统一的整体内存管理，支持多个大数据处理框架在混合内存环境下的运行

简介

以Spark和QuickCached等系统为代表的现代大数据计算对内存的需求极大。缺乏内存可能导致一系列严重的功能和性能问题，包括内存溢出崩溃、效率显著降低，甚至在节点故障时丢失数据。完全依赖DRAM来满足数据中心的内存需求在多个方面都很昂贵，例如，大容量的DRAM昂贵且能源效率低；此外，DRAM的相对小容量意味着通常需要大量的机器来提供足够的内存，导致不能轻松并行化的工作负载下CPU资源被低效利用。

新兴的非易失性存储器（NVM），如相变存储器（PCM）、电阻式随机存取存储器（RRAM）、自旋转矩存储器（STT-MRAM）或3D XPoint，是一种具有巨大内存容量、能效高和每GB成本低的有前途的技术，使其成为传统DRAM的补充。NVM位于内存总线上，可通过加载/存储指令访问，从而实现对内存中持久数据的直接操作。

NVM有两种典型的用途，第一种用途是利用其持久性特性，确保持久性数据结构在发生故障时保持一致，并在失败事件后恢复执行。第二种用途是作为传统DRAM设备的补充，即构建混合内存体系结构，充分利用DRAM的访问速度和NVM的大容量、低功耗和低每GB成本。我们提出的方法属于第二类。近年来，混合内存的系统受到了学术界和工业界的广泛关注。将NVM与DRAM混合在一起对大数据系统的好处显而易见——NVM的高容量使得实现统一的整体内存管理成为可能。使用少量计算节点的大数据工作负载的高内存需求，有望显著降低大型数据中心的硬件和能源成本。

问题

尽管将NVM用于大数据系统是一个有前途的方向，但这一思想尚未得到充分的探索。简单地添加NVM可能导致性能大幅下降，因为其访问延迟显著增加，带宽减少，例如，NVM读取的延迟比DRAM读取大2-4倍，NVM的带宽约为DRAM的1/8-1/3。因此，围绕混合内存的所有研究都围绕着一个关键的研究问题，即如何在DRAM和NVM之间执行智能数据分配和迁移，以在最大程度上提高整体能效，同时最小化性能开销？为了在大数据处理的背景下回答这个问题，存在两个主要挑战。

1. 与垃圾回收（GC）一起工作。管理混合内存的一种常见方法是修改操作系统或硬件，以监视物理内存页面的访问频率，并将热点（经常访问的）数据移到DRAM中。这种方法对本地语言应用程序非常有效，其中数据保持在其分配的内存位置。然而，在托管语言中，垃圾收集器通过将对象复制到不同的物理内存页面来不断更改内存中的数据布局，从而破坏了数据与物理内存地址之间的关联。大多数大数据系统都是用这些托管语言编写的，例如Java和Scala，因为它们提供了快速的开发周期和丰富的社区支持。托管语言在托管运行时上执行，例如JVM，它采用一组复杂的内存管理技术，如垃圾回收。由于传统的垃圾回收器不了解混合内存，因此在操作系统级别分配和迁移热/冷页面很容易导致这两个不同层次的内存管理之间的干扰。
2. 与应用程序级内存子系统一起工作。现代大数据系统都包含先进的内存子系统，这些子系统在应用程序级别执行各种内存管理任务。例如，Apache Spark使用弹性分布式数据集（RDDs）作为其数据抽象。RDD是一个分布式数据结构，分区在不同的服务器上。在底层，每个RDD分区是一个Java对象数组，每个对象表示一个数据元组。RDD通常是不可变的，但可以表现出多样的生命周期行为。例如，开发人员可以显式将RDD持久保存在内存中以进行记忆或容错。这样的RDD具有长寿命，而存储中间结果的RDD具有短寿命。

RDD可以处于多种存储级别之一（例如内存、磁盘、未物化等）。Spark还允许开发人员使用注释指定RDD应该在何处分配，例如在托管堆或本地内存中。在本地分配的对象不受GC的影响，从而提高了效率。然而，诸如洗牌、连接、映射或减少等数据处理任务是在托管堆上执行的。基于本地内存的RDD除非首先移入堆中，否则无法直接处理。因此，RDD在何处分配取决于何时以及如何处理它。例如，经常访问的RDD应该放置在DRAM中，而基于本地内存的RDD不太可能频繁使用，将其放置在NVM中是可取的。显然，有效地使用混合内存需要在这些正交的数据放置策略之间进行适当的协调，即堆、本地内存或磁盘与NVM或DRAM。

另一个例子是QuickCached，它是基于QuickServer [3]的Memcached服务器的纯Java实现。具体而言，它是一个用于存储来自数据库调用、API调用或页面渲染结果的任意数据块（字符串、对象）的内存中键值存储。

QuickCached利用ConcurrentHashMap和SoftReference来管理其内存，即使用ConcurrentHashMap存储键值数据，使用SoftReference在内存需求增加时由垃圾收集器自动清除数据。

3. 现状总结：在支持大数据处理的混合内存方面，关键挑战在于如何开发运行时系统技术，使内存分配/迁移决策与应用程序中实际数据使用方式相匹配。尽管诸如Espresso和Write Rationing等技术支持托管程序使用NVM，但它们都不是为大数据处理而设计的，大数据处理的数据使用方式与常规非数据密集型Java应用程序显著不同。

   例如，Espresso定义了一种新的编程模型，开发人员可以使用它来在持久内存中分配对象。然而，现实中的开发人员可能不愿意完全使用这种新模型从头开始重新实现他们的系统。Shoaib等人引入了Write Rationing GC，该GC将经历大量/小量写入的对象移入DRAM/NVM以延长NVM的寿命。Write Rationing是使用GC基于访问模式迁移对象的先驱工作。然而，大数据系统大量使用不可变数据集——例如，在Spark中，大多数RDD都是不可变的。将所有不可变RDD放入NVM可能会产生较大的开销，因为这些RDD中的许多经常被读取，而NVM读取速度比DRAM读取慢2–4倍。

贡献

我们的分析

我们分析了两个代表性的大数据系统，即用于数据处理的Spark和用于数据存储的QuickCached，我们观察到即使它们表现出多样化的内存行为，我们仍有机会在JVM中共享共同的内存管理策略。我们的观察如下：

- Spark应用具有两个独特的特征，这些特征可以极大地帮助混合内存管理。首先，它们执行批量对象创建，数据对象呈现强烈的时代行为和清晰的访问模式。例如，Spark开发人员使用RDD进行编程，每个RDD包含具有完全相同的访问/生命周期模式的对象。在运行时利用这些模式将使大数据应用更容易享受混合内存的好处。

  其次，数据访问和生命周期模式在用户程序中通常是静态可观察的。例如，在Spark中，RDD是一种粗粒度的数据抽象，不同RDD的访问模式通常可以从它们在程序中的创建和使用方式中推断出（第2节）。
- QuickCached具有一个可以帮助混合内存管理的独特特征。它使用一个巨大的哈希表来存储键值对，当处理每个查询请求时，它会创建大量临时对象，这些对象在很短的时间内会被频繁访问。因此，经常访问数据的生命周期很短，而寿命较长的数据很少被访问。

因此，与常规的非数据密集型应用程序需要使用性能分析来了解各个对象的访问模式不同，我们可以为大数据应用程序开发一个简单的静态分析，以推断每个粗粒度数据集的访问模式，其中所有对象共享相同的模式。这一观察与之前的工作（例如Facade或Yak）相吻合，这些工作需要简单的注释来指定时代，以执行大数据系统的高效垃圾收集。静态分析不会产生任何运行时开销，但它可以生成足够精确的数据访问信息，以便运行时系统执行有效的分配和迁移。

Panthera

根据我们在大数据应用方面的丰富经验，我们提出了Panthera。Panthera根据应用程序的语义将一堆数据对象划分为几个数据集，并通过轻量级静态程序分析和动态数据使用监视推断粗粒度的数据使用行为。Panthera利用垃圾收集来在DRAM和NVM之间迁移数据，几乎不会引起运行时开销。

在本文中，我们选择了两个大数据处理框架。首先，我们关注Apache Spark，因为它是业界广泛部署的事实上的数据并行框架。Spark托管一系列应用，涵盖机器学习、图分析、流处理等多个领域，因此构建一个专门的运行时系统是值得的，可以立即为所有运行在其之上的应用程序带来好处。此外，为了展示我们方法的通用性，Panthera还构建在QuickCached之上，这是Memcached的Java实现，第4节提供了对Panthera适用性的详细讨论。

Panthera通过两个主要创新来增强JVM和Spark/QuickCached。首先，基于这样一个观察：在大数据应用程序中，访问模式可以在静态情况下被识别，我们分别为Spark和QuickCached开发了两个静态分析器（第3节）。特别是，Spark分析器分析Spark程序以推断每个RDD变量的内存标记（即NVM或DRAM），这基于变量的位置以及在程序中使用的方式。而QuickCached分析器分析QuickCached的源代码以识别巨大的全局哈希表，然后推断其相应的内存标记。这些标记指示对象应该在哪个内存中分配。

其次，我们开发了一种新的语义感知和物理内存感知的分代GC（第4节）。我们的静态分析会修改Spark程序和QuickCached，以将推断的内存标记传递到运行时系统，该系统使用这些标记做出分配/迁移决策。由于我们的GC基于OpenJDK中的高性能分代GC，Panthera的堆有两个空间，分别表示年轻代和老年代。我们将整个年轻代放在DRAM中，同时将老年代分割为一个小的DRAM组件和一个大的NVM组件。这个设计的驱动思想基于我们在代表性Spark执行和QuickCached对象的生命周期和访问模式上所做的一系列关键观察（在第2节中详细讨论）。

- 大多数对象最初是在年轻代中分配的。由于它们在初始化过程中经常被访问，将它们放在DRAM中能够快速访问。
- 在Spark中，具有长寿命的对象大致可以分为两类：(1) 长寿命的RDD，在数据转换期间经常被访问（例如，对于迭代算法而言被缓存），(2) 长寿命的RDD，主要是为了容错而缓存。第一类RDD应该放在老年代的DRAM组件中，因为它们寿命长，而DRAM对于经常访问它们提供了理想的性能。而第二类RDD应该放在老年代的NVM组件中，因为它们很少被访问，因此NVM的大访问延迟对整体性能的影响相对较小。
- 对于Spark程序，还有存储临时中间结果的短寿命RDD。这些RDD迅速死亡并在年轻代中被回收，导致对该区域的频繁访问。这是我们将年轻代放在DRAM内的另一个原因。
- 对于QuickCached，只有一个长寿命对象，即用于存储键值数据的ConcurrentHashMap。在哈希表中，仅有一小部分在特定请求中经常被访问，这将在运行时被识别，几乎没有开销。因此，ConcurrentHashMap应该放在老年代的NVM组件中，除非是被识别的经常访问的部分。

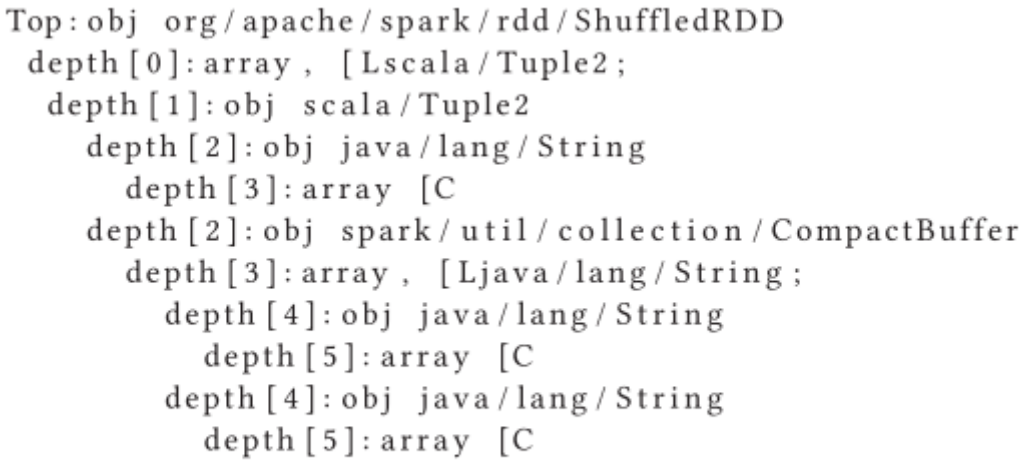


Fig. 1. The heap structure of an example RDD.

