存迁移到空间有限的 DRAM 区域以便分担 NVM 压力，是本文的目标。

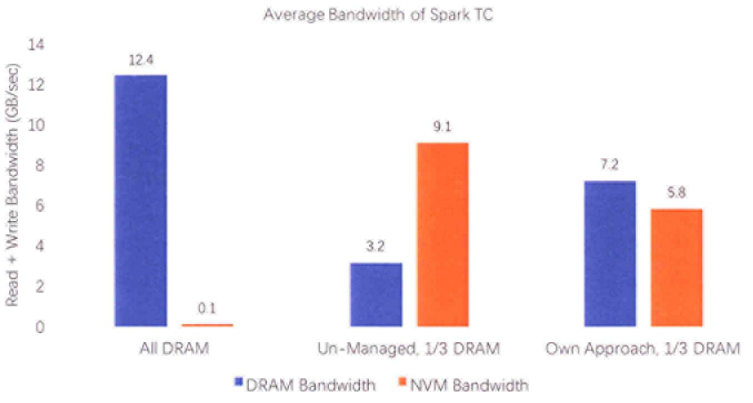


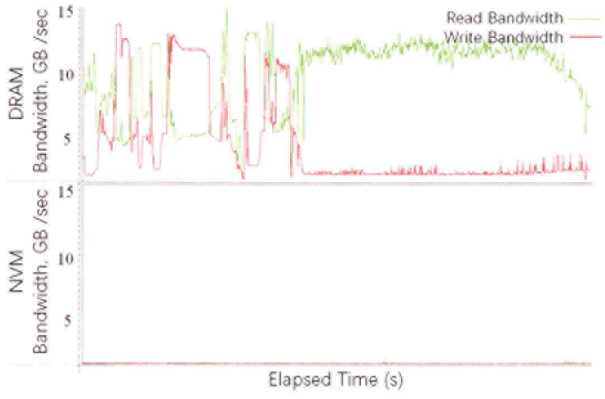
图 5.25　Spark TC 的平均带宽分析

Figure 5.25　Spark TC's average total bandwidth

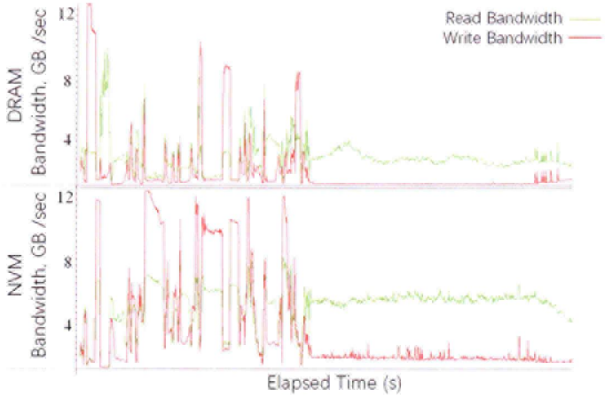
Spark 应用开发人员，将一些热 RDD 或者“支配 RDD”标注为 *DRAM Flag* 后，Spark 层次的 RDD Analyzer 便会将对应的 RDD handler object 进行 *DRAM Flag* 标注。以便相关 Stage 内部的后续计算均在 DRAM 上进行。数据布局后的平均带宽如图 5.26 所示，DRAM 端的访存又成为了主导。根据 5.26(c) 展示，NVM 端仍就有大量的访存，尤其是内存写操作。这是由于“支配 RDD”过于庞大，无法保证将其所有数据迁移到 DRAM，所以，在 1/3 DRAM 比例的异质内存下，经过合理布局后，和全部使用 DRAM 相比，仍旧有 20% 左右的性能差距。

5.8　本章总结

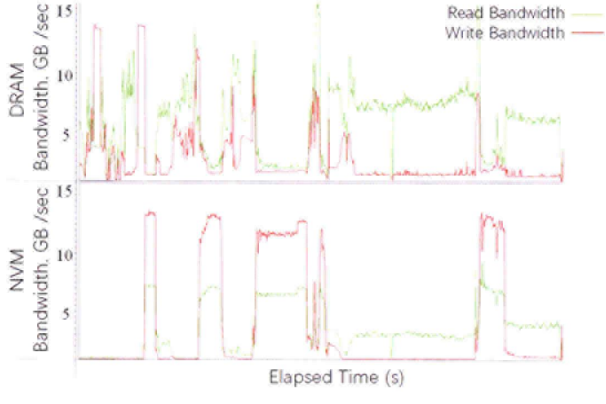
由于内存计算使用的内存容量巨大，因此细粒度的数据划分、布局将会带来严重的开销[5]。同时，内存计算系统的计算行为清晰、简洁，如图 1.3 所示的 Spark PageRank 计算，因此，是否可以让用户基于自己的理解，以大数据架构自定义数据结构为粒度进行数据布局，是本节研究的出发点。最终，本章节提出了一个种 Spark - JVM 协同工作，让 Spark 应用开发人员可以在 RDD 粒度进行数据的异质内存编程框架，并将其实现在 Spark、OpenJDK 之上。通过具体的实验证明，该种数据布局思路可以取得令人满意的数据布局效果。在异质内存中使用 25% - 33% 的 DRAM，便可以达到全部使用 DRAM 80% 以上的性



