在YARN上的Spark和MapReduce的跨平台资源调度

Dazhao Cheng, Xiaobo Zhou, Palden Lama, Jun Wu and Changjun Jiang

**摘要—虽然MapReduce本质上是为批处理和高吞吐量处理工作负载而设计的，但对大数据的非批处理流程（例如交互式作业，实时查询和流计算）的需求正在不断增长。新兴的Apache Spark填补了这一空白，它可以在已建立的Hadoop集群上运行并利用现有HDFS的优势。结果，Spark-on-YARN的部署模型被许多行业领导者广泛采用。但是，我们确定了在YARN上部署Spark的三个主要挑战，不灵活的基于预留的资源管理，任务间相关性盲目调度以及Spark和MapReduce应用程序之间的局部性干扰。这三个挑战导致资源利用效率低下和性能显着下降。我们提出并开发了一种跨平台的资源调度中间件iKayak，旨在提高多租户YARN集群中的资源利用率和应用程序性能。 iKayak依赖于三种关键机制：预留感知的执行器放置以避免长时间等待资源预留；依赖感知的资源调整以利用减少任务占用的未充分利用的资源；以及跨平台的本地感知任务分配，以协调之间的本地竞争。 Spark和MapReduce应用程序。我们在YARN中实施iKayak。与两种流行的YARN客户端模型和YARN群集模型相比，iKayak在Spark应用程序上的性能提高了50％，在MapReduce应用程序上的性能提高了19％。**

**1 介绍**

在过去的几年中，MapReduce彻底改变了大数据并行和分布式处理。 事实证明，MapReduce是实现复杂批处理应用程序的有效平台，这些应用程序包括对系统日志进行筛选，运行提取转换加载操作以及计算Web索引。 但是，其固定一次的计算模型使MapReduce不适合低延迟应用程序和迭代计算，例如机器学习和图形算法。 最近，新兴的Apache Spark [1]通过泛化MapReduce计算来解决这些限制。 Spark使应用程序能够可靠地将数据存储在内存中。 每个Spark应用程序都有多个称为执行程序的进程，它们在集群上运行，以代表其将相关数据加载到内存中，即使它没有运行任何作业。 它允许应用程序避免昂贵的磁盘访问，这是Spark高性能的关键。

因此，通过集成Spark进行交互式和流计算，当今许多大数据部署都超越了MapReduce。但是，将统一资源池划分为较小的池以用于不同的应用程序会导致资源利用效率低下。具有动态资源分配的单个群集管理器可以提高资源利用率。特别是，企业倾向于将新兴的Spark应用程序部署在其现有的Hadoop集群上，以利用已建立的集群，访问现有的HDFS数据集并利用Hadoop的安全环境。例如，eBay和Yahoo! [2]使用MapReduce生成报告并回答历史查询，同时部署Spark实时计算关键指标。 Hadoop YARN [3]是新兴的跨平台集群管理器。它允许多个计算平台在单个群集上共存和共享资源，并受益于其细粒度的资源管理方案。但是，我们发现YARN的基于预留的资源调度策略[4]与Spark应用程序的动态需求之间存在语义鸿沟，这导致资源利用效率低下和应用程序性能低下。具体而言，Spark-on-YARN提出了以下几个关键挑战。

首先，YARN的基于预留的资源调度策略使具有高资源需求的任务很难及时获得所需资源。不幸的是，Spark是这种应用程序。 Spark基于多线程编程模型。这种特性可能导致一个场景，即单个Spark执行者一次占用大量资源。因此，具有高资源需求的执行者可能不得不等待很长时间进行资源预留，从而导致资源利用率低和性能差。 更糟的是，当需要的资源将被保留但长时间无法满足时，对于资源需求非常大的某些作业（例如Spark流）可能会出现饥饿。

其次，YARN中的现有调度程序无法识别map和reduce任务之间的依赖关系。 由于保留了群集资源并与Spark应用程序共享，因此已经启动的reduce任务的MapReduce作业可能无法及时完成其所有映射任务。 由于归约任务要等到所有映射任务完成后才能执行其功能，因此启动的归约任务将继续占用资源并等待映射任务的完成。 因此，这导致分配给缩减任务的资源利用率低。

1. MapReduce和Spark应用程序都尝试将任务或执行程序放置在与其相关的HDFS块旁边，以提高对位置的了解，而它们需要与YARN进行协商以进行资源调度。 特别是，作业可能会在共同托管Spark执行程序和MapReduce任务的节点上显示较差的数据局部性。 例如，当Spark应用程序计算计数任务时，MapReduce作业可能需要相同的计数数据集。 但是，如果节点主要由Spark执行程序占用，则MapReduce作业可能无法获取具有本地数据集的节点上的资源。 因此，由于多租户竞争而引起的局部干扰会损害Spark和MapReduce应用程序的性能。