对于QuickCached，会创建一些临时对象来处理特定请求。这些对象被分配在年轻代中，并且应该放在DRAM中以实现快速访问。

基于这些观察，我们修改了小型和大型GC，根据它们的RDD类型和我们的静态分析推断的语义信息，将数据对象分配和迁移到最适合其生命周期和访问模式的空间。我们的运行时系统还监视对RDD对象执行的转换，以在运行时对RDD的访问模式进行（重新）评估。即使静态分析不能准确预测RDD的访问模式，并且RDD在不理想的空间中分配，Panthera仍然可以使用大型GC将RDD从一个空间迁移到另一个空间。

结果

我们已经用Spark应用程序对Panthera进行了广泛的评估，包括图计算(GraphX)、机器学习(MLlib)和其他迭代内存计算应用程序(表4)，以及使用Yahoo!云服务基准(YCSB)[22]。各种堆大小和DRAM比例的结果表明，Panthera有效地利用了混合内存——总体而言，Panthera增强的JVM减少了22%-34%的内存能量，而QuickCached的执行时间开销仅为1% - 9%，而Spark的平均执行时间开销仅为不到1%，减少了32%-53%的内存能量，而Write Rationing[11]将只读RDD对象移动到NVM会产生41%的时间开销。

背景与动机

本节提供了Apache Spark[6]和QuickCached[3]的必要背景知识，并提供了一些鼓舞人心的示例，说明了Spark程序中的访问模式。

Spark基础

Spark是一个支持无环数据流和内存计算的数据并行系统。Spark主要使用的数据表示是弹性分布式数据集（RDD）[91]，它表示一个只读的元组集合。RDD是一个在集群中分区的分布式内存抽象。每个分区是相同类型的数据项数组。每个节点维护一个RDD分区，它实际上是一个多层次的Java数据结构——一个顶层的RDD对象引用一个Java数组，而这个数组又引用一组元组对象，比如键值对。图1展示了一个示例RDD的堆结构，其中每个元素是一个字符串（键）和一个紧凑缓冲区（值）的对。

Spark的管道由一系列对RDD进行的转换和操作组成。转换从一组现有的RDD生成一个新的RDD；例如，map、reduce或join都是转换的例子。操作是从RDD计算统计信息的函数，比如聚合操作。Spark利用懒惰评估来提高效率，也就是说，一个转换可能不会在稍后执行对生成的RDD进行操作之前进行评估。在数据处理开始之前，首先从转换中提取RDD之间的依赖关系，形成一个血统图，该图可用于进行懒惰评估和在节点故障时重新计算RDD。

采用懒惰评估，一个转换只创建一个（顶级）RDD对象，而不实例化RDD（即，其内部数组和实际数据元组创建的点）。当血统很长或分支出时，重新计算所有RDD会耗费大量时间，因此，Spark允许开发人员将某些RDD缓存在内存中（通过使用API persist）。开发人员可以为持久化的RDD指定存储级别，例如在内存中或磁盘上，以序列化或反序列化形式等。未明确持久化的RDD是临时RDD，在不再使用时将被垃圾回收，而持久化的RDD则会被实例化并永远不会被回收。

Spark调度程序检查血统图以构建执行的阶段的DAG。基于血统（转换）的依赖被分类为“狭窄”和“宽广”。如果父RDD的每个分区最多被子RDD的一个分区使用，则存在从父RDD到子RDD的狭窄依赖。相反，如果父RDD的每个分区可能被多个子分区使用，则存在宽依赖。区分这两种依赖关系使Spark能够确定是否需要进行shuffle。例如，对于狭窄依赖，不需要shuffle，而对于宽依赖则需要。

Spark管道根据shuffle（因此是宽依赖）分为一组阶段。每个阶段以写入RDD到磁盘的shuffle结束，下一个阶段从磁盘文件中读取数据开始。展现狭窄依赖的转换被分组到同一阶段中并并行执行。

RDD特性

在底层，RDD是一个由Java对象数组组成的结构，由JVM中的语义无关的GC进行管理。RDD通常表现出可预测的生命周期和内存访问模式。我们的目标是将这些模式传递给GC，GC可以利用这样的语义信息进行有效的数据放置。我们提供一个具体的例子来说明这些模式。

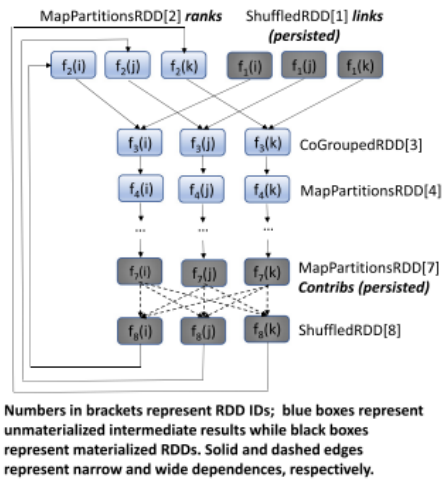
图2(a)展示了PageRank [19]的Spark程序，这是一种被广泛用于搜索引擎对网页进行排名的图算法。该程序迭代地计算每个顶点的排名，基于其入边的贡献。从其源代码可以看到三个RDD：links表示输入图的边，contribs包含每个顶点入边的贡献，ranks将每个顶点映射到其页面排名。links是从输入计算得到的静态映射，而contribs和ranks在循环的每次迭代中重新计算。除了在程序中可见的这三个由开发人员定义的RDD之外，Spark还生成许多不可见的RDD来存储执行期间的中间结果。一种特殊类型的中间RDD是ShuffledRDD。例子中循环的每次迭代形成一个阶段，以一个shuffle结束，将打乱的数据写入不同的磁盘文件。在下一个阶段的开始，Spark为该阶段创建一个ShuffledRDD作为输入。与其他永远不会实例化的中间RDD不同，ShuffledRDD会立即实例化，因为它们包含新鲜读取的磁盘文件中的数据。但是，由于它们没有被持久化，它们将在阶段完成时被回收。

总的来说，(1) 持久化的RDD在调用persist方法时被实例化，(2) 未持久化的RDD在它们是ShuffleRDDs或者对它们调用操作时才会被实例化。

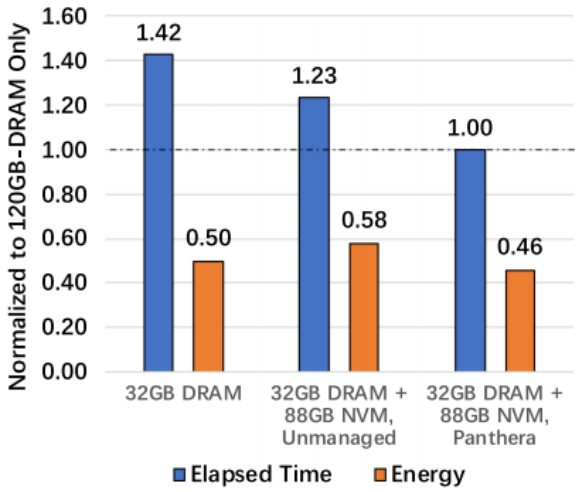
```scala
var lines = ctx.textFile(args[0], slices)
var links = lines.map{s=>
  var parts = s.split("\\s+")
  (parts(0), parts(1))
}.distinct().groupByKey()
.persist(StorageLevel.MEMORY_ONLY)

var ranks = links.mapValues(v => 1.0)
for(i <- 1 to iters){
  var contribs = links.join(ranks).values.flatMap{
    case(urls, rank) =>
      val size = urls.size
      urls.map(url=>(url, rank/size))
      .persist(StorageLevel
                .MEMORY_AND_DISK_SER)
  }
  ranks = contribs.reduceByKey(_ + _).
    mapValues(0.15 + 0.85 * _)
}
ranks.count()
```

(a) PageRank program.



(b) Transformations within a stage.



(c) Results of DRAM-only and DRAM+NVM, managed by the OS and by Panthera.

Fig. 2. Characteristics of RDDs in Spark PageRank.

例子

图2(b)展示了在一个阶段（即迭代）中存在的一组RDD及其依赖关系。假设每个RDD有三个分区（在三个节点上）。虚线表示宽依赖（即shuffle），是因为第17行的reduce。每次迭代总共生成八个RDD。ShuffledRDD[8]是从第17行的reduce产生的，通过map转换变成ranks。ranks与links进行联接形成CoGroupedRDD[3]，然后通过四个连续的map函数（即f4 – f7）进行处理，最终产生contribs。对于未实例化的（蓝色）RDD，转换序列（例如f4。...。f7）以流式方式通过迭代器应用于源RDD的每个记录，生成最终记录。

对于links和contribs，开发人员调用persist方法来实例化这些RDD。存储级别表示links在整个执行过程中都被缓存在内存中（因为它在每次迭代中都被使用），而在每次迭代中生成的contribs会保留在内存中，但在内存压力下会被序列化到磁盘上。ranks没有被明确持久化。因此，在执行到达第20行并调用RDD对象的count操作之前，它不会被实例化。

这些不同RDD的生命周期模式可以分为两类。未持久化的中间RDD的寿命较短，因为它们的数据对象仅在流水线执行期间生成。持久化的RDD的寿命较长，并保留在内存/磁盘中，直到执行结束。然而，它们的访问模式更加多样化。在中间RDD中的对象在流式处理期间最多被访问一次。在持久化的RDD中，对象可能表现出不同类型的行为。对于像links这样在每次迭代中都被使用的RDD，它们的对象经常被访问。相反，像contribs这样的RDD主要是为了加速从故障中恢复，因此，在生成后它们的对象很少被使用。

设计选择

DRAM和NVM的不同特性使它们适用于不同类型的数据集。DRAM具有低容量和快速访问速度，而NVM具有大容量但速度较慢。因此，DRAM是存储小型、频繁访问数据集的良好选择，而大型、不经常访问的数据集自然适用于NVM。不同RDD的寿命和访问模式的清晰区分使它们很容易被放置到适合其行为的不同内存中。例如，中间（蓝色）RDD永远不会被实例化。它们的对象在流式处理期间单独创建，然后被GC快速回收。这些对象被分配在young generation中，最终会在那里死亡。因此，young generation中的内存被这些短寿命对象频繁重用，导致该内存区域的读/写频率非常高。这促使我们选择将young generation放在DRAM中，这与先前研究的结论一致[11, 76]。

相反，持久化RDD的所有数据对象都是在同一时间创建的，因此需要大量的存储空间。由于它们无限期保持活动状态，它们应该直接分配在old generation中。持久RDD的一类包括那些经常访问的RDD，例如links；它们需要放置在DRAM中。另一类包括那些很少访问并且被缓存用于容错的RDD，例如contribs，这些RDD应该放置在NVM中。这种行为差异促使我们选择将old generation拆分为DRAM和NVM两个组件。

我们在一个具有128GB内存的系统上执行我们建议的操作，使用基于Spark的PageRank作为基准。对于这个实验，我们为Spark分配了120GB内存，并保留了8GB内存用于操作系统和其他服务。对于120GB的Spark内存，有32GB是DRAM，其余的是NVM。（我们在评估部分变化了DRAM比例。）图2(c)显示了性能和能耗相对于具有120GB DRAM的系统进行了归一化。与仅使用32GB DRAM相比，向系统添加88GB NVM提供了适度的性能提升（15%），但导致了16%更高的能耗，没有适当地在DRAM和NVM之间进行数据放置（见未管理部分，第7.2节）。应用了Panthera后，RDD links和contribs分别放置在DRAM和NVM中。通过在DRAM和NVM之间仔细放置数据，我们发现（1）性能相对于仅使用32GB DRAM增加了42%，并达到了使用120GB DRAM的性能水平；（2）能耗比仅使用32GB DRAM少了9%，比使用120GB DRAM少了54%。我们得出结论，在DRAM和NVM之间进行仔细的数据放置可以提供大型DRAM系统的性能，同时保持能耗在小型DRAM系统的水平。

QuickCached基础知识

QuickCached是基于QuickServer的Memcached服务器的纯Java实现，它充当一个用于存储来自数据库调用、API调用或页面渲染结果的小块任意数据（字符串、对象）的内存中键值存储。

QuickCached支持不同的后端来组织和管理键值数据，其默认后端利用ConcurrentHashMap和SoftReference，即ConcurrentHashMap用于存储键值数据，SoftReference用于根据内存需求在垃圾收集器自主判断下自动清除数据 [3]。在处理每个查询请求时，QuickCached会创建大量频繁访问的临时对象，这些对象在当前查询请求完成时将被销毁。因此，这些频繁访问的数据的寿命很短。相比之下，ConcurrentHashMap提供全生命周期的服务并具有较长的寿命，然而，在处理一个查询请求时，ConcurrentHashMap中只有一小部分数据会被频繁访问。因此，我们有机会在静态情况下识别ConcurrentHashMap的数据结构，并在运行时识别其频繁访问的对象。相应地，ConcurrentHashMap应该放置在old generation的NVM组件中，而已识别的频繁访问对象应该放置在old generation的DRAM组件中。

静态推断内存标签

根据我们对内存访问模式的观察，我们开发了一个简单的静态分析，提取了为高效数据放置所需的语义信息。对于Spark，RDD的访问模式通常可以从使用它们的程序中识别出来。我们的分析会自动推断每个在程序中可见的持久化RDD是否应该分配到DRAM或NVM。然后，将这些信息传递给运行时系统，以进行适当的数据分配。对于QuickCached，我们的分析以用户注释的源代码为输入，并自动生成运行时数据放置的代码，具体细节将在下文讨论。

spark 分析

静态分析。在Spark程序中，开发人员可以在RDD上调用具有特定存储级别的持久化来实现RDD，如图2所示。我们利用存储级别来进一步确定持久化RDD应该放在DRAM还是NVM中。特别是，Panthera静态分析程序来推断每个持久化调用的内存标签(即DRAM或NVM)。除了OFF_HEAP和DISK_ONLY之外，现有的十个存储级别(例如MEMORY_ONLY)中的每一个都扩展为两个子级别，分别用NVM和DRAM进行注释(例如MEMORY_ONLY_DRAM和MEMORY_ONLY_NVM)。OFF_HEAP被直接转换为OFF_HEAP_NVM，因为很少使用放在本机内存中的rdd，而DISK_ONLY不携带任何内存标记。

我们的静态分析是基于程序中声明的每个RDD变量的使用定义信息以及变量所在的循环的def-use信息。我们的主要洞察是，如果变量在计算循环的每次迭代中都被定义，那么变量表示的大多数RDD实例通常不会频繁使用。这是因为Spark RDD通常是不可变的，因此RDD变量的每个定义都会在运行时创建一个新的RDD实例，而旧的RDD实例则被缓存并未使用。因此，我们对变量进行“NVM”标记，指示运行时系统将这些RDD放在NVM中。例如，图2(a)中的contribs变量就是一个例子，它在循环的每次迭代中都被定义，尽管变量在每次迭代中也被使用，但使用指的是在上一次迭代中创建的最新RDD实例，而在所有其他过去的迭代中创建的实例则未被使用。