(a) All DRAM



(b) Un-Managed, 1/3 DRAM



(c) Own framework, 1/3 DRAM

图 5.26　Spark TC 的访存带宽

Figure 5.26　Spark TC's memory access bandwidth

能，在一些应用中，性能差甚至仅有 5% 。

另一方面，通过打通不同系统层次 (Spark、JVM)，传递消息来共同解决问题的方式，不但可以让底层系统获取上层语义，在运行时利用 GC 进行低开销的数据布局，其更带来了面向特定计算行为、数据使用行为进行优化的机会，正如 5.6节展示。

最终，本文提出的 Spark - JVM 协同编程框架，可以利用新型非易失性内存 (Non-Volatile Memory) 来扩展内存计算应用的主存，而非仅仅是作为替代磁盘、SSD 的存储。这给解决由于 DRAM 工艺限制，导致内存容量不足，而致使大数据应用计算规模难以快速发展的问题带来了新的思路。

第 6 章 结论与展望

6.1 本文工作总结

本文详细分析了基于托管式语言开发的应用在异质内存管理方面所面临的挑战与机遇，并提出了利用运行时系统管理异质内存的策略。由于托管式应用基于运行时系统执行，因此操作系统、硬件进行的数据布局策略会被 GC 机制打乱；另一方面，托管式语言自定义的数据对象模型 (如 Java Object Model) 的结构和使用方式与 C/C++ 语言有较大差距，因此大量基于 C/C++ 提出的数据访存分析策略、方法不再适用于托管式应用的访存分析。针对以上问题，本文提出了直接利用运行时系统来对数据进行多粒度的冷、热划分，并利用 GC 机制来低开销的在异质内存进行数据布局的方法。此时，操作系统仅需提供必要的 DRAM、NVM 空间申请支持，无需进行数据的访存监测、划分、移动等操作。该方法有效的解决了多层次数据移动带来的冲突问题，并提供了灵活粒度的数据划分、移动策略。

基于 Trace 的 NVM 模拟器难以支持大规模分布式应用的访存分析，亦难以支撑基于运行时系统的托管式应用的实验分析，本文提出了利用 NUMA 架构进行异质内存仿真的方案。通过利用 NUMA 平台两个处理器之间的 Intel QPI 来制造访存带宽和访存延迟的差异，并通过开发的“干扰程序”来灵活调整远端内存 (Remote Memory) 的访存带宽和延迟，对 NVM 进行仿真。基于该仿真平台，本文通过真机实验系统的分析了程序的物理访存、性能等指标，并考虑了诸如 Cache 等结构的影响。

针对单机应用，本文提出了一套两级粒度的冷、热数据划分策略。现有运行时堆 (Java heap) 按照数据的生命周期、结构特点、用途等被划分为了多个区域。本文定量分析了各个运行时堆区域内部数据的冷、热属性，以及基于异质内存仿真平台分析了各个运行时堆区域的布局对程序的性能影响，并基于此提出，可以直接按照运行时堆的现有划分来进行粗粒度的数据布局。另一方面，本文通过程序分析发现，一个托管式应用虽然包含了数千个函数，然而大部分的 LLC Miss 却由极少量的函数造成。基于该发现，本文提出了基于函数

的细粒度冷、热数据划分方法，并开发了工具来进行自动识别造成高访存密度(High Memory Access Density)的热数据。该策略有效的解决了传统C/C++数据访存分析方法无法对托管式应用的数据对象进行分析的问题。最后实验表明，通过应用本文的数据划分策略，并利用GC进行数据布局，可以在异质内存中仅含有15% - 25%的DRAM时，和全部使用DRAM仅有5% - 25%的性能差。本文的工作基于科研型运行时系统JikesRVM进行。

本文通过分析典型的内存计算迭代应用(Spark应用)，发现其主要设计思想为追求高并发、高容错性，往往具有自定义的数据单元抽象，如Spark RDD，同时其每个数据单元内部的数据具有类似的访存模式和生命周期。通过分析其数据的计算行为、数据使用行为发现，其内部的数据同时具有大数据迭代应用的“迭代周期性生命”特征和“大量数据快速死亡”的单机应用特征，同时其还具有内存计算所特有的“开发人员控制的生命周期”特征。此外，本文分析了内存不足时，给内存计算应用带来的性能瓶颈原因，以及其与运行时系统之间的数据交互行为。基于以上分析，本文提出了可以允许应用开发人员在自定义数据结构粒度(RDD)进行数据布局的编程接口，以及具体的数据布局策略指导规范。并通过实验证明，仅通过粗粒度布局便可显著优化内存计算迭代应用在异质内存中的性能。

此外，本文提出了“大数据框架-运行时系统协同数据布局”的策略来实现在运行时堆(Java heap)中以自定义数据结构粒度(RDD粒度)进行数据布局。RDD为Spark提出的分布式数据抽象，其穿透三层数据抽象(Spark RDD、Scala变量、Java Object Model)对应了运行时堆中数百万的独立数据对象(Object)。运行时系统无法感知这些数据对象(Object)之间的关系，并会在GC过程频繁、无序的移动这些数据。针对该问题，本文提出了一种大数据框架到运行时系统的数据信息传递通道。当应用开发人员对单一变量进行数据信息标注后，如DRAM、NVM等数据冷热信息，该通道便会在Spark层中的相关RDD之间传递标注信息，并按照本文提出的“Migration-Driven RDD Allocation”的策略在运行时系统中的数据对象之间传递标注信息，并重新调整相关数据的创建、移动行为。此外，当运行时系统中的数据对象按照上层数据属性进行划分后，我们针对不同类型的数据对象亦提出了对应的GC行为优化。