在这项工作中，我们提出并开发了一种跨平台的资源调度中间件iKayak，旨在提高多租户YARN集群中的资源利用率和应用性能。 iKayak依靠三个关键设计，分别利用了不同应用程序的时变资源需求，地图任务和约简任务之间的任务间依存关系以及跨平台局部性意识来分别应对上述挑战。我们首先设计并开发了一个预留感知执行器放置机制，以为Spark执行器选择高效的托管节点，以缩短预留时间。它旨在及时满足执行者对资源的高要求。然后，我们设计并开发一种依赖关系感知的资源调整机制，以自适应地控制减少任务未充分利用的资源。它允许地图任务从资源减少任务中抢占资源。我们进一步设计和开发了一种跨平台的任务分配机制，以协调MapReduce任务分配和Spark执行程序放置之间的局部性。它旨在增加共同托管Spark和MapReduce应用程序的节点上的本地数据访问机会。

我们在16节点Hadoop YARN群集上实施iKayak，并使用Pur Due MapReduce基准套件（PUMA）和“ Big-DataBench”基准以及从真实应用程序收集的数据集评估其优势。 我们将iKayak的性能与YARN两种流行的Spark-on-YARN部署模型进行了比较：YARN客户端模型和YARN集群模型。 通过运行不同的工作负载的实验结果表明，与两种流行的模型相比，iKayak分别将Spark工作负载的作业完成时间减少了50％和30％。 同时，与这两种模型相比，它还将MapReduce工作负载的作业完成时间减少了14％和19％，同时将CPU利用率分别提高了22％和15％。

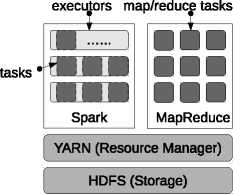


图1 [Spark+MapReduce]-on-YARN

在本文的其余部分，第2节提供了YARN上Spark的案例研究和动机。 第3节介绍了iKayak的设计和开发。 第4节提供了实现细节。 第5节介绍了实验结果。 第6节回顾了相关工作。 第7节总结了论文。

**2 动机**

**2.1 背景**

YARN中基于预留的计划。 与第一代Hadoop相比，YARN采用了细粒度的资源管理，这意味着应用程序可以在提交作业时针对单个任务配置其所需的资源（如CPU和内存）。 如果集群中没有足够的可用资源，则ResourceManager将开始在所选节点上为任务保留资源。 当此节点上的其他任务完成时，可以将更多可用资源分配给即将执行的任务。 在满足保留之前，该节点上已释放的资源将不允许分配给其他应用程序。 在预留过程中，新释放的资源被累积但不允许使用，这提供了良好的资源隔离能力，但导致资源利用效率低下。

使用Spark进行内存中计算。 Spark是一种通过使用高效的内存计算开发的新兴的大数据分析解决方案。 它允许应用程序在内存中显式缓存数据集，以便应用程序可以从内存而不是磁盘访问数据，这可以显着提高性能。 与MapReduce相比，Spark利用多个线程而不是多个进程来在单个节点上实现并行性，从而避免了多个JVM开销的内存，但导致了单个执行者一次占用大量资源。 当作业提交到集群时，特定的Spark应用程序的资源需求（即执行者的数量和每个执行者的资源请求）应由用户配置。 Spark可以在两种流行的模式下运行：Mesos上的Spark和YARN上的Spark。 Mesos上的Spark [5]采用“全有或全无”资源管理策略（例如，一次分配足够的资源或拒绝请求），这可能会导致饥饿，尤其是在群集繁忙时。 YARN将保留资源，而不是拒绝应用程序以避免此类饥饿。

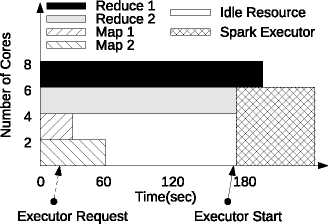
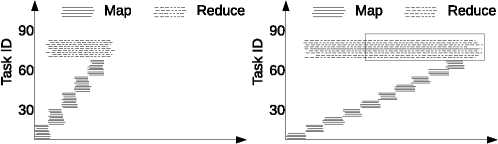


图2 资源利用效率低下

YARN上的Spark。 在YARN上的Spark部署模式下，Spark应用程序类似于MapReduce“作业”。 如图1所示，MapReduce在其自己的进程中运行每个任务。 任务完成后，该过程将消失。 在Spark中，许多任务可以在单个进程（即执行程序）中同时运行，并且即使没有作业在运行，该进程也会在Spark应用程序的整个生命周期内持续存在。 因此，MapReduce不会受到基于预留的调度策略的困扰，因为大多数map / reduce任务的预留周期由于资源需求小而非常短。 但是，在YARN的当前资源调度策略下，对单个容器具有巨大资源需求的Spark应用程序可能会因资源保留期过长而受苦。

**2.2 案例研究与挑战**

尽管YARN上的Spark提供了与YARN的群集范围内的资源管理策略的良好集成，但是由于两种编程模式的不同特性，它还是带来了巨大的挑战。为了说明由混合部署引起的效率低下，我们对（Spark + MapReduce）on YARN集群（由六台机器组成）进行了案例研究。我们为每个从属节点配置了8个内核和12 GB内存，并且在实验中将块大小设置为256MB。在实验中，使用30 GB的输入数据执行PUMA基准测试[6]中的代表性MapReduce应用程序，即WordCount。另一个有代表性的Spark应用程序，即Logistic回归[7]，是使用来自Wikipedia的10 GB输入数据执行的。 map和reduce任务的资源需求配置为2核和2GB内存。单个执行器的资源需求配置为6核和8GB内存。我们测量了各种应用程序的不同任务完成时间，并观察了混合平台上的资源利用情况，如下所示。