相反，如果一个变量在循环中仅被使用（即从未被定义），比如links，我们为其创建一个“DRAM”标记，因为只存在一个RDD实例，并且会被重复使用。Panthera不仅分析明确调用persist的RDD变量，还分析调用actions的RDD变量，比如图2(a)中的ranks变量。对于RDD变量（比如v）推断的标签会在每个RDD实例的实体化点通过由Panthera JVM提供的辅助（本地）方法的自动插装调用中传递到运行时系统。我们利用追踪GC来将这个标签从RDD对象传播到RDD中包含的每个数据对象——当GC运行时，它将具有相同标签的对象移动到同一个（DRAM或NVM）区域（参见第4节）。

我们的静态分析还需要额外考虑一个约束，即循环的位置相对于RDD的实体化点的位置。我们只分析在实体化点之前或之中的循环。否则，变量在循环中是使用还是定义都无关紧要，因为RDD尚未被实体化。例如，尽管ranks变量在从第17行开始的循环中被定义，但它直到第20行之后循环结束后才会被实体化。因此，它在循环中的行为不会影响其内存标签，实际上应该取决于其在第20行之后的循环中的使用定义情况。

如果程序中没有循环，则该程序只有一个迭代，所有RDD都会被标记为“NVM”，因为它们都不会被重复访问。如果存在多个循环要考虑的RDD变量，我们将其标记为“DRAM”，如果存在一个循环，其中该变量仅被使用，并且该循环在RDD的实体化点之后或包含在其中。否则，该变量将获得“NVM”标记。如果在分析结束时所有持久化的RDD都获得了“NVM”标记，我们将所有RDD的标签更改为“DRAM” - 目标是首先将RDD放入DRAM以充分利用DRAM。一旦DRAM容量耗尽，剩余的RDD，包括那些具有“DRAM”标签的RDD，将被放入NVM中。

请注意，我们的分析仅推断程序中显式声明的RDD变量的标签。在执行过程中生成的中间RDD不会被实体化，因此不会从我们的分析中获得内存标签。我们将在第4节中讨论如何处理它们。

RDD变量的内存标签是其访问模式的静态近似，可能无法反映在运行时由该变量表示的所有RDD实例的行为。然而，用于数据处理的用户代码通常具有简单的批处理逻辑。因此，从我们的分析中推断出的静态信息通常足以帮助运行时系统为RDD做出准确的放置决策。如果静态推断的标签没有准确捕捉到RDD的访问信息，Panthera具有在NVM和DRAM之间（在老年代内部）基于它们的访问频率移动RDD的能

力，当发生全堆GC时。动态数据迁移频率是静态分析准确性的一个很好的指标。第4节提供了有关此机制的详细讨论，第7.5节评估了静态分析的准确性和动态迁移的开销。

处理ShuffledRDD。回顾第2节，除了显式调用persist的RDD之外，还有一些从磁盘文件创建的ShuffledRDD也会被实体化，这些RDD通常是一个阶段的输入，但在程序代码中不可见。这里的挑战是确定它们的放置位置。我们的洞察是它们的放置应该取决于在相同阶段从它们转换而来（即依赖于）的其他已实体化的RDD。

例如，在图2(b)中，阶段的输入是两组ShuffledRDD：[1]和[8]。ShuffledRDD[1]是由links表示的RDD，我们的静态分析已经为它推断出了“DRAM”的标签。ShuffledRDD[8]是前一阶段的减少的结果。由于ShuffledRDD[8]追溯地生成了MapPartitionRDD[7]（由contribs表示），而MapPartitionRDD[7]通过我们的静态分析已经有了“NVM”的内存标签，因此我们也将ShuffledRDD[8]标记为“NVM”。主要原因是属于同一阶段的RDD可能出于优化目的共享许多数据对象。例如，只更改RDD A中键值对的

从图中最低的具体化RDD开始，它从我们的分析中收到了一个标签。

冲突可能在传播过程中发生——在向后遍历过程中遇到的RDD可能有一个与正在传播的标记不同的现有标记。为了解决冲突，我们定义了以下优先顺序:DRAM > NVM，这意味着在发生冲突时，结果标签总是DRAM。这是因为我们的目标是最小化nvm引起的开销;推断带有“DRAM”标签的rdd将经常被使用，将它们放在NVM中会导致很大的性能下降。

```
1:  G ← the lineage graph
2:  while there are materialized RDDs not being selected do
3:      V ← the lowest materialized RDD which has not been selected
4:      for all P ∈ parent(V) do
5:          if V.tag = DRAM then
6:              P.tag ← DRAM
7:          else if P.tag ≠ DRAM then
8:              P.tag ← V.tag
9:          end if
10:     end for
11: end while
12: end
```

Fig. 3. Algorithm to assign same tags to RDDs which share date objects.

QuickCached分析

对于QuickCached，核心对象是存储对象，即哈希表ConcurrentHashMap。QuickCached分析器要求用户使用以下语法对此对象进行注释@CoreHashObject，例如：

```
@CoreHashObject
ConcurrentHashMap hashTable;
```

根据观察，在处理一个查询请求时，哈希表的只有很小一部分会经常访问，因此该注释将指导Panthera将哈希表放置在NVM中，同时只保留在DRAM中经常访问的部分。特别是，QuickCached分析器识别带注释的哈希表ConcurrentHashMap并在静态上推断其内存标签为“NVM”。

然而，与Spark应用程序不同，ConcurrentHashMap中的经常访问数据无法在静态上识别，因为它由运行时的传入请求决定。因此，静态分析对于为存储在ConcurrentHashMap中的对象推断有意义的标签是低效的。为了解决这个问题，QuickCached分析器引入了一种动态机制来在运行时区分经常访问的数据并在运行时将其标记为“DRAM”。由于真实世界的工作负载中数据访问是高度倾斜的[16]，因此可以通过监视运行时的数据访问模式来识别经常访问的数据。具体而言，分析器在访问带注释的ConcurrentHashMap的get方法的调用点自动插入了一些插装代码。插装代码如图4所示，采用简单的LRU策略将最近访问的值数据标记为“DRAM”。

在Spark中，RDD是一个抽象层，低级别是一组Java对象的数组，这种语义有助于在Panthera中进行内存标签分析和传递。然而，QuickCached缺乏这种RDD抽象，因此为了与Spark共享相同的内存标签传递机制，我们合成了一个RDD，用来包装需要由Panthera管理的对象。

```
1  rdd_indicator("DRAM");
2  Object [] AggDramValues = new Object [MAX_OBJECTS_NUMBER];
3  long index = 0L;
4  ...
5  public Object get(...){
6    ...
7    /* retrieve value from hash table */
8    Object value = hashTable[key];
9    /* aggregate the most recently accessed data */
10   AggDramValues[(index++) % MAX_OBJECTS_NUMBER] = valule;
11   ...
12 }
```

Fig. 4. Example of QuickCached analyzers instrumentation codes.

特别地，检测代码的工作方式如下。首先，rdd_indicator(“DRAM”)((第1行))在QuickCached中声明了合成的RDD，它将表现为Spark中的RDD。其次，我们分配一个固定大小的辅助对象数组AggDramValues，它将被包装在合成的RDD中，将最近访问的值数据聚合在一起(第2行和第3行)。最后，最近访问的元素将在访问时被复制到AggDramValues(第10行)。

通过合成的RDD, Panthera提供了一个统一的内存标签传递机制，可以同时支持Spark和QuickCached，这将在4.2节中讨论。

Panthera垃圾收集器

虽然我们的静态分析（第3节）确定了RDD应该被分配到哪里，但这些信息必须传达给运行时系统，而运行时系统只识别对象，而不是RDDs。因此，我们的目标是开发一种新的垃圾回收器，当放置/移动数据对象时，它能够意识到（1）关于这些RDDs应该被放置在哪里（DRAM还是NVM）的高级语义，以及（2）关于这些对象所属的RDDs的低级信息。

我们已经在OpenJDK 8（版本jdk8u76-b02）[8]中实现了我们的新收集算法。具体而言，我们修改了对象分配器、解释器、两个JIT编译器（C1和Opto）以及Parallel Scavenge垃圾回收器。

设计概念

堆的设计。Panthera GC基于并行清除收集器，这是OpenJDK8中的默认GC。收集器将堆划分为年轻代和老代。如前面第1节所讨论的，Panthera将年轻一代放在DRAM中，并将老一代拆分为DRAM组件和NVM组件。堆外本机内存完全放在NVM中。我们从每个对象的头中保留两个未使用的位，称为MEMORY_BITS，以指示该对象应该分配到DRAM(01)还是NVM(10)中。这些位的缺省值是00 -没有收到标签的对象有这个缺省值。如果他们活得足够长，他们将被提升到老一代的NVM部分。图5说明了堆结构和分配策略。

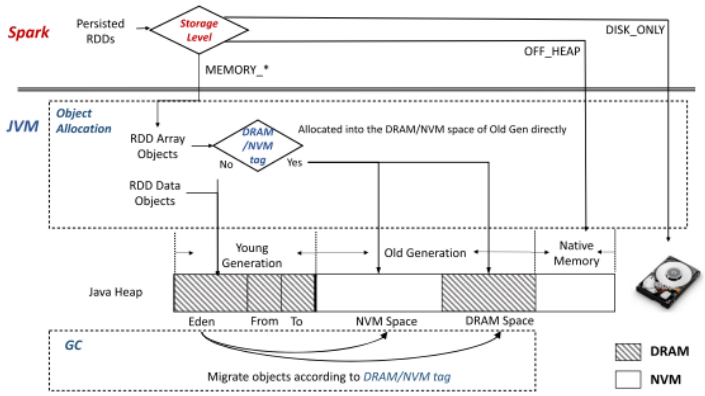


Fig. 5. The Panthera heap and allocation policies. Here RDD array objects refer to RDDs' backbone arrays while data objects refer to other non-array objects in an RDD structure.

分配策略。如第3节所讨论的，每个被实体化的RDD都带有一个来自我们的静态分析或基于血统的标签传播的内存标签。然而，在低级别上，一个RDD是一个对象的结构，如图1所示，这些对象在执行的不同点创建。我们的目标是将属于同一个逻辑RDD的所有对象——包括顶层对象、数组对象、元组对象和从元组可达的其他对象——放在RDD的内存标签建议的空间中，因为这些对象可能具有相同的访问模式和生命周期。

然而，这是相当具有挑战性的——我们的静态分析为用户程序中的每个顶级RDD对象（其类型是org.apache.spark.rdd.RDD的子类型）推断一个内存标签，我们无法仅通过分析用户程序就知道其他对象属于该RDD。在静态地识别属于逻辑数据结构的对象需要精确的上下文敏感静态分析，涉及用户和系统代码，由于Spark极大的代码库和静态分析的可伸缩性问题，这很难实现。

我们解决这个问题的思路是，我们不试图直接将RDD的所有对象分配到建议的空间（假设为S），而是仅在其创建时将数组对象分配到S。这样做要容易得多——Panthera在用户程序中的每个实体化点（例如，在调用persist或Spark操作之前）进行插装，将标签传递给运行时系统，而无需分析Spark系统代码。由于数组是在实体化时创建的，运行时系统可以使用标签来确定在哪里放置它。RDD中的所有其他对象不会因为在找到它们的分配位置时的困难而立即分配到S中。它们而是在young generation中分配。随后，我们使用GC在进行追踪时将这些对象移动到S中。

首先将数组对象分配到S的另一个重要原因是，数组对象通常比顶层和元组对象大得多。直接将其分配到其所属的空间要比在其他地方分配并稍后移动效率高得多。

表1显示了我们对RDD中不同类型对象的分配策略。对于标签为"DRAM"的RDD，数组对象将直接分配到old generation的DRAM组件中，如果有足够的空间。否则，它们必须分配到NVM组件中。对于标签为"NVM"的RDD，数组对象将直接分配到NVM组件中。没有标签的中间RDD都分配在young generation中（DRAM）。它们中的大多数最终死在那里，永远不会晋升，而一小部分最终变得足够老会被提升到old generation的NVM空间中。顶级RDD对象和数据元组对象，如前面讨论的，都分配在young generation中，并由GC稍后移动到包含它们相应数组的空间中。

Table 1. Panthera's Allocation Policies

Tag	Obj Type	Initial Space	Final Space
DRAM	RDD Top	Young Gen.	DRAM of Old Gen.
	RDD Array	DRAM of Old Gen.	DRAM of Old Gen.
	Data Objs	Young Gen.	DRAM of Old Gen.
NVM	RDD Top	Young Gen.	NVM of Old Gen.
	RDD Array	NVM of Old Gen.	NVM of Old Gen.
	Data Objs	Young Gen.	NVM of Old Gen.
NONE	RDD Top	Young Gen.	Young Gen. or NVM of Old Gen.
	RDD Array	Young Gen.	Young Gen. or NVM of Old Gen.
	Data Objs	Young Gen.	Young Gen. or NVM of Old Gen.