最终，本文总结以上异质内存管理策略，基于内存计算框架Spark、运行

时系统 OpenJDK 提出了贯穿大数据应用、大数据框架、运行时系统与异质内存的一体化异质内存管理框架。该框架提供了易用的编程接口，仅需对现有 Spark 应用进行轻微修改，便可将其数据合理布局到异质内存。通过针对 Spark PageRank、K-Means、Transitive closure、Logistic regression based classification 等应用的测试，在异质内存中仅有 25% - 35% DRAM 时，通过该框架调整数据布局后，可以达到使用和异质内存等量 DRAM 时 80% - 97% 的性能。

6.2 未来研究内容

针对托管式应用在异质内存之上的数据布局问题，本文提出直接通过运行时系统本身来进行数据的划分、布局，并取得了初步的成果。由于托管式语言被广泛的应用于嵌入式、分布式等领域的应用开发，因此未来还有广泛的研究空间：

- 将“大数据框架-运行时系统协同工作”的思想进一步推广到其他大数据框架，并尝试提出一种更广泛的运行时系统管理异质内存数据布局的策略。诸如内存数据库、图计算等诸多领域均有基于托管式语言开发的编程框架，同时这些大数据框架的数据使用特点具有共性，因此，分析这些大数据框架的计算行为、数据使用行为，并利用类似的“跨层次协同合作”的思想优化其异质内存管理策略是一个值得探索的方向。
- 利用“信息传递通道机制”进行其他方向的性能优化。本文提出的数据信息传递通道可以将上层程序语义传递到底层运行时系统。在获取这些信息后，运行时系统可以将孤立的数据对象进行分类，并针对不同的数据类型进行更多的数据优化。亦或是根据获取的语义信息调整运行时的 GC 等机制，使大数据应用和底层运行时系统之间的计算行为、数据使用行为重新匹配。
- 将运行时系统的粗粒度数据划分和操作系统页粒度数据划分结合。设计更多级的灵活粒度数据划分策略，在运行时的粗粒度划分之下，操作系统可以在页 (Page) 粒度对部分数据进行精细的冷、热数据划分。同时，可以调整 GC 机制不再去打乱操作系统的良好数据布局。
- 优化大数据框架自身的内存管理效率。Spark 等内存计算框架往往具有自己的内存管理模型，然而通过本文分析可知，其自身的内存信息统计精确度较差、策略粗糙。因此，可以从运行时系统获取其所需的信息用

于调整自身的内存管理效率。

参考文献

[1] Deloitte. In-Memory Computing technology. The holy grail of analytics?[EB/OL]. (2013) [2013]. https://www2.deloitte.com/content/dam/Deloitte/de/Documents/technology-media-telecommunications/TMT_Studie_In_Memory_Computing.pdf.

[2] Atikoglu B, Xu Y, Frachtenberg E, et al. Workload Analysis of a Large-scale Key-value Store[C/OL]// Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems. SIGMETRICS '12. London, England, UK: ACM, 2012: 53–64. ISBN: 978-1-4503-1097-0. http://doi.acm.org/10.1145/2254756.2254766. DOI: 10.1145/2254756.2254766.

[3] Ousterhout J, Agrawal P, Erickson D, et al. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM[J/OL]. SIGOPS Oper. Syst. Rev., 2010, 43(4): 92–105. http://doi.acm.org/10.1145/1713254.1713276. DOI: 10.1145/1713254.1713276. ISSN: 0163-5980.

[4] Gonzalez J E, Low Y, Gu H, et al. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs[C/OL]// Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012: 17–30. ISBN: 978-1-931971-96-6. http://dl.acm.org/citation.cfm?id=2387880.2387883.

[5] Wei W, Jiang D, McKee S A, et al. Exploiting Program Semantics to Place Data in Hybrid Memory[C]// PACT. IEEE, 2015: 163–173.

[6] Zilberberg O, Weiss S, Toledo S. Phase-change Memory: An Architectural Perspective[J/OL]. ACM Comput. Surv., 2013, 45(3): 29:1–29:33. http://doi.acm.org/10.1145/2480741.2480746. DOI: 10.1145/2480741.2480746. ISSN: 0360-0300.

[7] Loh G H. 3D-Stacked Memory Architectures for Multi-core Processors[C/OL]// Proceedings of the 35th Annual International Symposium on Computer Architecture. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008: 453–464. ISBN: 978-0-7695-3174-8. http://dx.doi.org/10.1109/ISCA.2008.15. DOI: 10.1109/ISCA.2008.15.

[8] Zaharia M, Chowdhury M, Das T, et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing[C/OL]// Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). San Jose, CA: USENIX, 2012: 15–28. ISBN: 978-931971-92-8. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[9] Apache. Apache™ Hadoop[EB/OL]. (2014) [2018]. http://hadoop.apache.org.

[10] Dulloor S R, Roy A, Zhao Z, et al. Data Tiering in Heterogeneous Memory Systems[C]// Proceedings of the Eleventh European Conference on Computer Systems. EuroSys '16. London, United Kingdom: ACM, 2016: 15:1–15:16. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901344.