0 120 240 360

Time(sec)

0 120 240 360

Time(sec)

(a) MapReduce + Spark

(a) MapReduce Only

图3

**资源利用效率低下** 图2显示了短时间窗口内从属节点上的资源调度图的详细信息。当在20秒提交Spark执行器请求时，ResourceManager会发现节点上没有足够的可用资源。然后ResourceManager开始为提交的Spark应用程序保留资源，并拒绝将其他MapReduce任务分配给所选节点。图2显示完成Map 1和Map 2后存在大量空闲资源。这是因为Map 1和Map 2释放的资源（即4个核心）小于执行器的需求（即6个核心）。当节点上的资源积累满足Spark执行器的需求时，执行器最终开始在节点上执行，直到Reduce 2完成。在此资源调度过程中，我们发现基于预留的资源分配策略显然无效，并且导致资源利用率低，从而导致Spark应用程序的性能也很差。如果在选定的节点上出现散乱的任务（通常会长时间占用资源），则这种无效的资源调度可能会变得更糟。

**未充分利用的简化任务**  在第二种情况下，我们首先单独运行MapReduce应用程序，然后在集群上一起运行MapReduce和Spark应用程序。 图3比较了两种不同场景下MapReduce应用程序的任务执行差异。 它显示了混合部署显着增加了MapReduce应用程序的地图波形数（即从5个波形增加到10个波形），并延迟了减少任务（从120s到300s）。 这是由于与单独的MapReduce部署相比，集群资源的一半分配给了Spark应用程序。 在这种情况下，我们发现，reduce任务完成相同任务时，其资源占用时间几乎是唯一MapReduce部署所花费时间的三倍。 显然，为混合环境中的缩减任务预留的资源超出了其需求，因此导致缩减任务的利用率大大降低。当将reduce任务配置为以多波形式完成时，这种现象甚至更糟。

**缺少地区的分配** 我们还观察到MapReduce和Spark执行中任务分配的局部性意识。我们发现，当MapReduce应用程序仅在专用群集中运行时，具有数据局部性的地图任务可完成已完成地图任务总数的95％。但是，当MapReduce应用程序与Spark应用程序共同托管时，本地地图任务率降低到68％。这是由于Spark执行程序通常在各个节点上占用大量资源这一事实，从而极大地阻碍了MapReduce应用程序对位置感知的地图任务访问。例如，集群中有两个节点（即M1和M2），每个节点有4个核心。 Spark执行程序需要3个核心，每个映射任务需要1个核心。 M1和M2节点上的数据存储互不相同，因此映射任务可以分为两种类型，即M1本地任务和M2本地任务。当Spark执行程序在节点M1上占用一个大容器（即3个核心）时，只有1个核心可用于MapReduce任务分配。对于地图任务，M2节点上的空闲资源（即4个核心）比M1节点上的空闲资源（即1个核心）更多。由于MapReduce采用“基于拉”的任务分配策略，一些M1-本地任务被调度到M2节点，在这种情况下，由于Spark的竞争，在M2节点上分配的M1-本地任务错过了局部性意识应用。

HDFS通常在群集中具有三个数据副本，这可以改善数据的局部性。 例如，可能存在另一个节点，该节点具有与M1节点相同的数据文件，这为M1本地任务提供了具有局部性的替代项。 但是，随着群集大小和应用程序数量的增加，此类多副本仍然太受限制，难以避免此类局部性缺失的任务分配。 特别是，Spark和MapReduce都应用了“延迟调度” [8]来基于位置偏好来调度任务，即它们尝试将任务调度到具有本地数据的节点上。 因此，即使集群中有多个副本，Spark和MapReduce之间的这种局部性意识竞争也会使他们俩都遭受缺乏局部性意识的困扰。

**2.2 机会**

YARN的新功能（即基于预留的资源调度）由于具有细粒度的资源管理功能，非常适合处理小规模的MapReduce任务。 但是，在YARN上运行Spark会导致性能显着下降（例如，资源利用效率低下，依赖项盲资源分配和局域性任务分配不足），这是由于其特殊特性，即单个执行者的巨大资源需求。 虽然Spark-on-YARN提出了上述挑战，但也带来了新的机遇。 直观地，我们发现三种策略分别应对上述三个挑战。

•Spark执行器应放置在适当的节点上，以便及时满足其资源预留，并避免等待时间太长和资源利用率低。  
 •为减少任务而分配的资源应根据其随时间变化的需求进行重新平衡，以避免在将任何Spark应用程序提交到集群时浪费资源。  
 •应该优化Spark和MapRe duce应用程序的任务分配，以减少对位置感知的竞争，同时考虑Spark和MapReduce应用程序的位置感知。

Spark 11 MapReduce  
**jKayak <**

Dependency-aware  
resource adjustment

Locality-aware assignment coordination

YARN (ResourceManager)

图4 iKayak概述

所有这些激励我们开发一种整体资源调度方法，以提高YARN上Spark的群集利用率和应用程序性能。 在下一部分中，我们将重点介绍利用这些机会来优化YARN上Spark的资源调度。