实现与优化

本小节描述我们的实现技术和各种优化。

通过标签

在每个实体化点（即调用persist或Spark操作之前），我们的分析插入了对一个本地方法rdd_indicator(rdd, tag)的调用，其中RDD的顶层对象（rdd）和推断的内存标签（tag）作为参数。该方法首先将线程本地状态变量设置为DRAM或NVM，根据标签，通知当前线程即将为RDD分配一个大数组。接下来，rdd_indicator根据标签设置顶层对象rdd的MEMORY_BITS。无论它目前在哪里，这个顶层对象最终都将由GC移动到与标签相对应的空间。

然后，线程转入"等待"状态，等待这个大数组。在这个状态下，第一个长度超过用户定义阈值的数组的分配请求（在我们的实验中为100万）被识别为RDD数组。然后，Panthera直接将数组分配到标签指示的空间。为了实现这一点，我们修改了快速分配路径（JIT编译器生成的汇编代码）和慢路径（在C++中实现的函数）。

在此分配后，状态变量被重置，线程退出等待状态。如果标签为空，则该数组将在young generation中分配，最好通过线程本地分配缓冲区（TLAB），并且顶层对象的MEMORY_BITS保持为默认值（00）。

对象迁移

在如何移动对象方面存在两个主要挑战：跨代迁移和对象压缩。由于Panthera依赖分代垃圾收集，当JVM无法为新对象分配空间时，会触发小型GC。在年轻代中经过几次小型GC而幸存的对象被视为"长寿"，并移动到旧代。当旧代内存满时，会触发大型GC。我们利用这个机会将属于同一逻辑RDD的对象一起移动，正如之前讨论的，这些对象最初可能未分配在相同的空间中。

小型GC： 为了实现这一点，我们修改了Panthera构建在其上的Parallel Scavenge GC中的小型收集算法。现有的小型GC包含三个任务：根任务，从根（例如，堆栈和全局变量）执行对象跟踪；老到年轻任务，扫描从旧代对象到年轻代对象的引用，以识别（直接或传递地）可达的对象；偷取任务，用于负载平衡的工作窃取。

为了支持我们的对象迁移，我们将老到年轻任务分为DRAM到年轻任务和NVM到年轻任务，分别查找应该移动到旧代DRAM和NVM部分的对象。

对于这两个任务，我们修改了跟踪算法以传播标签——例如，从基于DRAM的RDD数组（带有标签"DRAM"）到元组对象（在年轻代中）的引用扫描将标签传播到元组对象（通过设置其MEMORY_BITS）。因此，在跟踪完成时，从数组可达的所有对象都将其MEMORY_BITS设置为与数组相同的值。在原始GC算法中，对象在从年轻代晋升到旧代之前需要经历几次小型GC（在本文中，我们使用阈值为15）。然而，在Panthera中，我们将在跟踪中其MEMORY_BITS设置为01（10）的对象立即移动到旧代DRAM（NVM）空间，我们将这种机制称为急切晋升。在跟踪中其MEMORY_BITS没有设置的

对象，00，属于中间RDD或与任何RDD无关的控制对象。这些对象的迁移遵循原始算法，即仅在它们经过几次小型GC后才会移动。在急切晋升期间，如果旧代DRAM空间不足，Panthera将相应的对象放入NVM，并在执行期间让大型GC调整数据布局。

此外，我们还需要将RDD顶级对象移动到旧代的适当部分。

这些顶级对象的MEMORY_BITS是通过在它们的实体化点调用rdd_indicator时进行插装设置的，因为这些对象是由堆栈变量直接引用的。我们修改了根任务算法以识别带有设置MEMORY_BITS的对象。这些RDD顶级对象也将通过小型GC移动到旧代（DRAM（01）或NVM（10）空间）。

GC主要部分

当主GC运行时，它通过将对象移动到一起(在老一代中)来执行内存压缩，以减少碎片并改善局部性。我们修改了主GC，以确保压缩不会跨DRAM和NVM之间的边界发生。此外，当主GC执行全堆扫描时，Panthera会根据RDD的运行时访问频率重新评估每个RDD数组对象的实际位置。

这个频率是通过使用工具来计算在这个RDD对象上调用一个方法(例如map或reduce)的次数来测量的。rdd按照访问频率排序。如果最频繁访问的rdd被放置在NVM中，它们将被迁移到DRAM。如果没有足够的DRAM空间用于这些高级rdd, Panthera将从DRAM中驱逐访问频率较低的rdd。

我们维护一个哈希表，将每个RDD对象映射到对该对象的调用次数。

我们的静态分析在每个这样的调用站点插入一个JNI (Java本机接口)调用，该调用调用一个本机JVM方法来增加RDD对象的调用频率。经常(不经常)访问的数组对象从NVM (DRAM)空间移动到老一代内的DRAM (NVM)空间，并且从这些数组可访问的所有对象也被移动。它们的MEMORY_BITS将相应更新。在每个主GC结束时，重置每个RDD的频率。

老一代的DRAM空间比NVM空间小得多，因此可以很快填满。当DRAM空间满时，小GC将所有对象从年轻代移动到老一代的NVM空间，而不管它们的内存标签如何。

冲突

如果可以从多个引用访问对象，并且通过它们传播不同的标记，则会发生冲突。如前所述，我们通过赋予“DRAM”比“NVM”更高的优先级来解决冲突。只要对象从任何引用中接收到“DRAM”，它就是一个DRAM对象，并将被移动到老一代的DRAM空间中。

卡片优化

在OpenJDK中，堆被分成许多卡，每个卡代表一个512字节的区域。每个对象都可以取一张或多张卡，写屏障维护一个卡表，在引用写时将某些卡标记为脏卡。卡片表可以用来在跟踪过程中有效地识别引用。例如，当a.f = b时，包含a引用的对象的卡被设置为dirty。当小型GC运行时，如果卡所表示的内存区域中包含的(老到年轻)引用的目标对象已被复制到老一代，则老到年轻的清除任务将清除卡。

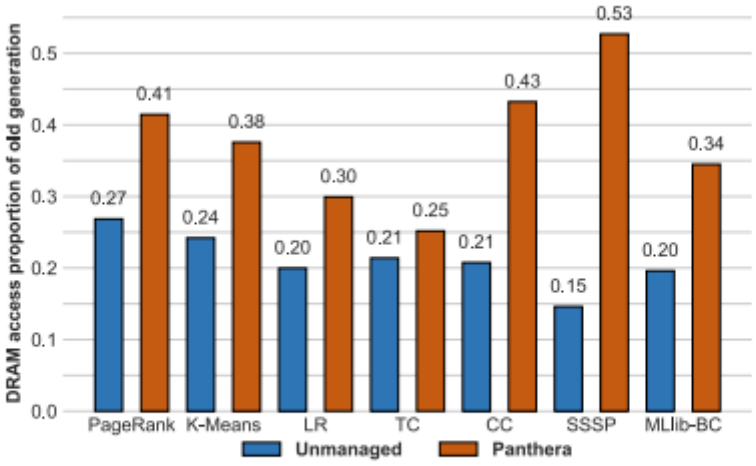


Fig. 6. DRAM access proportion in old generation.

然而，如果一张卡包含两个大数组（比如A和B），例如A在卡的中间结束，而B从卡的中间开始，那么当两个不同的GC线程扫描它们时，可能会导致显著的低效。即使A和B引用的所有对象都已从年轻代移动到老年代，卡仍然会保持脏状态，因为每个线程都无法清理由另一个线程扫描的数组的状态。这将导致每次发生大型GC之前，每次小型GC都要扫描每个数组的每个元素。

这对于大数据应用程序，特别是那些频繁使用大数组的应用，是一个严重的问题。当这些数组经常分配和释放时，共享的卡片是普遍存在的。多个线程频繁扫描这样的卡片可能会在NVM上产生大量开销，因为NVM具有更高的读取延迟和降低的带宽。我们实施了一个简单的优化，为每个RDD数组的分配添加了对齐填充，使数组的末尾与卡的末尾对齐。尽管这会导致空间浪费，但浪费的空间量很小（例如，对于每个数百兆字节的数组，浪费的空间少于512字节），而数组之间的卡片共享被完全消除，从而显著减少了GC时间。

基于性能分析的优化（PGO）

如第3节所述，Panthera背后的核心思想是静态地推断rdd的内存标签，并将它们传递给运行时系统，以指导对象迁移。为了更精确地分析对象的运行时行为，我们在本节中提出了profiling-guided optimization (PGO)。

内存访问分布

图6显示了老一代中DRAM内存访问占总内存访问的比例。与Unmanaged相比，Panthera的DRAM访问比例平均提高了16.7%，这表明Panthera的策略可以有效地提高DRAM访问比例。在本节中，我们设计了一个新的实验，以细粒度分析老一代对象的访问行为。

我们将旧代空间划分为大小为1 gb的块，并在图7中绘制每个块的访问行为，使用四个应用程序，即LR、TC、GraphX-CC和MLlib。其中横轴用Chunk ID表示所有块，纵轴表示每个块的访问比例，即访问该块的次数除以访问所有块的次数。图7中的块分为四类，(1)红色，访问频繁的块，放在DRAM中;(2)黄色，访问不频繁的块，放在DRAM中;(3)蓝色，访问频繁的块，放在NVM中;(4)绿色，访问不频繁的块，放在NVM中。红点和绿点表示正确识别和放置的块。黄色和蓝色的点表示放置错误的块。

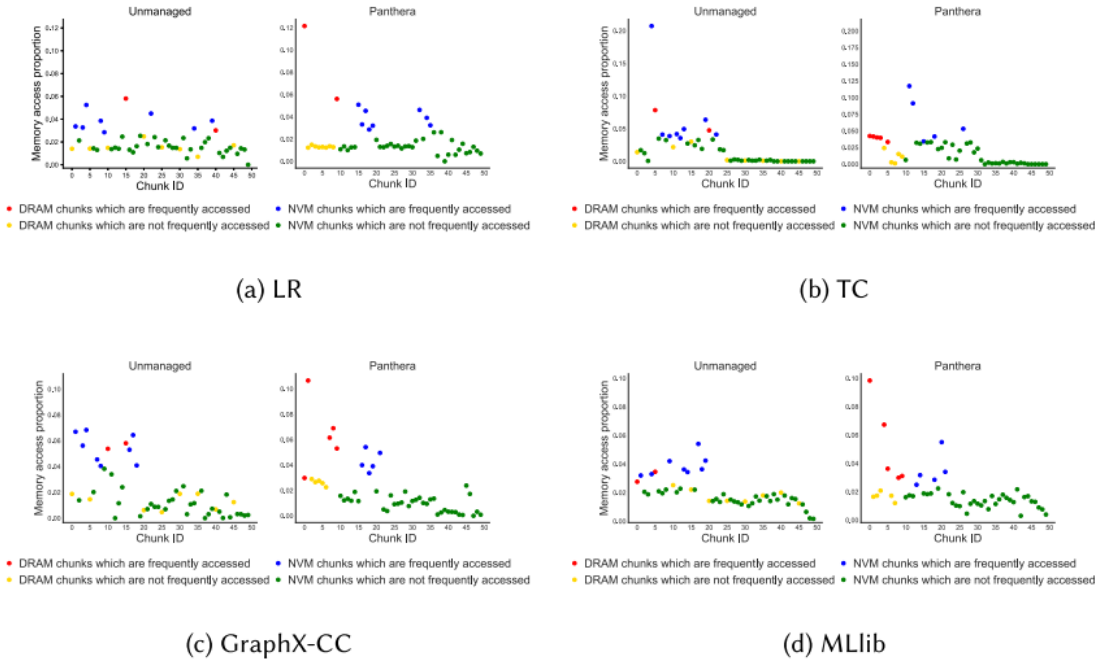


Fig. 7. Spatial distribution of memory access.

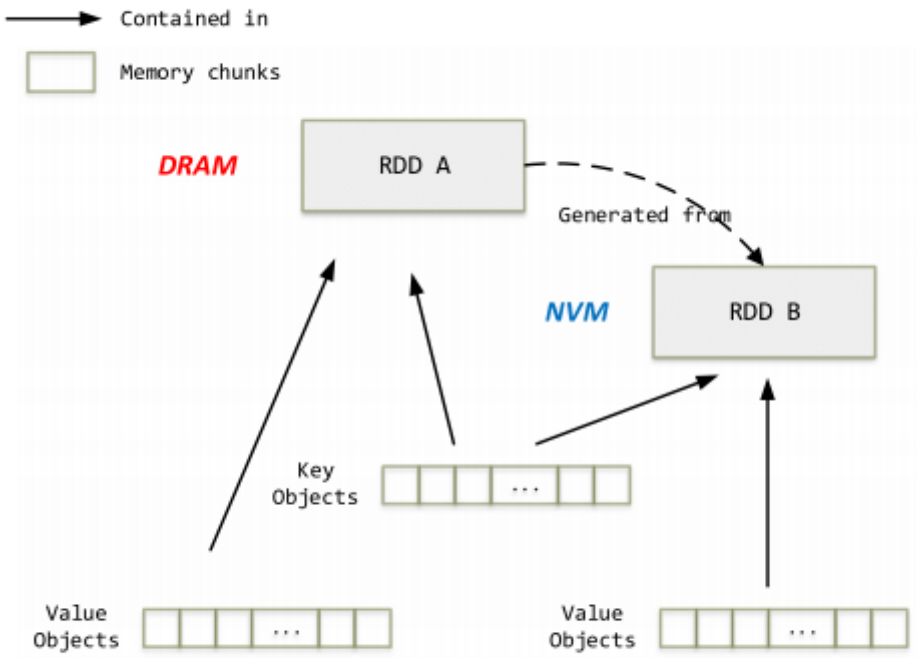


Fig. 8. Example of data sharing.