[11] Freitas R F, Wilcke W W. Storage-class memory: The next storage system technology[J]. IBM Journal of Research and Development, 2008, 52(4.5): 439–447. DOI: 10.1147/rd.524.0439. ISSN: 0018-8646.

[12] Ramos L E, Gorbatov E, et Al. Page placement in hybrid memory systems[C]//ICS. ACM, 2011: 85–95.

[13] Qureshi M K, Srinivasan V, et Al. Scalable high performance main memory system using phase-change memory technology[C/OL]//ISCA '09. ACM, 2009: 24–33. http://doi.acm.org/10.1145/1555754.1555760. DOI: 10.1145/1555754.1555760.

[14] Mogul J C, Argollo E, Shah M, et al. Operating System Support for NVM+DRAM Hybrid Main Memory[C/OL]// Proceedings of the 12th Conference on Hot Topics in Operating Systems. HotOS'09. Monte Verità, Switzerland: USENIX Association, 2009: 14–14. http://dl.acm.org/citation.cfm?id=1855568.1855582.

[15] Agarwal N, Nellans D, Stephenson M, et al. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems[C/OL]// Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '15. Istanbul, Turkey: ACM, 2015: 607–618. ISBN: 978-1-4503-2835-7. http://doi.acm.org/10.1145/2694344.2694381. DOI: 10.1145/2694344.2694381.

[16] Zhou P, Zhao B, Yang J, et al. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology[C]// Proceedings of the 36th Annual International Symposium on Computer Architecture. ISCA '09. Austin, TX, USA: ACM, 2009: 14–23. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555759.

[17] Oracle. HotSpot[EB/OL]. (2018). http://www.oracle.com/technetwork/java/whitepaper-135217.html.

[18] Wang C, Cao T, Zigman J, et al. Efficient Management for Hybrid Memory in Managed Language Runtime[M]// Network and Parallel Computing: 13th IFIP WG 10.3 International Conference, NPC 2016, Xi'an, China, October 28-29, 2016, Proceedings. Ed. by Gao G R, Qian D, Gao X, et al. Cham: Springer International Publishing, 2016: 29–42. DOI: 10.1007/978-3-319-47099-3_3. ISBN: 978-3-319-47099-3.

[19] Blackburn S M, McKinley K S. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance[C/OL]// PLDI. 2008. http://doi.acm.org/10.1145/1375581.1375586. DOI: 10.1145/1375581.1375586.

[20] Tang X, Zhai J, Yu B, et al. Self-Checkpoint: An In-Memory Checkpoint Method Using Less Space and Its Practice on Fault-Tolerant HPL[C]// Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '17. New York, NY, USA: ACM, 2017.

[21] Lu L, Shi X, Zhou Y, et al. Lifetime-based Memory Management for Distributed Data Processing Systems[J/OL]. Proc. VLDB Endow., 2016, 9(12): 936–947. http://dx.doi.org/10.14778/2994509.2994513. DOI: 10.14778/2994509.2994513. ISSN: 2150-8097.

[22] Zaharia M, Konwinski A, Joseph A D, et al. Improving MapReduce Performance in Heterogeneous Environments[C/OL]// Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. OSDI'08. San Diego, California: USENIX Association, 2008: 29–42. http://dl.acm.org/citation.cfm?id=1855741.1855744.

[23] Ananthanarayanan G, Kandula S, Greenberg A, et al. Reining in the Outliers in Map-reduce Clusters Using Mantri[C/OL]// Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010: 265–278. http://dl.acm.org/citation.cfm?id=1924943.1924962.

[24] Nguyen K, Fang L, Xu G, et al. Yak: A High-performance Big-data-friendly Garbage Collector[C/OL]// Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. OSDI'16. Savannah, GA, USA: USENIX Association, 2016: 349–365. ISBN: 978-1-931971-33-1. http://dl.acm.org/citation.cfm?id=3026877.3026905.

[25] Kgil T, Roberts D, Mudge T. Improving NAND Flash Based Disk Caches[C]// Proceedings of the 35th Annual International Symposium on Computer Architecture. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008: 327–338. ISBN: 978-0-7695-3174-8. DOI: 10.1109/ISCA.2008.32.

[26] Mangalagiri P, Sarpatwari K, Yanamandra A, et al. A Low-power Phase Change Memory Based Hybrid Cache Architecture[C]// Proceedings of the 18th ACM Great Lakes Symposium on VLSI. GLSVLSI '08. Orlando, Florida, USA: ACM, 2008: 395–398. ISBN: 978-1-59593-999-9. DOI: 10.1145/1366110.1366204.

[27] Lee B C, Ipek E, Mutlu O, et al. Architecting Phase Change Memory As a Scalable Dram Alternative[C]// Proceedings of the 36th Annual International Symposium on Computer Architecture. ISCA '09. Austin, TX, USA: ACM, 2009: 2–13. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555758.

[28] Huang H, Pillai P, Shin K G. Design and Implementation of Power-aware Virtual Memory[C/OL]// Proceedings of the Annual Conference on USENIX Annual Technical Conference. ATEC '03. San Antonio, Texas: USENIX Association, 2003: 5–5. http://dl.acm.org/citation.cfm?id=1247340.1247345.

[29] Zhang W, Li T. Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures[C]// PACT '09. 2009. DOI: 10.1109/PACT.2009.30.