**3 iKAYAK设计**

iKayak是跨平台的资源调度中间件，旨在优化YARN上Spark集群中的资源管理。 它利用了Spark和MapReduce应用程序的不同资源需求特征，并在考虑两个应用程序的数据局部性的同时动态优化了资源调度。 图4显示了iKayak的体系结构。 它具有三个主要组件：预留感知的执行器放置，依赖项感知的资源调整和本地性任务分配协调。 我们简要描述它们的主要功能。

•预留意识的执行器位置自适应地将Spark执行器放置在高效的托管节点上，该节点可以及时满足各个执行器的高资源需求。  
 •依赖关系的资源调整动态方法利用分配给减少任务的资源来缓解由于资源过度配置而导致的资源利用率低下的问题，尤其是当减少任务处于空闲状态并等待来自映射任务的中间数据时。  
 •位置感知任务分配协调im证明了Spark和MapReduce应用程序的数据位置感知能力，因此它们可以在多租户群集节点上共享有限的本地数据访问机会。

**3.1预留感知的执行器放置**

将Spark作业提交给集群时，执行程序的位置对于应用程序性能和集群利用率至关重要。我们提出并开发了一种预留感知的执行器放置机制，以缓解由于Spark的高资源需求而导致的资源利用效率低下的问题。关键的见解是，Spark执行程序应放置在具有足够可用资源或承载小型地图任务的节点上。请注意，reduce任务通常比map任务占用更长的时间。如果执行者必须等待缩减任务释放的资源，则应将其放置在托管缩减任务即将完成的节点上。

Spark提供了两个默认的执行程序放置调度程序，即SpreadOut和Non-^^ readOut [9]，用于将执行程序放置在集群中的多个节点上。它们都没有能力以资源预留感知方式放置执行器。在YARN上运行Spark时，每个Spark执行程序都在YARN容器中运行。 Spark ApplicationMaster负责与YARN协商资源请求，并找到一组有效的主机来放置和运行执行程序。但是，YARN无法识别Spark应用程序的有效工作程序节点，因为它最初是为MapReduce计算而设计的。 MapReduce计划一个容器并为每个任务启动一个JVM，而Spark在同一容器内托管多个任务以进行多线程处理。相应地，Spark的单个容器的资源需求远远大于MapReduce的单个任务的资源需求。因此，MapReduce作业的默认基于“拉动”的任务调度方法不适用于Spark应用程序的执行器放置。

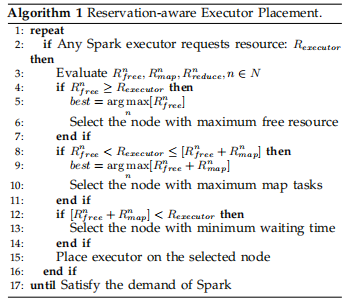
**3.1.1资源预留分析**

为Spark执行程序保留的资源可能来自选定节点n上的三个来源，即可用自由资源（Rfg），运行映射任务（R ^ ap）和运行归约任务（R ^ educe）。我们定义了一个高效的辅助节点来托管Spark应用程序，该节点可以及时满足Spark各个执行者的资源需求。我们分析以下三种来源的特征。

•保留后，所选节点上的免费资源可立即用于执行者，这是保留执行者资源的最有效资源。

•来自运行地图任务的资源可以在很短的时间内（即几秒钟或几分钟）用于执行者。这是因为地图任务重量轻且通常很小。它们将在完成后释放占用的资源。

•来自运行reduce任务的资源可以在不确定的时间内用于执行程序，这取决于reduce任务的执行进度。



算法1

我们基于集群中每个可能的目标节点的资源预留意识，开发了一种有效的执行程序放置算法。该算法旨在使执行程序一旦放置在选定的工作程序节点上，便尽快运行。如算法1所示，它首先选择具有足够可用资源的高效工作程序节点，以满足各个执行者的需求。如果节点上没有足够的可用资源，则iKayak会选择高效的工作程序节点，该节点承载足够的运行映射任务来放置执行程序。如果仍然没有足够的资源，iKayak最终会选择一个节点，该节点托管减少任务即将完成的任务。

前两个场景很简单，因此我们将重点放在第三个场景的分析上。当集群中的大多数MapReduce工作负载是减少负载的工作负载时，Spark执行者可能不得不等待reduce任务释放的资源来满足他们的需求。在这种情况下，iKayak的目标是查找具有减少任务即将完成的目标节点，从而使Spark应用程序不必等待很长时间。不能简单地将Spark执行程序分配给由reduce任务释放的第一个资源插槽，因为不能保证该机器上的其他reduce任务可以在短时间内完成。为了找到哪些还原任务即将完成，有必要预测还原任务的执行进度和剩余时间。

**3.1.2估算减少剩余时间**

尽管有许多现有的MapReduce执行预测方法，但我们开发了一种自适应模糊模型，可根据任务的输入大小和资源分配来预测化简任务的剩余执行时间。 模糊模型通常用于捕获资源分配和任务的细粒度执行进度之间的复杂关系。 但是，任务的进度可能受许多因素影响。 首先，减少任务进度在不同的执行阶段是不一致的，例如，shuffle，sort 和reduce 阶段。 其次，即使在同一阶段，任务之间的数据偏斜也会导致在不同的时间间隔执行不同的ent任务。 最后，共同运行的任务可能会意外地干扰任务的执行，从而使资源到任务进度的映射变得可变。 许多现有的预测方法并未考虑来自其他共存应用程序的多租户干扰。 因此，我们设计了基于实时测量的在线自适应模糊模型。