从图中可以看出，与Unmanaged相比，Panthera可以将访问频率更高的块放在DRAM上，而将访问频率较低的块放在NVM上。以MLlib(图7(d))为例，在前10个经常访问的块中，Unmanaged方法在DRAM上分配两个块，而Panthera在DRAM上分配五个块。然而，仍然有一些经常访问的块放在NVM上，如蓝点所示，还有一些不经常访问的块放在DRAM上，如黄点所示，这是不希望看到的。数据错位的原因是没有PGO的Panthera将RDD视为一个整体，无法区分RDD内部的访问频率差异。

总之，我们在Spark上的关键发现是，属于同一个RDD的数据并不总是表现出相似的访问模式。这一发现促使我们引入细粒度块安置决定到Panthera。因此，我们只需要执行基于块的分析属于带注释的RDD的数据，而不是在过程中对所有数据对象进行分析程序执行。因此，我们对一些特殊的rdd集成了全局的粗粒度rdd级分析和细粒度基于块的分析。这将以轻量级的方式获得精确的内存访问模式。

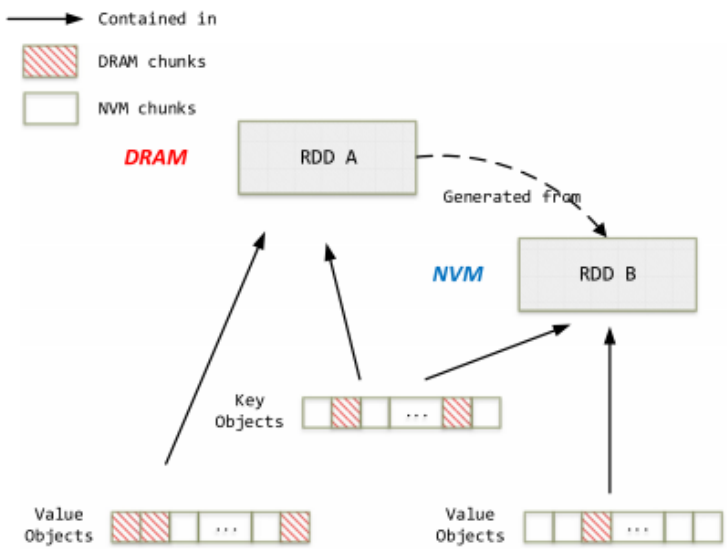


Fig. 9. Memory tags for each chunk after the profile-guided optimization.

运用优化的时机

由于RDD是多层Java数据结构，因此在没有使用性能分析优化（PGO）的情况下，Panthera在RDD的粒度上分析行为，因此所有属于一个RDD的对象都被确定为具有相同的访问模式和生命周期。然而，一些对象可能被多个被识别为具有不同行为的RDD共享，这为更精确地在DRAM/NVM之间分配对象提供了机会。

如第3.1节所讨论的，文中提到了一个对象共享的例子，并在静态情况下提出了一种算法，试图为共享对象分配相同的内存标记，如图3所示。然而，这种尝试可能会失败，并在为来自不同RDD的对象推断标记时可能会导致冲突。例如，在图8中，RDD B是从RDD A生成的，B和A分别被赋予“NVM”和“DRAM”的标记。因此，在生成B时，由A和B共享的数据对象将被移至“NVM”，即使其中一些属于被标记为“DRAM”的A。

基于这一观察，我们有机会动态切换共享对象的内存标记，即在访问B时使用“NVM”标记，在访问A时使用“DRAM”标记。为了实现这一目的，我们需要从RDD的粒度细化分析到更小的块，并利用对象的运行时行为。

最优化

为了在更小的块而不是RDD的粒度上表征行为，并将更频繁访问的块分配给DRAM，我们提出了一种基于性能分析的方法，其工作方式如下：首先，我们使用VTune收集所有块的内存访问分布，其中块大小为CS，如图7所示。其次，从访问分布中选择前K个频繁访问的块，其中K由老年代DRAM容量C和块大小CS使用K = C/CS的等式来确定。最后，将K个块的ID传递给JVM，当JVM启动时，我们通过调用mbind系统调用将前K个块绑定到“DRAM”，将其他块绑定到“NVM”以确定每个块的内存地址。
例如，我们假设块大小为1GB。当使用64GB堆且DRAM到内存比例为1/3且nursery空间为堆大小的1/6时，老年代中有10.66GB的DRAM。然后，将前11个频繁访问的块的ID传递给JVM，其中前10个块和第11个块的前0.66GB将被绑定到“DRAM”。

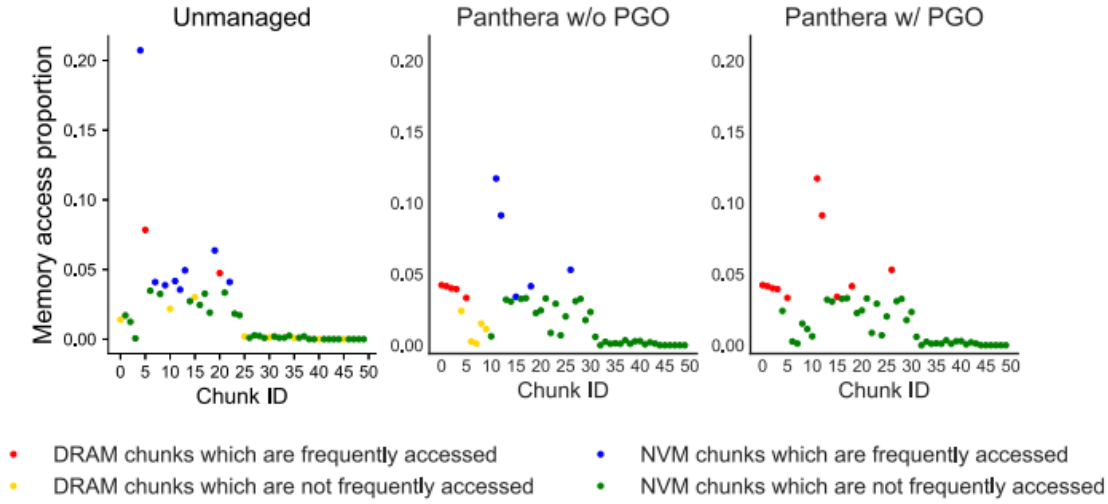


Fig. 10. Allocation results of DRAM chunks after the profile-guided optimization.

图9展示了带有性能分析优化的增强型Panthera框架，用于执行图8中的示例。具体而言，在Spark分析器中，RDDs B和A被标记为“DRAM”和“NVM”，分别。当JVM启动时，经过性能分析的K个块将绑定到DRAM，其他块将绑定到NVM。因此，从RDD的角度来看，B仍然会分配到老年代DRAM部分，而A仍然会分配到老年代NVM部分。然而，通过我们的底层块绑定，只有A和B的性能分析热门K块实际上会分配到DRAM。

因此，我们利用内存块的性能分析访问频率，优化静态确定的DRAM/NVM划分，仅将真正的热块分配到DRAM。图10显示了将PGO应用于Panthera的结果。从这个图中可以看出，对于带有PGO的Panthera，大多数频繁访问的块都正确地标记为“DRAM”标签并放置在DRAM中。与未管理的Panthera和没有PGO的Panthera相比，带有PGO的Panthera显著提高了老年代DRAM的累积访问比例，从19.55%和25.2%提高到53.5%。

关于适用性和普遍性的讨论

Panthera的设计包括两个关键的独立机制，即分析依赖于框架的数据访问模式(如第3节中的Spark Analyzer和QuickCached Analyzer)，以及一组独立于框架的api，这些api使得任何使用大型数组作为骨干数据结构的内存大数据系统都可以轻松地进行伪装、迁移和动态监控。在我们的设计中，清晰和可预测的数据访问模式是不同的顶部框架和增强的JVM之间的连接。

通用内存管理策略

增强的JVM是通用的，因为Panthera运行时系统提供的数据放置和迁移机制可以用于管理任何具有清晰和可预测数据访问模式的大数据系统的内存。示例包括Apache Hadoop、Apache Flink或数据库系统(如Apache Cassandra)。

Panthera通过三个一般政策确定数据的放置和迁移。这些策略被集成到增强的JVM中，因此它们通常适用于其他基于JVM的大数据系统。

- 首先，一些数据结构可以根据其静态确定的行为使用一些标记进行预保管。Panthera会将这些数据结构直接分配到相应标签所指示的空间中。
- 其次，需要一些数据结构来收集它们的行为并确定它们在运行时的位置。在我们当前的Panthera实现中，这些数据结构的放置取决于它们的访问频率和生命周期，因此Panthera将收集这些运行时特征并在应用程序执行期间做出放置决定。此外，Panthera可以扩展到集成新的内存访问特性和新的策略。
- 第三，需要一些数据结构来做出更细粒度的放置决策。对于这些数据结构，Panthera利用了一种更细粒度的基于块的内存访问分析方法，该方法支持在不同的内存空间放置不同的块。

然而，Panthera目前不支持那些不能清晰区分数据访问模式的应用程序，例如随机内存访问的应用程序。

特定于框架的访问模式注释/分析器

内存访问模式是通过用户注解和静态分析器获取的。Panthera提供了两个主要的API，一个用于带有标签的预年老数据结构，另一个用于动态监控和迁移。第一个API以数组和标签作为输入，执行如前文第4节所述的数据放置。标签可以来自程序中开发人员的注解或专为优化框架设计的静态分析。
举例来说，考虑Apache Hadoop，其中map工作器和reduce工作器可能需要在内存中保存大型数据结构。其中一些数据结构是从HDFS加载的不可变输入，而另一些则经常被访问。在HashJoin的情况下，这是SQL引擎的一个构建块，一个输入表完全加载到内存中，而第二个表被分割到map工作器之间。如果map工作器在单独的线程中执行，它们都共享第一个表，并将第二个表的各自分区与其连接。第一个表寿命长且频繁被访问。因此，它应该被标记为DRAM并放置在老年代的DRAM空间中，而第二个表的不同分区可以放置在年轻代，并且它们会在那里迅速死亡。
Panthera的第二个API以数据结构对象作为输入，以跟踪对该对象的调用次数。如果使用此API来跟踪数据结构的访问频率，则该数据结构（及从中可达的所有对象）将不会被预先老化（由第一个API指定），而是会在主要GC中进行动态迁移。我们可以使用此API来动态监视某些对象，并在其访问模式难以在静态情况下预测时对其进行迁移。
使用上述两个API使得一种灵活的分配/迁移机制成为可能，允许数据结构的某些部分（例如，可以轻松推断内存标签的部分）被预先老化，而其他部分则动态迁移。此外，特定于框架的内存访问模式分析器将执行def-use分析，以将用户注解传播到运行时系统。在第3节中，我们分别演示了两个用于Spark和QuickCached的个别分析器，它们不容易被其他新框架重用。

将Panthera应用于新框架

在将Panthera扩展到新框架时，我们需要考虑以下关键问题：

首先，我们可能需要引入新的特定于框架的注解，以便运行时系统了解关键数据结构。例如，如第3节所述，为了将Panthera应用于QuickCached，引入了@CoreHashObject注解，以说明这是QuickCached中全局哈希表的数据结构。特别是，这些注解与静态分析器一起设计，并向分析器提供应用级别的知识。

其次，我们需要设计一个新的特定于框架的静态分析器来揭示内存访问模式。例如，开发了一个新的静态QuickCached分析器，以根据用户的注解识别核心数据结构并分配辅助数组。背后的见解是只有少量哈希表被频繁访问，这一见解指导我们引入辅助数组来保存这些经常访问的数据。同时，分析器将标记辅助数组，以便它可以被预先提升到老年代DRAM空间。请注意，分析器可以利用用户注解和一些特定于框架的启发式方法，以获取更精确的内存访问模式。

通过框架特定的注解和分析器，底层运行时机制将利用第二个API并动态确定数据结构的放置和迁移，而无需进行任何特定于框架的修改。例如，对于QuickCached，运行时系统将收集辅助数组中的频繁访问数据，因此这些数据将被分配到DRAM上。

评估

我们在OpenJDK (build jdk8u76-b02)中添加/修改了9186行c++代码来实现Panthera GC，编写了979行Scala代码来实现Spark的静态分析，编写了762行Java代码来实现QuickCached。

NVM仿真与硬件平台

先前的混合记忆体研究大多使用模拟器进行实验。但是，它们都不能很好地支持Java应用程序。我们不能在这些模拟器上执行基于托管运行时的分布式系统。也有仿真器，如Quartz[75]和PMEP[29]，支持使用商品多套接字(NUMA)硬件的大型程序的NVM仿真，但Quartz和PMEP都不能运行OpenJDK。这些模拟器要求开发人员使用自己的库进行NVM分配，这使得Panthera GC不可能在不使用这些库从头开始重

新实现整个分配器和GC的情况下迁移对象。

正如在[10]和[75]中所观察到的，NUMA的远程内存延迟接近NVM的延迟，因此，研究人员使用NUMA架构作为基准来测量仿真精度。

根据这一观察，我们在NUMA机器上构建了自己的模拟器，以模拟基于jvm的大数据系统的混合内存。

在实现我们的模拟器时，我们遵循了Quartz[75]。Quartz有两个主要组成部分:(1)它使用热控制寄存器来限制DRAM带宽;(2)为每个应用程序进程创建一个守护线程，并插入延迟指令来模拟NVM延迟。例如，如果一个应用程序的CPU停滞时间是S, Quartz将CPU停滞时间缩放为S × NVM_latency DRAM_latency，以模拟NVM的延迟效果。对于(1)，我们使用相同的热控制寄存器来限制读/写带宽。和Quartz一样，我们目前也不支持不同的读写带宽。对于(2)，我们遵循Quartz的观察，使用NUMA的远程内存的延迟来模拟NVM的延迟。