[30] Pandey V, Jiang W, Zhou Y, et al. DMA-aware memory energy management[C]// The Twelfth International Symposium on High-Performance Computer Architecture, 2006. 2006: 133–144. DOI: 10.1109/HPCA.2006.1598120.

[31] Mesnier M P, Akers J B. Differentiated Storage Services[J/OL]. SIGOPS Oper. Syst. Rev., 2011, 45(1): 45–53. http://doi.acm.org/10.1145/1945023.1945030. DOI: 10.1145/1945023.1945030. ISSN: 0163-5980.

[32] Sivathanu M, Prabhakaran V, Popovici F I, et al. Semantically-Smart Disk Systems[C/OL]// Proceedings of the 2Nd USENIX Conference on File and Storage Technologies. FAST '03. San Francisco, CA: USENIX Association, 2003: 73–88. http://dl.acm.org/citation.cfm?id=1090694.1090702.

[33] Hassan A, Vandierendonck H, et Al. Software-managed energy-efficient hybrid DRAM/NVM main memory[C/OL]// CF. ACM, 2015: 23:1–23:8. http://doi.acm.org/10.1145/2742854.2742886. DOI: 10.1145/2742854.2742886.

[34] Dulloor S R, Kumar S, Keshavamurthy A, et al. System Software for Persistent Memory[C/OL]// Proceedings of the Ninth European Conference on Computer Systems. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014: 15:1–15:15. ISBN: 978-1-4503-2704-6. http://doi.acm.org/10.1145/2592798.2592814. DOI: 10.1145/2592798.2592814.

[35] NERSC. National Energy Research Scientific Computing Center - Cori[EB/OL]. (2017). http://www.nersc.gov/users/computational-systems/cori/.

[36] Oak Ridge Leadership Computing Facility - Summit[EB/OL]. (2018). https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.

[37] Texas Advanced Computing Center - Stampede.[EB/OL]. (2016). https://www.tacc.utexas.edu/stampede/.

[38] Kim J, Lee S, Vetter J S. PapyrusKV: A High-performance Parallel Key-value Store for Distributed NVM Architectures[C/OL]// Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '17. Denver, Colorado: ACM, 2017: 57:1–57:14. ISBN: 978-1-4503-5114-0. http://doi.acm.org/10.1145/3126908.3126943. DOI: 10.1145/3126908.3126943.

[39] Piccoli G, Santos H N, Rodrigues R E, et al. Compiler Support for Selective Page Migration in NUMA Architectures[C/OL]// Proceedings of the 23rd International Conference on Parallel Architectures and Compilation. PACT '14. Edmonton, AB, Canada: ACM, 2014: 369–380. ISBN: 978-1-4503-2809-8. http://doi.acm.org/10.1145/2628071.2628077. DOI: 10.1145/2628071.2628077.

[40] Huang H, Shin K G, Lefurgy C, et al. Improving Energy Efficiency by Making DRAM Less Randomly Accessed[C/OL]// Proceedings of the 2005 International Symposium on Low Power Electronics and Design. ISLPED '05. San Diego, CA, USA: ACM, 2005: 393–398. ISBN: 1-59593-137-6. http://doi.acm.org/10.1145/1077603.1077696. DOI: 10.1145/1077603.1077696.

[41] Oracle. OpenJDK[EB/OL]. (2018). http://openjdk.java.net/.

[42] Appel A W. Simple Generational Garbage Collection and Fast Allocation[J/OL]. Softw. Pract. Exper., 1989, 19(2): 171–183. http://dx.doi.org/10.1002/spe.4380190206. DOI: 10.1002/spe.4380190206. ISSN: 0038-0644.

[43] Stefanović D, McKinley K S, Moss J E B. Age-based Garbage Collection[C/OL]// Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA '99. Denver, Colorado, USA: ACM, 1999: 370–381. ISBN: 1-58113-238-7. http://doi.acm.org/10.1145/320384.320425. DOI: 10.1145/320384.320425.

[44] Bu Y, Borkar V, Xu G, et al. A Bloat-aware Design for Big Data Applications[C/OL]// Proceedings of the 2013 International Symposium on Memory Management. ISMM '13. Seattle, Washington, USA: ACM, 2013: 119–130. ISBN: 978-1-4503-2100-6. http://doi.acm.org/10.1145/2464157.2466485. DOI: 10.1145/2464157.2466485.

[45] Nguyen K, Wang K, Bu Y, et al. FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications[C/OL]// Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '15. Istanbul, Turkey: ACM, 2015: 675–690. ISBN: 978-1-4503-2835-7. http://doi.acm.org/10.1145/2694344.2694345. DOI: 10.1145/2694344.2694345.

[46] Xin R, Rosen J. Project Tungsten: Bringing Apache Spark Closer to Bare Metal[EB/OL]. (0201). https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[47] Boyapati C, Salcianu A, Beebee W Jr., et al. Ownership Types for Safe Region-based Memory Management in Real-time Java[C/OL]// Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. PLDI '03. San Diego, California, USA: ACM, 2003: 324–337. ISBN: 1-58113-662-5. http://doi.acm.org/10.1145/781131.781168. DOI: 10.1145/781131.781168.

[48] Beebee W S, Rinard M C. An Implementation of Scoped Memory for Real-Time Java[C/OL]// Proceedings of the First International Workshop on Embedded Software. EMSOFT '01. London, UK, UK: Springer-Verlag, 2001: 289–305. ISBN: 3-540-42673-6. http://dl.acm.org/citation.cfm?id=646787.703892.