**模糊模型** 控制间隔t中的减少任务剩余执行时间（即y（t））表示为输入输出NARX类型（具有异源输入的非线性自动回归模型）

捕获1

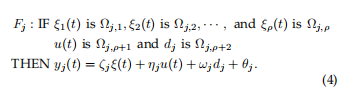
F是输入变量和输出变量之间的关系。 输入变量是当前资源分配u（t），作业输入大小d和回归向量g（t）。 在此，u（t）表示归约任务的资源分配。 回归向量g（t）包含多个先前控制周期的滞后输出和输入。 它表示为

捕获2

其中ny和nu分别是输出和输入的滞后值的数量。 令p表示回归向量g（t）中的元素数，即

捕获3

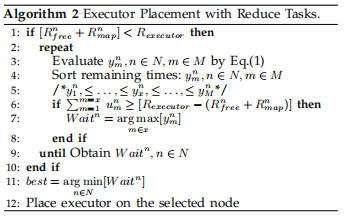
F是由高木Sugeno规则组成的基于规则的模糊模型[10]。 规则Fj表示为



在此，Qj是第j个规则的先行模糊集，它由一系列子集组成：Qj，i，Qj，2，•••，Qj，p + 2。 Zj，nj和3j是参数，Bj是偏移量。 他们的价值观是通过离线培训获得的。 每个模糊规则表示特定还原任务类型的分配资源与性能之间的非线性关系。

**在线自学** 由于MapRe对工作行为的动态影响（例如数据偏斜，不同阶段和多租户干扰），我们设计了一个在线自学习模块来适应模糊模型。 目的是使模糊模型e（t）的预测误差最小，该误差是实际测量的作业进度与预测值之间的误差。

如果e（t）= 0，则我们应用递归最小二乘（RLS）方法来适应当前模糊规则的参数。 当从运行时系统中采样新的测量值时，该技术会更新模型参数



算法2

我们将等式（1）中的模糊模型输出表示为

捕获5

其中e（t）是实际输出之间的误差和预测的输出。由模型参数组成的向量。 X = [σ1X（t），σ2X（t），..，σρX（t）]其中σj是归一化的第j条规则的满足程度或解雇强度X（t）= [ξ（t）T，u（t）]。 对参数向量φ（t）进行估计，以使等式（6）中的误差函数最小。我们同时应用当前误差e（t）和先前误差

误差e（t 1）来估计参数向量。

捕获6

τ称为折现因子，因为它给出更高的优化中最近样本的权重。 它确定当前预测误差的方式和旧错误会影响参数估算值的更新。

**3.1.3执行带有简化任务的执行器放置**

根据不同节点上约简任务的估计剩余时间，我们使用算法2选择执行程序放置的托管节点。 如算法2所示，我们假设约简任务m（m e M）在节点n（n e N）上运行。 我们首先根据等式（1）评估每个节点上的reduce任务的剩余执行时间（即y ^）。 然后，我们对每个节点中约简任务的剩余执行时间进行排序，并从第n个节点上的约简任务计算最小的累积资源（即Ek二X um），以满足执行者的需求。 我们获得相应的估计等待时间（Waitn），以累积来自第n个节点的所需资源。 最后，放置执行程序的最佳人选是等待时间最少的节点。

**3.2依赖关系的资源调整**

我们在两个级别（即应用程序级别和任务级别）上确定Spark-on-YARN中的依赖关系。 在应用程序级别上，存在明显的依赖性，即多租户集群中Spark和MapReduce应用程序之间的资源竞争。我们测量了各种MapReduce作业的完成时间和吞吐量，同时在Spark和MapReduce应用程序之间提供了不同的输入数据比例。在集群上运行了来自PUMA基准[6]的三个MapReduce应用程序，即WordCount，Terasort和Grep，它们具有各种输入数据大小。 BigDataBench [7]基准测试的另外三个代表性Spark应用程序，即K-means，PageRank和Index，以及来自Wikipedia的输入数据集，也都在集群上运行。通过将各种作业完成时间与具有足够资源（即100％）的专用集群中实现的作业完成时间进行比较，我们量化了MapReduce作业执行的速度。图5显示，随着集群中工作负载比例的降低，MapReduce的执行速度降低。它还显示，随着集群中MapReduce工作负载的减少，已完成的地图任务的吞吐量将大大降低。结果，在任务级别，由于映射任务和归约任务之间的依赖性，延迟的映射任务执行进一步影响了归约任务的资源需求。我们开发了一种依赖关系感知的资源调整机制，以动态控制分配给减少任务的资源。

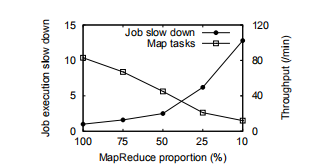


图 5 工作比例的影响

我们想利用分配的资源来减少MapReduce作业之后的任务开始执行时，分配给MapReduce工作负载的集群资源发生变化，因为在其中共同托管了Spark应用程序集群。 我们定义一个延迟因子λ= RSpark/（RSpark + RMR）表示导致地图任务执行的延迟通过多租户中的动态资源分配集群环境。 在此，RSpark和RMR代表分别用于Spark和MapReduce应用程序的资源分配。 延迟因素是事件驱动的通过申请提交。