模拟NVM延迟的另一种方法是在JIT编译期间测量加载/存储，在每个加载/存储中注入软件创建的延迟。然而，这种方法的局限性在于它没有考虑缓存效果和内存级并行性。

Table 2. Emulated DRAM and NVM Parameters

	DRAM	NVM
Read latency (ns)	120	300
Bandwidth (GB/s)	30	10 (limited by the thermal control register)
Capacity per CPU	100s of GBs	Terabytes
Estimated price	5×	1×

我们使用一个CPU来运行所有计算，CPU本地的内存作为DRAM，远程内存作为NVM。具体来说，DRAM和NVM分别使用两个本地和两个远程内存通道进行仿真。仿真的NVM的性能规格与[75]中使用的规格相同，见表2。为了模拟NVM的较慢写入速度，我们使用了热控制寄存器来限制远程内存的带宽 - 仿真的NVM是全双工的，读写带宽各为10 GB/s。在我们的设置中，远程内存的延迟是本地内存的2.5倍。

能源估算。我们遵循李等人的方法[49]，用于估算NVM的能源。我们使用了Micron的DDR4设备规格[61]来建模DRAM的功耗。 NVM的能源包括静态和动态两个组成部分。与DRAM相比，静态部分可以忽略不计[50]。动态部分包括读取和写入的能量消耗。由于PCM需要高温操作，PCM数组读取的能量消耗约为DRAM的2.1倍[49]。
与DRAM写入相比，NVM写入消耗的能量要多得多。在发生行缓冲区未命中时，每次写入的能量消耗有三个组成部分：(1)数组写入，将数据从行缓冲区驱逐到银行数组，(2)数组读取，从银行数组获取数据到行缓冲区，以及(3)行缓冲区写入，将新数据从CPU最后一级缓存写入行缓冲区。假设行缓冲区未命中比率为0.5，我们通过考虑行缓冲区的写入能量（1.02 pJ/bit）、大小（即DRAM的8K比特[61]，32位宽的部分写回到NVM[49]）和未命中率（0.5），以及数组的写回能量（16.8 pJ/bit × 7.6% 适用于NVM）和读取能量（NVM为2.47 pJ/bit）来分别计算这三个组成部分。7.6%的因子是由于李等人的优化[49]，只将脏字的7.6%写回NVM数组。
使用VTune[7]收集的CPU未核心事件用于计算读取和写入的次数。特别是，我们使用的事件是UNC_M_CAS_COUNT.RD和UNC_M_CAS_COUNT.WR。 VTune还可以区分对本地和远程内存的读取和写入。

实验设置

我们建立了一个小型集群，用一个主节点和一个从节点来运行Spark——这两个服务器有一个特殊的Intel芯片组，带有一个"可扩展内存缓冲区"，可以调优为远程内存访问产生2.5倍的延迟，这与NVM的读/写延迟相匹配。由于我们的重点不是分布式计算，因此这个集群足以让我们在Spark上执行实际工作负载，并了解它们在混合内存上的性能。表3给出了Spark主节点和从节点的硬件配置。每个节点有两个8核CPU，并行清除收集器(Panthera构建在其上)在每个GC中创建16个GC线程来执行并行跟踪和压缩。

GC延迟的负面影响随着计算节点数量的增加而增加。如文献[57]所述，在单个节点上运行的GC可以占用整个集群——当一个节点从另一个正在运行GC的服务器请求数据分区时，请求节点在第二个节点上完成GC之前不能做任何事情。由于Panthera可以显著提高NVM上的GC性能，我们期望Panthera在大型NVM集群上执行Spark时提供更大的好处。

Table 3. Hardware Configuration for Our Servers

Arch	NUMA, 4 sockets QPI 6.4 GT/S, directory-based MESIF
CPU	E7-4809 v3 2.00 GHz, 8 cores, 16 HW threads
L1-I	8 way, 32 KB/core, private
L1-D	8 way, 32 KB/core, private
L2	8 way, 256 KB/core, private
L3	20 way, 20 MB, shared
Memory	DDR 4, 1,867 MHz, SMI 2 channels

系统配置。每个CPU都有128GB的DRAM。我们为操作系统和Spark可用的最大DRAM量保留了8GB的DRAM，Spark运行的JVM的堆大小有两种（64GB和120GB），DRAM大小有三种（堆大小的1/4、1/3和100%；堆的其余部分是NVM）。对于QuickCached，我们使用了64GB的堆大小，并尝试了两种不同的DRAM大小（堆大小的1/3和100%）。100% DRAM的配置被用作计算Panthera在混合内存下的开销的基准。

以前关于NVM的工作通常在其配置中使用较小的DRAM比例。例如，Write Rationing [11]在实验中使用了1GB的DRAM和32GB的NVM。然而，由于我们处理大数据系统，使用非常小的DRAM比例对我们来说是不可能的——在我们的实验中，一个常规的RDD需要10-30GB的内存，因此，我们必须使DRAM足够大以容纳至少一个RDD。

幼儿园空间完全放置在DRAM中。我们尝试了几个不同的大小（堆大小的1/4、1/5、1/6和1/7）的幼儿园空间。在原始JVM下，1/4、1/5和1/6配置之间的性能差异很小，而1/7配置导致性能较差。我们最终在Spark和QuickCached的实验中都使用了1/6的配置，以获得良好的幼儿园性能，同时为老年代留下更多的DRAM。

程序和数据集。对于Spark，我们选择了七个多样化的程序。表4列出了这些程序、用于运行它们的数据集以及它们的内存占用情况。这些是各种任务的代表性程序，包括数据挖掘、机器学习、图形和文本分析。PR、KM、LR和TC直接在Spark上运行；CC和SSSP是在GraphX [33]上运行的图形程序，GraphX是构建在Spark之上的分布式图引擎；BC是MLib中的一个程序，MLib是构建在Spark之上的机器学习库。我们使用真实世界的数据集运行了这七个程序。请注意，尽管这些输入数据集的大小并不是很大，但在计算过程中可能会生成大量的中间数据。

为了评估QuickCached的性能，我们使用Yahoo! Cloud Serving Benchmark（YCSB）[22]。 YCSB是一个常用于评估云存储服务性能的基准套件。我们在加载了3000万条记录的数据库之后运行其A、B、C、D和F工作负载。默认情况下，每个记录是1KB。对于每个工作负载，我们执行1000万次操作。

基线。我们最初的目标是将Panthera与Espresso [80]和Write Rationing [11]进行比较。然而，它们中的任何一个都不是公开可用的。Espresso提出了一个供开发人员开发新应用程序的编程模型。将其应用

于大数据系统意味着我们需要重写每个分配点，这显然是不切实际的。此外，Espresso不根据访问模式迁移对象。

Table 4. Spark Programs, Datasets and Memory Footprints

Program	Dataset	Initial Size	Footprint
PageRank (PR)	Wikipedia Full Dump, German [4]	1.2 GB	63.0 GB
K-Means (KM)	Wikipedia Full Dump, English [4]	5.7 GB	49.5 GB
Logistic Regression (LR)	Wikipedia Full Dump, English [4]	5.7 GB	63.4 GB
Transitive Closure (TC)	Notre Dame Webgraph [2]	21 MB	43.8 GB
GraphX-Connected Components (CC)	Wikipedia Full Dump, English [4]	5.7 GB	59.1 GB
GraphX-Single Source Shortest Path (SSSP)	Wikipedia Full Dump, English [4]	5.7 GB	62.2 GB
MLlib-Naive Bayes Classifiers (BC)	KDD 2012 [1]	10.1 GB	63.1 GB

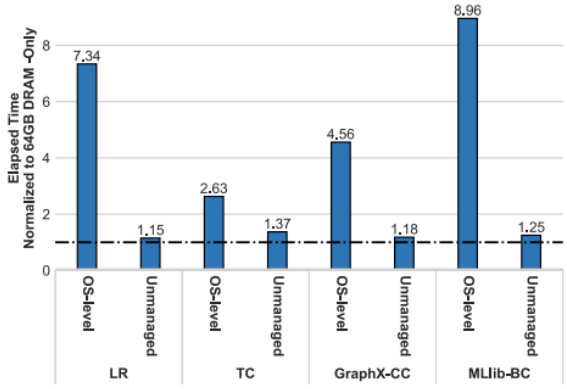


Fig. 11. Overall performance comparison between OS-level management and *unmanaged*. The heap size is 64GB and DRAM to memory ratio is 1/3.

Write Rationing GC有两种实现：Kingsguard-Nursery（KN）和KingsguardWrites（KW）。KN将年轻代放在DRAM中，老年代放在NVM中。KW也将年轻代放在DRAM中。与KN不同，KW监视对象写入并动态将写入密集型对象迁移到DRAM中。虽然我们无法直接将Panthera与这两个GC进行比较，但我们在OpenJDK中实现了类似的算法。在KW下，几乎所有的持久化RDD都迅速移动到了NVM中。这些RDD的频繁NVM读取，以及用于监视对象写入的写入屏障，导致我们的基准测试平均性能开销达到41%。这是因为大数据应用表现出与常规非数据密集型Java应用程序不同的特征。

KN乍一看似乎是一个很好的基准。然而，在Parallel Scavenge收集器中天真地实现它可能导致非常大的开销——NVM中带宽的降低可能对多线程程序的性能产生巨大影响；对于试图充分利用CPU资源执行并行对象扫描和压缩的Parallel Scavenge来说，尤其如此。

为了获得更好的基准，我们将年轻代放在DRAM中，并使用DRAM和NVM的混合来支持老年代。具体来说，我们将老年代的虚拟地址空间划分为多个1GB的块，使用概率确定一个块是否应映射到DRAM或NVM。概率是从系统中的DRAM比例派生的。例如，在DRAM与内存比例为1/4（1/4 DRAM）的系统中，每个块映射到DRAM的概率为1/4，映射到NVM的概率为3/4。请注意，这是一种常见的做法[32, 77]，用于利用DRAM和NVM的联合带宽。我们将这种配置称为未管理的，它在我们的基准测试中优于KN和KW。

还有一些基于OS级的数据迁移工作，例如Thermostat [9]、Translation Ranger [86]和HeteroOS [43]。我们尝试将这些框架移植到我们的模拟NVM平台，但这些工作都不适用于我们使用的基准测试。例如，在运行Thermostat时，Spark应用程序在执行过程中总是卡住，而HeteroOS则针对虚拟化环境中的混合内存，而不是像Panthera那样在裸机上运行。Translation Ranger的目标是通过主动合并分散的页面来加速虚拟到物理内存地址的转换。这是与Panthera的正交优化，因此我们没有在评估中包含它。

为了评估OS级的混合内存管理策略，我们利用基于LRU（Least Recently Used）的内核分页系统进行数

据迁移管理。我们在模拟的NVM上创建了一个ramdisk，并将其挂载为交换分区。我们根据最新的工作[12, 59]调整了分页系统的性能。在这些设置下，NVM充当辅助内存，类似于Optane DC Memory Mode [37]。内核将最近未使用的数据逐出到NVM，并将最近使用的数据保留在DRAM中。正如图11所示，OS级别的管理策略比未管理的策略差得多。与未管理相比，OS级别管理的减速可以达到6.20倍。这是因为GC总是打乱OS放置的数据布局，导致更多无用的数据迁移开销，正如我们在第1节中所述。因此，在后续的评估中，我们将未管理的作为基准。

无PGO的Panthera的性能和能量

图12报告了在使用64 GB堆的情况下Spark的整体性能和能耗结果，然后DRAM与内存比为1/3（1/3 DRAM）。每个配置的性能和能耗结果都是相对于64 GB仅DRAM版本的结果进行标准化的。在我们的实验中，能耗结果包括Panthera运行时的能耗，但不包括静态分析器和性能分析工具的能耗。与仅DRAM版本相比，未管理版本通过21.4%的执行时间开销减少了26.7%的能耗。相比之下，Panthera通过4.3%的执行时间开销减少了32.3%的能耗。

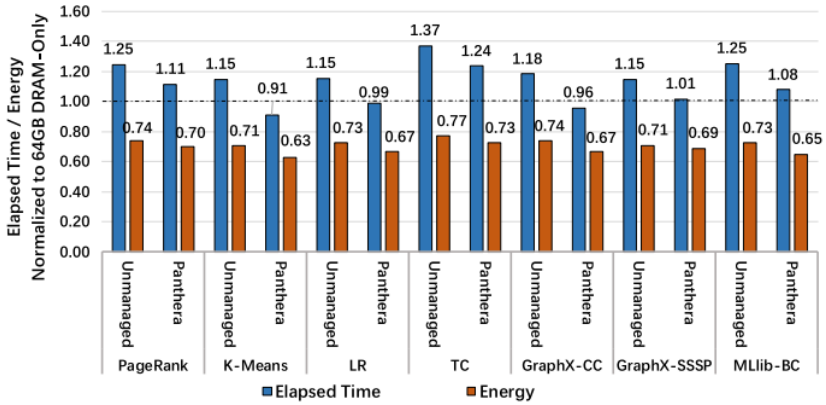


Fig. 12. Overall performance and energy results of Spark under a 64-GB heap; DRAM to memory ratio is 1/3.