[49] He W, Cui H, Lu B, et al. Hadoop+: Modeling and Evaluating the Heterogeneity for MapReduce Applications in Heterogeneous Clusters[C/OL]// Proceedings of the 29th ACM on International Conference on Supercomputing. ICS '15. Newport Beach, California, USA: ACM, 2015: 143–153. ISBN: 978-1-4503-3559-1. http://doi.acm.org/10.1145/2751205.2751236. DOI: 10.1145/2751205.2751236.

[50] Grossman M, Sarkar V. SWAT: A Programmable, In-Memory, Distributed, High-Performance Computing Platform[C/OL]// Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing. HPDC '16. Kyoto, Japan: ACM, 2016: 81–92. ISBN: 978-1-4503-4314-5. http://doi.acm.org/10.1145/2907294.2907307. DOI: 10.1145/2907294.2907307.

[51] Ghasemi E, Chow P. A Scalable Heterogeneous Dataflow Architecture For Big Data Analytics Using FPGAs (Abstract Only)[C/OL]// Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. FPGA '16. Monterey, California, USA: ACM, 2016: 274–274. ISBN: 978-1-4503-3856-1. http://doi.acm.org/10.1145/2847263.2847294. DOI: 10.1145/2847263.2847294.

[52] Gao T, Strauss K, Blackburn S M, et al. Using managed runtime systems to tolerate holes in wearable memories[C]// Boehm H, Flanagan C. PLDI '13. Ed. by Boehm H, Flanagan C. ACM, 2013: 297–308. DOI: 10.1145/2462156.2462171.

[53] Wu M, Ziming Z, Haoyu L, et al. Espresso: Brewing Java For More Non-Volatility[C]// Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '18. VA, USA: ACM, 2018.

[54] Oracle. Java Persistence API[EB/OL]. (2018). https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html.

[55] Intel. Persistent Memory Programming[EB/OL]. (2018) [2018]. http://pmem.io/.

[56] Nakagawa G, Oikawa S. NVM/DRAM hybrid memory management with language runtime support via MRW queue[C]// SNPD. IEEE, 2015: 357–362. DOI: 10.1109/SNPD.2015.7176226.

[57] Nakagawa G, Oikawa S. Preliminary Analysis of a Write Reduction Method for Nonvolatile Main Memory on Jikes RVM[C]// CANDAR. 2013: 597–601. DOI: 10.1109/CANDAR.2013.107.

[58] 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]// IIAIAAI. 2014. DOI: 10.1109/IIAI-AAI.2014.128.

[59] Nakagawa G, Oikawa S. Language runtime support for NVM/DRAM hybrid main memory[C]// COOL Chips XVII, 2014 IEEE. 2014: 1–3. DOI: 10.1109/CoolChips.2014.6842949.

[60] Blackburn S M, Garner R, Hoffmann C, et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis[C/OL]// Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications. OOPSLA '06. Portland, Oregon, USA: ACM, 2006: 169–190. ISBN: 1-59593-348-4. http://doi.acm.org/10.1145/1167473.1167488. DOI: 10.1145/1167473.1167488.

[61] Yang T, Hertz M, Berger E D, et al. Automatic heap sizing: taking real memory into account[C]// ISMM. ACM, 2004: 61–72. DOI: 10.1145/1029873.1029881.

[62] Hertz M, Feng Y, Berger E D. Garbage collection without paging[C]// PLDI. ACM, 2005: 143–153. DOI: 10.1145/1065010.1065028.

[63] Yang T, Berger E D, Kaplan S F, et al. CRAMM: Virtual Memory Support for Garbage-Collected Applications[C]// OSDI. USENIX Association, 2006: 103–116.

[64] Inoue H, Nakatani T. Identifying the sources of cache misses in Java programs without relying on hardware counters[C]// ISMM. ACM, 2012: 133–142. DOI: 10.1145/2258996.2259014.

[65] Frampton D, Bacon D F, Cheng P, et al. Generational Real-time Garbage Collection: A Three-part Invention for Young Objects[C/OL]// Proceedings of the 21st European Conference on Object-Oriented Programming. ECOOP'07. Berlin, Germany: Springer-Verlag, 2007: 101–125. ISBN: 3-540-73588-7, 978-3-540-73588-5. http://dl.acm.org/citation.cfm?id=2394758.2394767.

[66] Kannan S, Gavrilovska A, Schwan K. PVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage[C/OL]// Proceedings of the Eleventh European Conference on Computer Systems. EuroSys '16. London, United Kingdom: ACM, 2016: 13:1–13:16. ISBN: 978-1-4503-4240-7. http://doi.acm.org/10.1145/2901318.2901325. DOI: 10.1145/2901318.2901325.

[67] Coburn J, Caulfield A M, Akel A, et al. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories[C/OL]// Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011: 105–118. ISBN: 978-1-4503-0266-1. http://doi.acm.org/10.1145/1950365.1950380. DOI: 10.1145/1950365.1950380.

[68] Bobrow D G. Managing Reentrant Structures Using Reference Counts[J/OL]. ACM Trans. Program. Lang. Syst., 1980, 2(3): 269–273. http://doi.acm.org/10.1145/357103.357104. DOI: 10.1145/357103.357104. ISSN: 0164-0925.