当分配任务为资源时，我们使用JVM暂停功能来暂停reduce任务的执行。由于地图任务的延迟而未充分利用执行。悬挂时间长短取决于延迟因子λ。令T代表控制间隔。的悬挂长度的计算公式为：Tsubspend =λv×T，其中v是权重参数。较大的λ表示存在Spark应用程序占用的资源比由MapReduce应用程序占用，因此减少任务的暂停时间应更长。在此期间暂停期，减少任务执行被暂停并且释放了他们的CPU资源。内存资源由于相关数据无法立即发布的过程仍然保留在内存中以避免数据丢失当还原任务恢复时。注意相关可以将挂起的reduce JVM的数据换出由具有开销的操作系统[11]。因此，我们使用Tsuspend = [λv-θ]×T确定悬架长度，其中θ是代表暂停开销。

当作业的简化任务暂停时，iKayak利用释放的资源来分配地图任务节点执行相同的任务以提高资源利用率。首先，地图产生的更多中间数据任务可以用来加速减少任务，因为减少任务需要中间数据才能执行。 其次，许多地图任务占用大量CPU，因此资源暂停的reduce任务可以释放更有效地映射任务。

**3.3位置感知分配协调**

Spark和MapReduce应用程序都希望分配执行程序和任务及其数据块，以实现本地数据访问。从两个方面来看，Spark应用程序对数据局部性都很贪婪。首先，执行者会在应用程序的整个生命周期内坚持不懈，即使没有作业在运行。其次，与MapReduce任务不同，单个Spark执行程序通常在选定的工作程序节点上占用大量资源。相应地，在共同托管Spark执行程序和MapReduce任务时，经常会出现局部干扰。

如果没有足够的内存来存储作业的数据，Spark的性能将大大降低[1]。因此，与MapReduce任务相比，Spark执行程序的资源配置更加不灵活。由于地图任务比归约任务对数据局部性更为敏感，因此iKayak专注于MapReduce工作负载的地图任务的局部性协调。当本地数据访问机会仅限于地图任务时，ResourceManager必须在允许远程数据访问和分配比其需求少但具有本地数据访问权限的资源之间做出选择。例如，M1节点上有2个CPU内核和2GB内存可用。每个MapReduce任务的资源需求配置为2个cpu核心和2GB内存。目前必须分配两个M1本地地图任务。任务分配有两种潜在的解决方案。解决方案a是在节点M1上仅分配一个本地映射任务，而允许在节点M2上分配另一任务而没有数据局部性。解决方案b是将两个映射任务的资源需求缩减到（1 cpu内核，1GB内存），然后可以在具有本地数据访问权限的M1节点上分配它们。相应地，两个地图任务的容器在初始化时必须按比例缩小。但是，减少任务的资源分配肯定会降低性能。

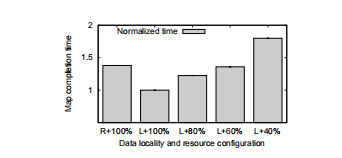
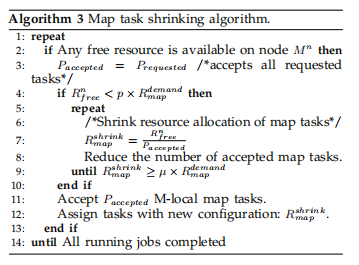


图6 数据局部性和资源配置

我们探索数据局部性对性能的影响和个别地图任务的资源配置。 图6比较了地图任务完成时间远程数据访问和本地访问数据访问和不同的资源配置。 在本案例研究中，地图任务配置了2个核心和2 GB的内存资源需求（表示为<2core，2G>）。 “ R + 100％”表示地图任务依赖于远程数据访问并具有100％的资源，即<2core，2G>。“ L + 100％”表示地图任务依赖于本地数据访问，拥有100％的资源，即<2core，2G>。 它表明具有100％资源访问本地数据的地图任务配置具有最短的完成时间。 它也是展示了地图任务（即L + 80％）资源配置超出其需求，但具有本地性数据访问比地图任务具有更好的性能（即，R + 100％）有所需的全部资源，但必须进行远程数据访问。



算法 3

为了协调当地的意识，我们开发了一个iKayak的按任务配置方法[12]，它可以定位不同数量的资源来映射任务维护相同的资源配置所有地图任务的容器。如算法3所示，iKayak会定期检查每个资源的状态节点通过心跳连接。如果节点中有任何可用的免费资源，则ResourceManager会分配将具有数据局部性的任务映射到节点。假设在那里是请求的M本地地图任务分配通过ApplicationManager。节点M具有免费的Rf资源，每个任务都请求Rdemand地图资源。在一开始，iKayak接受所有地图任务。如果有没有足够的资源来满足所有的地图任务此刻，iKayak逐渐减少了接受任务并缩小资源配置个别任务。最终，iKayak接受了收缩后的节点M上的M局部地图任务资源配置Rshrink地图。为了避免性能恶化，我们设置收缩率μ以防止iKayak从缩减资源配置个别任务太多。地图的收缩率μ任务定义为µ = Nlocal\（Nlocal + Nremote），其中Nlocal和Nremote代表本地和远程地图的数量分别访问任务。