当堆大小为120 GB时（图12中未显示，但在图14和图15中总结），未管理版本通过19.3%的执行时间开销减少了39.7%的能耗。相比之下，Panthera在不到1%的执行时间开销下减少了47.0%的能耗。显然，考虑到数据放置中的RDD语义在能耗和性能上都带来了显著的好处。

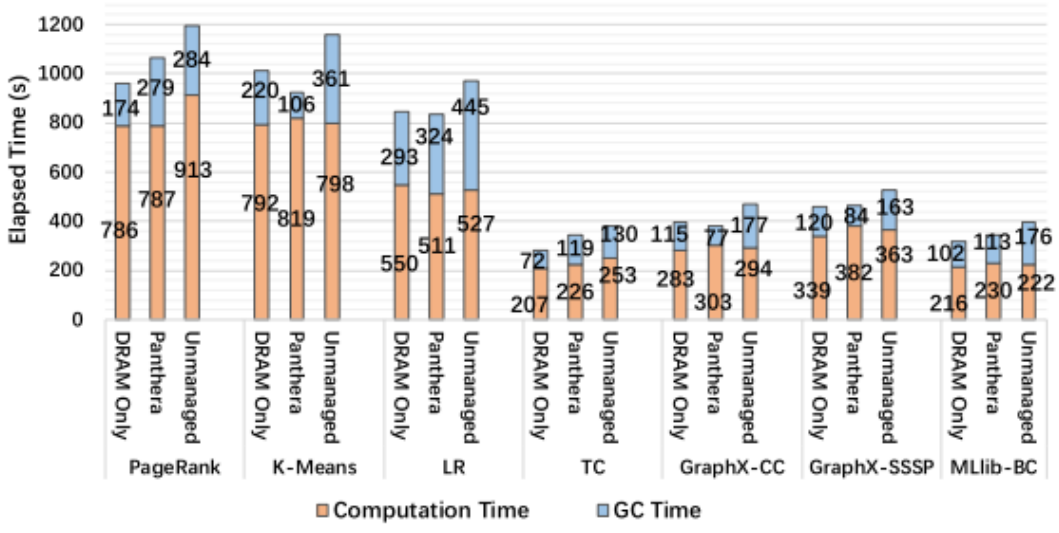


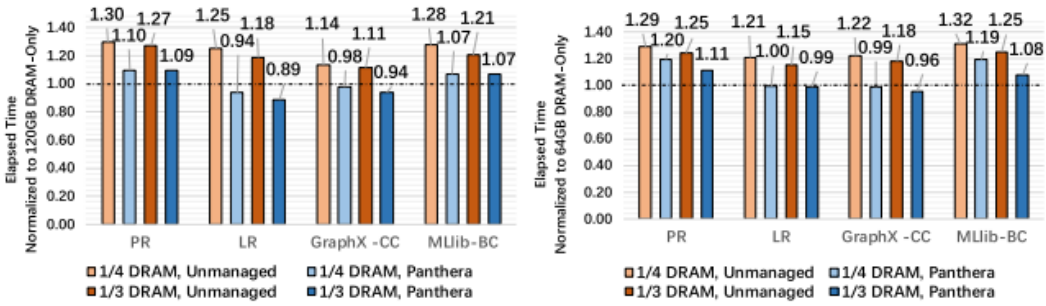
Fig. 13. GC performance (64-GB heap).

GC性能。为了了解GC性能，我们将每个程序的运行时间分解为mutator时间和GC时间；这些结果（在64 GB堆下）显示在图13中。与基准相比，未管理版本在GC和计算中分别引入了60.4%和6.9%的性能开销；而对于Panthera，这两个开销分别为4.7%和4.5%。在120 GB堆下，未管理版本和Panthera的GC性能开销分别为58.0%和3.1%。注意，由于产生了大量的中间数据，这些程序经常触发GC。

由于GC是一个内存密集型的工作负载，不恰当的数据放置可能导致内存访问时间显著增加，从而带来较大的惩罚。惩罚主要来自两个方面。首先，NVM的有限带宽（约为DRAM的1/3）对Parallel Scavenge的性能产生了很大的负面影响，Parallel Scavenge在每个（nursery和full-heap）GC中启动16个线程以执行并行追踪和对象复制。在这种高度并行性的情况下，当在NVM中扫描对象时，nursery GC的性能显著下降。其次，对象追踪是一个读密集型任务，受到NVM更高读取延迟的严重影响。

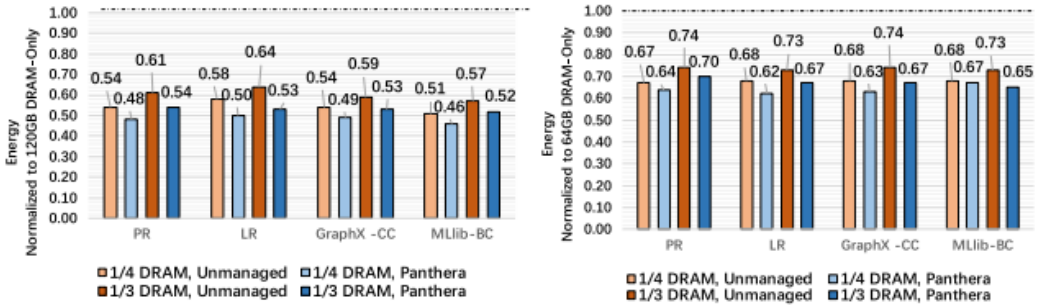
Panthera通过在DRAM中pretenuring频繁访问的RDD对象并执行优化（包括Section 4.2.2中的eager promotion和Section 4.2.3中的card padding）来提高GC性能。eager promotion减少了每个minor GC中的（old-to-young）追踪的成本，而card padding消除了在NVM中不必要的数组扫描，这对延迟和带宽都很敏感。进一步的细分显示，仅eager promotion就贡献了总GC性能提升的平均9%。card padding的贡献更为显著——如果没有这个优化，由于NVM的带宽大大受限以及并行卡扫描性能的增加，GC时间增加了60%。实际上，这种影响是如此之大，以至于在禁用card padding时，其他优化效果不佳。

不同的堆大小和比率。为了了解堆大小和DRAM比率（DRAM与总内存的比率）的影响，我们对四个程序PR、LR、CC和BC进行了两个堆大小（64 GB、120 GB）和两个DRAM比率（1/3、1/4）的实验。图14报告了这些配置的时间结果。在这四个配置（64 GB，1/4）、（64 GB，1/3）、（120 GB，1/4）和（120 GB，1/3）下，Panthera的时间开销平均分别为9.5%、3.4%、2.1%和0%。在这些相同的四个配置下，未管理版本的开销分别为25.9%、20.9%、23.9%和19.3%。



(a) 120GB Heap. (b) 64GB Heap.

Fig. 14. Performance for two DRAM ratios + two heaps.



(a) 120GB Heap. (b) 64GB Heap.

Fig. 15. Energy for two DRAM ratios + two heaps.

我们进行了两个有趣的观察。首先，Panthera对DRAM比率的敏感性大于堆大小。当DRAM比率从1/4增加到1/3时，时间开销几乎可以减少10%。原因是更多频繁访问的RDD被移动到DRAM，减少了NVM的内存延迟和带宽限制。另一个观察是未管理版本对DRAM比率的敏感性要小得多——当DRAM比率增加到1/3时，时间开销仅减少了5%。这是因为任意的数据放置使得大量频繁访问的数据留在NVM中，当访问NVM时，CPU会陷入重度停滞。

图15展示了两个堆大小和两个DRAM/NVM比率的能耗结果。对于64 GB堆，未管理版本在1/4和1/3 DRAM比率下分别平均减少了32.2%和26.5%的能耗，而Panthera在这些比率下分别减少了36.0%和32.7%的能耗。对于120 GB堆，未管理版本在1/4和1/3 DRAM比率下分别平均减少了45.7%和39.7%的能耗，而在这两个比率下，Panthera的能耗减少增至51.7%和47.0%。

我们还评估了NVM和DRAM的价格，以显示从使用混合内存中获益的硬件成本节省。最便宜的NVM的价格是每GB 7.85，而最便宜的$DRAM$是每GB16.61 [35]。与仅使用DRAM相比，使用DRAM比率为1/3的混合内存可以减少35.2%的硬件成本，当DRAM比率为1/4时，甚至可以减少39.6%。

QuickCached的结果。图16报告了在使用64 GB堆且DRAM到内存比率为1/3（1/3 DRAM）时，QuickCached的整体性能和能耗结果。每个配置的性能和能耗结果都是相对于64 GB DRAM版本的。与仅使用DRAM版本相比，未管理版本在9.1%的执行时间开销下能耗减少了25.0%。相反，Panthera在5.2%的执行时间开销下减少了28.7%的能耗。

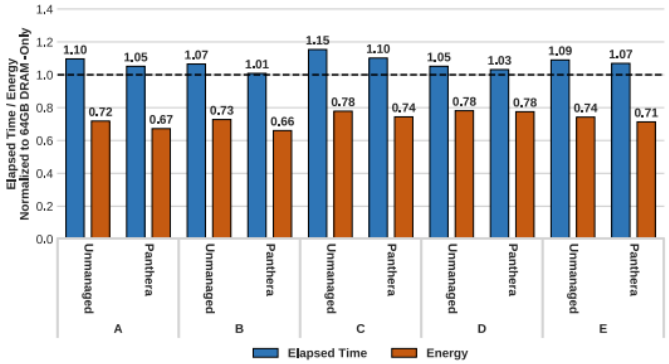
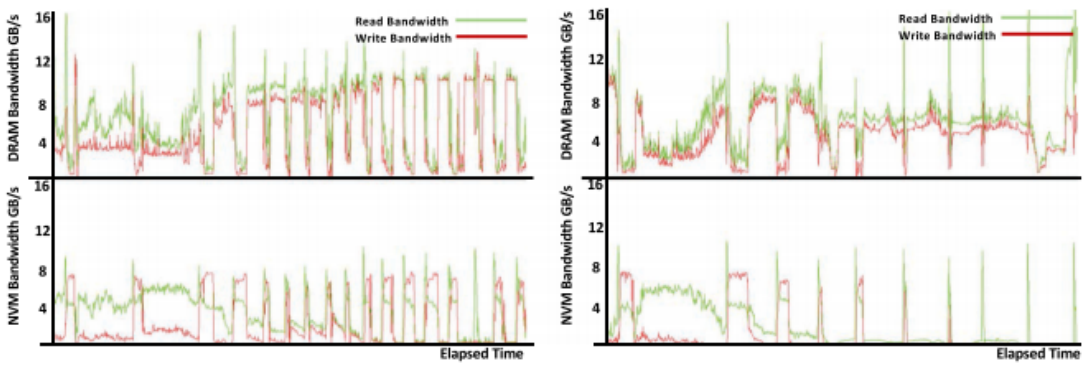


Fig. 16. Overall performance and energy results of QuickCached under a 64-GB heap; DRAM-to-memory ratio is 1/3.



(a) Unmanaged, DRAM ratio = 1/3　　(b) Panthera, DRAM ratio = 1/3

Fig. 17. GraphX-CC's memory access bandwidth.

内存访问分析

NVM具有高延迟和低带宽的特点。通常，高延迟导致的性能损失随着内存访问次数的增加而增加。对于相同数量的内存访问，对于瞬时带宽需求超出NVM带宽的应用程序，NVM会导致更高的性能损失。图17描述了GraphX-CC上非托管和Panthera的读/写带宽。与非托管版本相比，Panthera将大部分内存读写从NVM迁移到DRAM，并降低了高瞬时内存访问带宽需求(即图中的峰值)。因为Panthera分配/移动频繁访问的数据到DRAM，它减少了不必要的NVM访问(章节4.2.2和4.2.3)。

监视和迁移的开销

Table 5. Dynamic Monitoring and Migration

Program	# Calls monitored	# RDDs migrated
PR	328	0
KM	550	0
LR	333	0
TC	217	0
CC	2,945	1
SSSP	3,632	1
BC	336	0

正如在第4.2节中讨论的那样，Panthera对RDD对象执行轻量级的方法级监控，以检测放错位置的RDD以进行动态迁移。本小节更详细地讨论了动态迁移的开销。

由于我们只监控在RDD对象上调用的方法，我们发现动态监控的开销可以忽略不计，即在我们的所有基准测试中都不到1%。例如，对于PageRank，仅在20分钟的执行中观察到了所有RDD对象中的约300次调用。表5的第二列报告了每个应用程序监控的调用次数。对于具有数千个RDD调用的GraphX应用程序，监控开销仍然低于1%。

在我们的实验中，动态迁移（由主GC执行）很少发生，如表5的第三列所示。这主要有两个原因。首先，主要收集的频率非常低，因为大多数对象都很快就会死亡，而大部分收集工作由次要GC完成。其次，对于四个应用程序（PR，KM，TC和LR），我们的静态分析结果足够准确，因此永远不需要进行动态迁移。

我们观察到只有两个RDD（在CC和SSSP的执行期间）被动态迁移。请注意，CC和SSSP都是GraphX应用程序。处理的每个迭代都会创建新的RDD，表示更新后的图，并将它们持久化。在每个迭代结束时，表示旧图的RDD被显式取消持久化。由于静态分析不支持unpersist调用，它将旧图和新图的RDD都标记为热数据，并为它们生成了一个DRAM标签。然后，在DRAM中分配这些RDD对象，并将它们的数据对象急切地提升到老年代的DRAM空间。表示旧图的RDD对象（如果它们可以在主GC中幸存）由于其低访问频率而迁移到老年代的NVM空间。

为了更好地了解预分配和动态迁移的各自贡献，我们已禁用了监控和迁移，并重新运行了整个实验。性能差异微不足道（即，小于1%）。因此，我们得出结论，Panthera的大部分好处来自于预分配，它改善了mutator和GC的性能。然而，动态监控和迁移增加了Panthera优化的通用性，使其适用于具有不同访问特性的应用程序。

PGO的性能和能量

为了检验我们的基于分析的优化的有效性，我们使用表4中列出的基准测试评估了启用了PGO的Panthera。我们尝试了两种堆大小，64 GB和120 GB，nursery空间占堆大小的1/6，而堆的1/3是DRAM。我们将启用PGO的Panthera与禁用PGO的Panthera以及未经管理的版本进行了比较。请注意，我们的分析是离线应用的，因此不会引入额外的开销。