[69] Gay D, Aiken A. Language Support for Regions[C/OL]// Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. PLDI '01. Snowbird, Utah, USA: ACM, 2001: 70–80. ISBN: 1-58113-414-2. http://doi.acm.org/10.1145/378795.378815. DOI: 10.1145/378795.378815.

[70] GmbH J. The Linux Programming Interface - mbind[EB/OL]. (2018). http://man7.org/linux/man-pages/man2/mbind.2.html.

[71] 王晨曦, 吕方, 崔慧敏, 等. 面向大数据处理的基于Spark的异质内存编程框架[J/OL]. 计算机研究与发展, 2018, 55(2), 1: 246–264. http://crad.ict.ac.cn/CN/10.7544/issn1000-1239.2018.20170687. DOI: 10.7544/issn1000-1239.2018.20170687.

[72] Intel. Memory Latency Checker[EB/OL]. (2018) [2018]. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[73] McCalpin J D. STREAM Benchmark[EB/OL]. (2018). https://www.cs.virginia.edu/stream/.

[74] Bhattacharjee A, Martonosi M. Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors[C/OL]// Proceedings of the 36th Annual International Symposium on Computer Architecture. ISCA '09. Austin, TX, USA: ACM, 2009: 290–301. ISBN: 978-1-60558-526-0. http://doi.acm.org/10.1145/1555754.1555792. DOI: 10.1145/1555754.1555792.

[75] Ferdman M, Adileh A, Kocberber O, et al. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware[C/OL]// Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XVII. London, England, UK: ACM, 2012: 37–48. ISBN: 978-1-4503-0759-8. http://doi.acm.org/10.1145/2150976.2150982. DOI: 10.1145/2150976.2150982.

[76] Xin R, Rosen J. ASM 2.0 Bytecode Framework[EB/OL]. (2018). http://asm.ow2.org/doc/tutorial-asm-2.0.html.

[77] Lausanne (EPFL) Lausanne. The Scala Programming Language[EB/OL]. (2018) [2018]. https://www.scala-lang.org/.

[78] Intel. Intel® Optane™ Technology[EB/OL]. (2018) [2018]. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[79] Ousterhout K, Rasti R, Ratnasamy S, et al. Making Sense of Performance in Data Analytics Frameworks[C/OL]// 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). Oakland, CA: USENIX Association, 2015: 293–307. ISBN: 978-1-931971-218. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout.

[80] Zhan J. A Big Data and AI Benchmark Suite[EB/OL]. (2018). http://prof.ict.ac.cn/.

[81] Lê N M, Pop A, Cohen A, et al. Correct and Efficient Work-stealing for Weak Memory Models[C/OL]// Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '13. Shenzhen, China: ACM, 2013: 69–80. ISBN: 978-1-4503-1922-5. http://doi.acm.org/10.1145/2442516.2442524. DOI: 10.1145/2442516.2442524.

[82] Arora N S, Blumofe R D, Plaxton C G. Thread Scheduling for Multiprogrammed Multiprocessors[C/OL]// Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures. SPAA '98. Puerto Vallarta, Mexico: ACM, 1998: 119–129. ISBN: 0-89791-989-0. http://doi.acm.org/10.1145/277651.277678. DOI: 10.1145/277651.277678.

致 谢

在博士生涯即将结束之际，回想五年的岁月，感慨万千。对所有陪伴我走过这一路的人，谨在此写下感激之言。

首先衷心感谢我的导师冯晓兵研究员。是您给了我这个宝贵的机会，才使我得以进入科研领域探索未知。在我刚入学时，是您耐心的教育我、引导我，帮我敞开了学术的大门。在科研上遇到挫折时，是您一次次耐心的跟我分析问题，探讨解决方案。即使面对未接触过的全新问题，您也会积极思想、学习并和我反复讨论，最终找到答案。是您的帮助使我逐渐有勇气面对未知、探索未知、战胜未知。是您教育我要成长为一个可以独立应对困难的科研人，也因此让我有勇气下决心踏上去海外的全新学术征程。感谢您的帮助和鼓励，也感谢您一直以来的支持和信任。

我要感谢吕方老师。作为老师您的第一个博士生，我能切身感觉到您努力从师姐转变为老师的角色时的艰辛和付出。在和老师一次次的问题探讨中，我才逐渐完成了自己知识体系的构建。是老师的包容和理解，让我不断的成长为有担当的、有耐心的博士生。感谢老师一次次的帮我解决科研难题和生活中的坎坷，使我从一次次的在挫折中成长。

我要感谢曹婷师姐。是您影响了我博士生涯以及未来的学术风格。第一次见面，是在师姐的一个报告会上。拥有丰富知识和杰出成果的师姐，并没有看低我略显稚嫩的科学问题，而是用全面、严谨的回答让我对虚拟机方向有了浓厚的兴趣，并乐于在这个领域思考如何解决遇到的未知问题。在随后的学习中，是您和 John 的严谨学术风格，让我养成了一个学者应有的严谨品质。

我还要感谢崔慧敏老师。是您向我展示了杰出学者的智慧和乐观的品质。每每遇到难以跨越的方向性问题，都怀着忐忑的心情会向忙碌的老师请教。然而，您却每次都会从百忙之中抽出时间帮我分析问题，并给出建设性的指导。