**4实施**

**4.1YARN修改**

我们通过基于Hadoop版本2.6.0修改类ResourceManager，ApplicationManager和LaunchTaskAction来实现iKayak。我们添加了一个新的exePlace接口，以实现可识别预留的执行程序放置算法。当应用程序将其注册请求提交给Spark的Ap ^ licationMaster时，它将调用exePlace方法为执行者分配资源。 exePlace方法负责将可用资源分配给应用程序，这由任何应用程序提交或群集资源更改由事件驱动。此外，我们添加了另一个新接口taskRes，该接口用于指定各个任务的资源配置，同时将其分配给从属节点。每个执行程序放置和任务资源分配都用其相应的AttemptTaskID进行标记。在作业执行期间，我们使用TaskCounter和TaskReport创建了一种方法taskAnalyzer来收集已完成任务的状态。 TaskTrackers通过心跳连接定期报告任务的资源利用率和执行时间

**4.2实验设置**

我们在由3个T110（8核CPU和16 GB RAM），2个T420（24核CPU和32 GB RAM），1个T320（12核CPU和24 GB RAM），2个T620（ 24核CPU和32 GB RAM）和8个Dell台式机（8核CPU和16 GB RAM）。 每台计算机都有1 TB硬盘。 主节点托管在群集中的一个Dell桌面上。 服务器与千兆以太网连接。 HDFS的块大小设置为256 MB，HDFS的副本数设置为3。iKayak基于Hadoop实现的2.6.0版本和Spark实现的1.4.0版本。 我们在集群的主节点上实现iKayak。 暂停控制间隔（T）设置为5分钟，这是制图任务与缩小任务之间的权衡

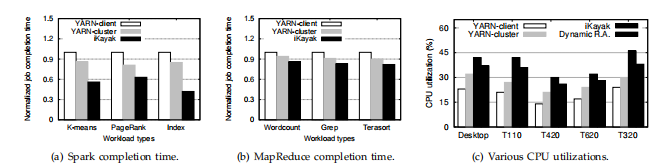
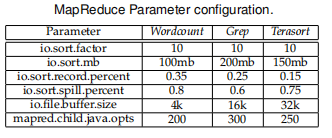


图 7 iKayak对Spark，MapReduce和群集资源利用的有效性

任务完成时间。 在实验中，根据经验确定悬架参数v和开销（θ）为1和0.15。

表 1



**4.3 实际工作量**

为了了解iKayak在生产环境中的有效性，我们使用了合成的工作负载“ MicroSoft-Derived（MSD）”，该工作负载在2011年的一个月中为Microsoft数据中心的174,000个工作的生产工作负载建模[13] 作为MapReduce应用程序，MSD通过运行PUMA基准测试[6]中的Wordcount，Terasort和Grep应用程序（具有各种输入数据大小）来模拟生产作业的分布特征，它是[13]中研究的工作负载的缩减版本。 ），因为我们的集群要小得多。我们通过两种方式缩减工作量：将工作总数减少到87，并消除最大的10％的工作和最小的20％的工作使用了一组来自“ BigDataBench [7]”基准，即K-means，PageRank和索引，以及来自Wikipedia的输入数据集。我们混合MapReduce工作负载和Spark工作负载实验中1：1的比例，这是基于来自行业的工作量分析[14]，[2]。如表1所示，我们设置了Hadoop配置根据Cloudera [15]建议的规则。我们使用相同的配置来评估各种实验中的方法。在YARN中，没有“广告位”是旧版本的基本组成部分，而且系统不再区分地图和缩小分配资源时的任务。而是，每个任务以<1core，2GB的形式指定资源请求。（即请求1个CPU内核和2GB内存），它将被分配给具有足够容量的节点。我们计算出MapReduce工作负载的资源需求（如表2所示）和Spark工作负载（如所示）在表3中）基于先前的研究[16]，[1]和我们的实验经验。

**5实验评估**

**5.1 iKayak的有效性**

我们将iKayak的性能与在YARN群集上运行Spark的两种代表性的部署模型进行了比较[17]：YARN群集模型和YARN客户端模型。减少Spark作业完成时间。 图7（a）显示，与YARN客户端模型和YARN群集模型相比，iKayak分别将K-means，PageRank和Index工作负载的工作完成时间显着缩短了约50％和30％。 由于两个因素，Spark作业的性能优于MapReduce作业。 首先，由于Spark是内存计算应用程序，因此Spark作业比MapReduce作业对数据位置更加敏感。 如果没有足够的内存来存储作业数据，Spark的性能将大大降低。 因此，我们的工作重点是为MapReduce工作负载缩减映射任务的资源配置，同时为Spark应用程序保留足够的资源。 其次，它受益于iKayak的功能，该功能可以根据保留时间意识自适应搜索高效的工作程序节点以放置执行程序。 结果还表明，iKayak对于CPU更有效密集型工作负载（即K均值和索引）比I / O密集型工作负载（即PageRank）高。 这是由于群集中机器的内存配置受到相对限制的事实。 I / O密集型工作负载的预留时间比CPU密集型工作负载的预留时间更长。