图18显示了使用64 GB堆时的整体性能和能量结果。结果是相对于仅DRAM版本的结果进行归一化的。与未启用PGO的Panthera相比，PGO可以在平均3.9%的执行时间减少的情况下减少5.8%的能量。与64 GB DRAM-only版本相比，启用PGO的Panthera在0.2%的执行时间开销下可以实现36.4%的能量减少。然而，我们可以看到一些应用程序，例如LR，Graphx-CC和Graphx-SSSP，只能从PGO中获得微小的好处。有两个基本原因。首先，这些应用程序的RDD中的访问模式是均匀的，不需要将RDD划分为更细的块。在这种情况下，未启用PGO的Panthera可以识别并将数据迁移到正确的位置，如图7所示。其次，一些内存访问模式，例如流式处理，在RDD上容易被识别。硬件和操作系统的预取机制对数据运行良好。在这种情况下，即使启用了PGO的Panthera通过识别和迁移具有更高访问频率的块可以比未启用PGO的Panthera更好地放置数据，它也不能获得太多的好处。

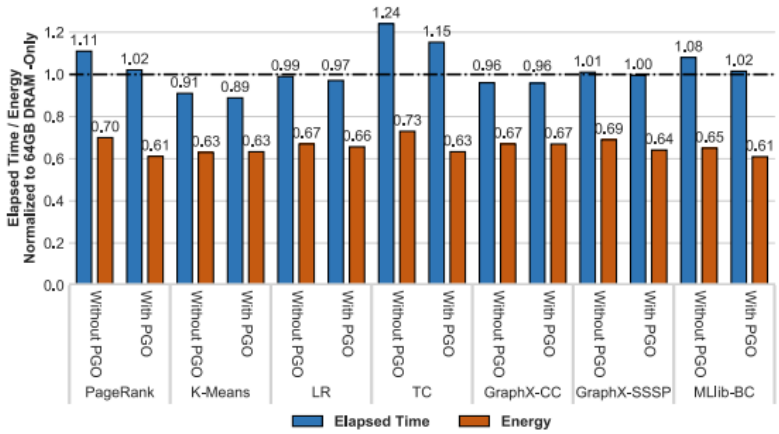


Fig. 18. Performance and energy results of Panthera with/without PGO. Heap size is 64 GB.

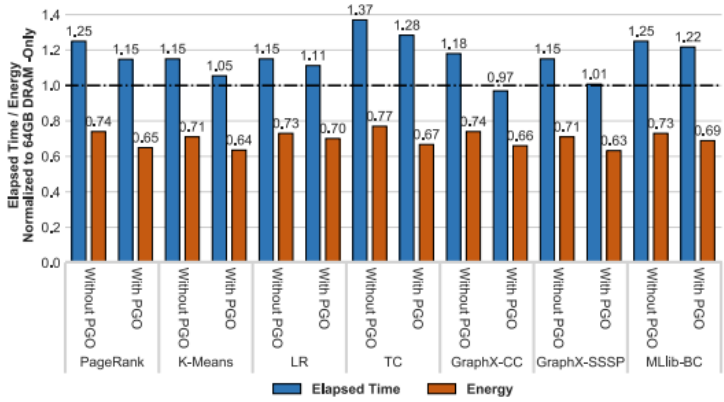
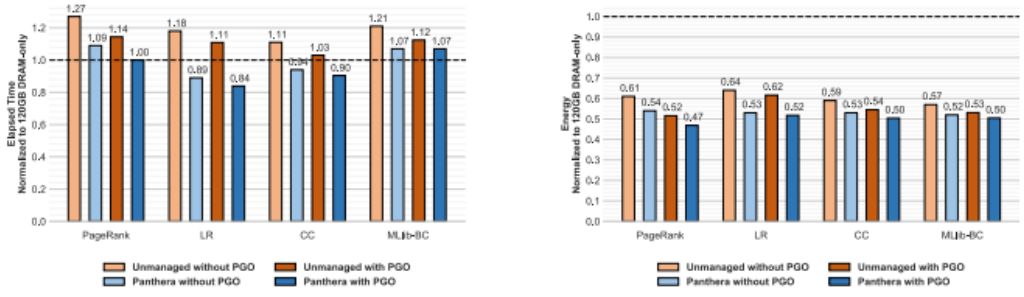


Fig. 19. Performance and energy results of Unmanaged with/without PGO. Heap size is 64 GB.



(a) Elapsed time. (b) Energy.

Fig. 20. Overall performance and energy results under a 120-GB heap.

我们的PGO可以与Panthera解耦，并集成到未管理的版本中，图19显示了在未管理的版本中实现PGO时的整体性能和能量结果。结果相对于64 GB DRAM-only版本进行了归一化。与未启用PGO的Unmanaged相比，启用PGO的Unmanaged在平均8.7%的执行时间减少的情况下减少了9.6%的能量成本。此外，与64 GB DRAM-only版本相比，启用PGO的Unmanaged在11.8%的执行时间开销下可以实

现33.8%的能量减少，而未启用PGO的Unmanaged在21.4%的执行时间开销下减少了26.7%的能量。

图20显示了使用120 GB堆时的整体性能和能量结果。结果相对于DRAM-only版本进行了归一化。通过PGO，Unmanaged在平均10.2%的执行时间开销下减少了44.8%的能量，而Panthera在平均不到1%的执行时间开销下减少了50.1%的能量。

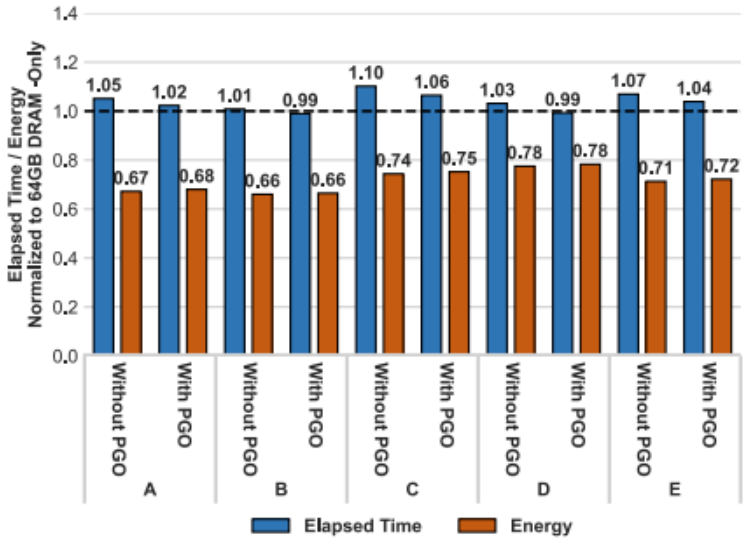


Fig. 21. Performance and energy results of Panthera with/without PGO.

快速缓存的PGO结果。图21报告了使用PGO到QuickCached时的性能和能耗结果。我们使用与7.2节中描述的相同的实验配置。结果与仅使用dram的版本相比是标准化的。与无PGO的Panthera相比，PGO平均减少2.9%的执行时间，而能耗几乎相同。PGO对QuickCached的好处不如Spark，因为:(1)引入PGO进行细粒度块放置，并在静态粗粒度rdd级分析的基础上进一步提高性能。对于QuickCached，Panthera在运行时推断每个对象的内存标签，这已经是细粒度的;(2) QuickCached的NVM访问比率小于Spark，例如，占所有内存访问的10%以下，因为在2.5中描述的有偏差的对象访问特征。

讨论

离理想状态有多远. 对于具有托管运行时的混合内存系统，理想的解决方案是根据对象的热度正确地放置每个对象，并在考虑对象迁移的成本和收益时动态调整其位置。由于在正常执行期间大数据系统的堆中通常包含数十亿个对象，要对每个对象的放置位置以及何时进行迁移进行分析是困难的。Panthera引入了一个简化的观察，即我们可以开发一个简单的静态分析来推断每个粗粒度数据集的访问模式，其中所有对象共享相同的模式。此外，正如在第7.5节中所述，这一简单的假设足够准确。

仿真NVM规格的影响. 由于Quartz的限制，尽管大多数仿真的延迟/带宽结果与先前研究中使用的规格相匹配[75]，但与真实的NVM硬件（例如Optane DC [37]）仍然存在一些差异。仿真NVM和真实NVM硬件之间存在两个主要差异。首先，仿真的NVM未显示NVM在读/写延迟和带宽方面的不对称性。其次，仿真规格与实际硬件并不完全相同。例如，我们仿真的NVM的读/写延迟为300 ns/300 ns，而Optane DC [37]的规格为305 ns/94 ns。然而，我们强调不同的NVM技术具有不同的规格，并且仿真的NVM已经展示了NVM的性能特性以及DRAM和NVM之间的性能差异。我们对仿真NVM的全面评估可以展示我们提出的静态/动态分析的可忽略开销和高准确性，以及我们数据迁移策略的有效性。例如，我们将仿真的NVM读/写延迟从300 ns/300 ns调整到120 ns/120 ns，基准的Unmanaged版本仍然由于受限的读/写带宽而导致显着的性能降低。在这些设置下，Panthera的性能仍然平均优于基准11.6%。在固定的300 ns读/写延迟下，我们还将读/写带宽从5 GB/s调整到12 GB/s（12 GB/s是我们服务器QPI的带宽限制），Panthera在各种设置下始终优于基准，平均提高了从32.3%到11.9%。

相关工作

托管运行时的混合内存。据我们所知，Panthera是第一个为基于管理运行时的分布式大数据平台优化混合存储器数据布局的实际工作。现有的努力[11、17、32、39、40、67、72、76、80]试图支持持久化Java，但重点放在常规应用程序上，或者需要重新构建平台。

Inoue和Nakatani[36]确定了Java应用程序中可能导致L1和L2缓存丢失的代码模式。Gao等人[31]提出了一个框架，包括硬件、操作系统和运行时的支持，以延长NVM的生命周期。最近两个接近Panthera的作品是Espresso[80]和Write Rationing[11]。然而，它们并不是为大数据系统设计的。Espresso是一个基于jvm的运行时系统，支持持久堆。当运行时系统为堆提供崩溃一致性时，开发人员可以使用新的指令计划在持久堆中分配对象。使用Espresso需要使用pnew重写大数据平台(例如Spark)，这是不实际的。

Write Rationing[11]是一种GC技术，它将高度变异的对象放在DRAM中，而将大部分读对象放在NVM中，以增加NVM的生命周期。与Espresso一样，这种GC关注于单个对象，而不考虑应用程序语义。Panthera的托儿所空间也被放置在DRAM中，类似于写配给中的御林铁卫托儿所。然而，Panthera不是专注于单个对象，而是利用Spark语义在数组粒度上获取访问信息，从而实现有效的预定义和高效的运行时对象跟踪。

内存结构。混合存储器结构有两种:一种是平面结构，即DRAM和NVM共享一个存储空间;另一种是垂直结构，即DRAM作为NVM存储热数据的缓冲区。垂直结构通常由硬件管理，对操作系统和应用程序透明[44,49,54,58,69,88,90,93]。Qureshi等[69]表明，仅含3% DRAM的垂直结构可以达到与仅含DRAM的版本相似的性能。

然而，页面监控和迁移的开销随着工作集的增加呈线性增加[77]。空间开销，如DRAM缓冲区的标签存储空间，在大量NVM的情况下也会很高[60]。

页面迁移。大量现有的作品使用内存控制器来监控页面读/写频率[20,25,30,34,53,68,70,77,88,92]，并将排名靠前的页面迁移到DRAM中。另一种由3d堆叠DRAM和商品DRAM组成的混合内存也采用类似的页面监控策略[28,38]。然而，这些技术都不是为大数据系统设计的。Hassan等[34]表明，对于某些应用，在对象级别迁移数据可以降低功耗。

对于内存消耗非常大的大数据应用程序，以页面粒度进行持续监控可能会导致不合理的开销。页面迁移还会导致时间和带宽方面的开销。Bock等人[18]报告说，页面迁移可以使执行时间平均增加25%。Panthera使用静态分析来跟踪RDD粒度的内存使用情况，结合程序语义来减少动态监控开销。

Static Data Placement. 有一系列的研究致力于根据数据的访问频率[20, 53, 68, 74, 88]或程序分析的结果[30, 34, 77]直接将数据放置在适当的空间中。通常使用静态数据活跃性分析或离线分析来计算访问频率。Chatterjee等人[20]将单个高速缓存行放置在多个内存通道上。高速缓存行中的关键字（通常是第一个字）放置在低延迟通道中。Wei等人[77]表明，在源代码中由相同站点分配的对象组表现出相似的生命周期行为，可以利用这些信息进行静态数据放置。Dulloor等人[30]将内存访问分类为三种模式，并对给定映射的数据结构到不同内存类型的访问时间进行建模，以获得最佳映射配置。

Li等人[53]开发了一个二进制插装工具，用于统计报告堆栈、堆和全局数据的内存访问模式。Phadke和Narayanasamy[68]通过对应用的MLP和LLC缺失进行分析，确定应用程序最能从哪种类型的内存中受益。Kim等人[45]为具有大型分布式NVM的高性能计算机开发了一个键值存储，该存储为开发人员提供了一个高级接口，以使用分布式NVM。然而，这些技术都不是为托管的大数据系统设计的。

结论

我们提出了Panthera，这是第一个在混合存储器上管理大数据处理的内存管理技术。Panthera结合了静态分析、GC技术和配置文件引导优化，在混合内存系统中执行语义感知的数据放置。我们的评估表明，Panthera在不增加额外时间开销的情况下显著降低了能耗。