在此，还要感谢编译实验中对我帮助的几位老师，感谢李炼老师、武成岗老师、王蕾老师、陈莉老师、刘磊老师对我的帮助和关心。是刘磊老师跟我探讨科学问题、探讨论文写作，让我提升了自己的视野和能力。

感谢师兄、师姐对我的帮助和指导。在此感谢王蕾师兄、赵家程师兄、王

喆师兄。也要感谢陪伴我的小伙伴们，是你们陪我度过了五年中的每一天，感谢阮功、任晟民、杨帆、郭磊、胡丹琪、庄良吉、李登辉、夏春伟、伍明川、谢梦瑶、杜围韦。

由于这五年来，学业繁重，没能有时间和朋友们多联系，然而，每当我有任何需要，你们总是第一个站出来帮我解决难题。在此，感谢我的发小王腾瑶、刘辉、郭冠宁，谢谢你们的帮助和陪伴。

最后，要特别感谢我的家人。我的父亲是一位睿智而严厉的教育工作者，然而却对狂野成长的我充满了容忍。容忍我遇到困难时懦弱，容忍我遇事时的急躁和不成熟。然而，父亲却始终坚信我能战胜自己。父亲在自己的笔记本上写下“莫道晨曦微，红霞将满天”。这让我自豪而羞愧。愿自己终有一天成长为一个问心无愧的人。

母亲将自己的青春奉献给了这个家庭。母亲常说，他这一辈子最自豪的就是我的兄长和我，我相信母亲的话，也知道母亲为我们受的苦。父亲曾对我说，“不管你们到哪里，你们的母亲都会在深夜为你们留一盏灯，等待你们回家。”我曾经无法理解这种感情，觉得造作。然而，每次深夜加班回宿舍，在空无一人的北京马路上，都会想起这句话。我相信不管我在哪里，不管我是成功还是失败，我的母亲都会给我留一盏灯。

我要感谢我的兄长。他是我从小的偶像。随着年近 30，见到了很多天才，然而，我心里还是最敬佩我的兄长。并会随时自豪的说一句，我有一个亲哥哥。感谢我的嫂子。嫂子在我和我爱人最困难的时候，对我们伸出了援助之手。让我和爱人有勇气在这个陌生的城市生活下去。当然，也要感谢刚出生的大侄女给我们全家带来的快乐！

我的爱人，谢谢你。我们已经一起走过了 7 年的风风雨雨。是你陪伴我从懵懂走向了稳重和成熟。是你的谅解和隐忍促进了我的成长。我将赴美深造，而你却支持我，鼓励我去拼搏。你说，你看出了我心中的志向，你想让我在科研的路上闯一闯。我能看到这句话背后，你付出的艰辛和痛苦。谢谢你。

谨在此感谢 ucasthesis，它的存在让我的论文写作轻松自在了许多。这是我的好朋友朝鲁开发的模板，无论是在我撰写博士论文期间，还是平时的科研中，朝鲁都会耐心的帮我解决问题。在此对我的好朋友表示衷心的感谢。

作者简历及攻读学位期间发表的学术论文与研究成果

姓名：王晨曦　　性别：男　　出生日期：1990.3.10　　籍贯：河北

2013.9 – 今　　中国科学院计算技术所直接攻读博士研究生

2009.9 – 2013.7　　天津大学本科生

攻读博士学位期间发表的论文：

[1] **Chenxi Wang**, Ting Cao, John Zigman, Fang Lv, Yunquan Zhang, and Xiaobing Feng. Efficient Management for Hybrid Memory in Managed Language Runtime[C] , IFIP International Conference on Network and Parallel Computing, 2016. (CCF C)

[2] **王晨曦**, 吕方, 崔慧敏, 曹婷, John Zigman, 庄良吉, 冯晓兵. 面向大数据处理的基于 Spark 的异质内存编程框架 [J]. 计算机研究与发展, 2018, 55(2), 1: 246-264

[3] Danqi Hu, Fang Lv, **Chenxi Wang**, Huimin Cui, Lei Wang, Ying Liu, Xiaobing Feng. NVM Streaker: a Fast and Reconfigurable Performance Simulator for Non-Volatile Memory Based Memory Architecture, Journal of Supercomputing, (Accepted, CCF C)

攻读博士学位期间申请的专利：

[1] **王晨曦**; 吕方; 冯晓兵; 刘颖. 为内存控制器分配硬件加速指令的方法和装置, CN105988952A, 已公开（正在申请美国、欧洲专利, US20170351525A1、EP3252611A4 ）

[2] 吕方; **王晨曦**; 黄磊; 冯晓兵; 崔慧敏; 王蕾. 一种面向可变粒度内存系统的二进制文件重写方法, CN104679477A, 已授权

攻读博士学位期间参加的科研项目：

[1] “高通量服务器研究项目”，课题 5，面向消息式内存的编译支持，2013 年 10 月 - 2014 年 7 月

[2] 国家高技术研究发展计划 (863)，“面向大数据的内存计算关键技术与系统”，2015AA015306，2015 年 - 今

攻读博士学位期间的获奖情况：

[1] 2014 年获得中国科学院大学生奖学金

[2] 2015 年获得中国科学院计算技术研究所“所长优秀奖”

[3] 2016 年获得中国科学院计算技术研究所“曙光博士生奖学金”

[4] 2017 年被评为中国科学院大学“三好学生”