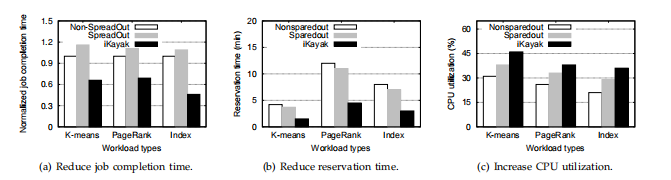


图8 Spark应用程序的预留感知执行程序放置的有效性

减少MapReduce作业完成时间。图7（b）显示，与YARN客户端模式和YARN群集模型相比，iKayak分别将MSD工作负载的总体作业完成时间缩短了14％和19％。这是由于以下事实：iKayak以任务相关性和位置感知方式在MapReduce应用程序的资源管理和任务调度方面更加灵活。另一方面，图7（b）显示YARN-client模型比YARN-cluster模型具有更好的性能。这是因为YARN客户端模型一次仅托管一个Spark应用程序。与YARN群集模型相比，它为MapReduce应用程序分配了更多的群集资源增加群集资源利用率。图7（c）显示了由于不同的群集资源调度方法而导致的各种类型计算机的CPU利用率。它表明，与YARN客户端模型和YARN集群模型相比，iKayak分别将群集中所有类型计算机的总体CPU利用率提高了22％和15％。请注意，与群集中的其他机器相比，iKayak在T420和T620机器上实现的利用率较低。这是由于其相对较低的内存资源可用性和强大的CPU。因此，难以充分利用CPU，特别是用于存储器密集型计算。我们进一步比较了Spark的动态资源分配（Dynamic R.A.）策略和建议的iKayak方法下的CPU利用率。与YARN客户端模型和YARN集群模型相比，粗粒度动态资源分配策略分别将CPU利用率提高了16％和9％。结果表明，iKayak的细粒度资源分配比动态分配策略提供更高的资源利用率。

**5.2执行安置的好处**

图8比较了在群集上运行三个Spark工作负载时，不同执行程序放置策略实例（即SpreadOut，Nons-preadout和iKayak）实现的作业完成时间和CPU利用率。结果表明，所提出的预留感知执行器放置方法显着减少了作业完成时间和Spark作业的预留时间，同时提高了主机的CPU利用率，减少工作完成时间。图8（a）显示，与NonspreadOut策略和Spread-Out策略相比，iKayak显着减少了K-means，PageRank和Index工作负载的作业完成时间，分别高达55％和37％。它可以自动选择目标计算机来托管Spark执行程序，目的是避免不必要的等待时间来预留资源。特别是，预留感知执行程序放置机制有效地减少了将执行程序放置在承载长期运行的reduce任务的工作程序节点上的机会。结果还表明，就工作完成时间而言，NonspreadOut方法比SpreadOut策略具有更好的性能。这是由于以下事实：NonspreadOut鼓励将执行程序放置在群集中的一部分计算机上，以便减少中间数据流量以提高性能。减少预订时间。图8（b）显示，与NonspreadOut策略和SpreadOut策略相比，iKayak的预留感知执行器放置机制显着减少了K-mean，PageRank和Index工作负载的预留时间，分别高达67％和53％。它还说明，资源需求较少（即K均值）的执行者的保留时间显然比资源需求较多（即PageRank和Index）的执行者的保留时间短。这是由于以下事实：通过YARN中当前基于预留的资源管理方案可以及时满足较小的资源需求。结果还显示，就预留时间而言，SpreadOut方法比NonspreadOut策略具有更好的性能。原因是，与SpreadOut相比，NonspreadOut的集中部署导致更多执行程序与减少任务共同托管，从而导致等待资源预留的时间较长。

增加CPU利用率。图8（c）显示执行器放置机制稍微增加了CPU这三种不同工作负载的利用率（即10％），因为大多数Spark工作负载是占用大量内存的应用程序，并且对CPU资源的影响有限。 另一方面，SpreadOut比NonspreadOut具有更高的CPU利用率。 这是由于SpreadOut策略试图将整个执行器分散到整个群集中。 它使执行者位于群集中的大多数节点上，这导致CPU利用率高于NonspreadOut。



图 9 任务剩余时间估计的准确性

**5.3剩余时间估计的准确性**

iKayak应用模糊模型来估计减少任务剩余时间，同时为Spark执行程序选择高效的托管计算机。为了评估模糊模型的准确性，我们比较了预计的减少任务剩余时间与实际剩余时间之间的误差。准确度是通过标准化的均方根误差（NRMSE）（偏差的标准度量）来衡量的。来自PUMA基准测试[6]的三个MapReduce应用程序（即Terasort，Grep和WordCount）分别具有300 GB的输入数据，它们在集群中的四台不同计算机（即T420，T320，T620和T110）上运行。因此，我们记录了12个化简任务的实际剩余时间和预测剩余时间，如图9（a）所示。结果表明，该预测非常准确，NRMSE平均为7.8％。图9（b）显示，与iKayak的真实预测精度所获得的执行时间相比，随着剩余时间预测误差从10％增加到35％，Spark作业的执行时间从7.5％增加到34％。这是由于以下事实：不正确的剩余时间估计可能会导致额外的Spark执行程序等待时间来预留资源。该观察结果确认了预留意识的执行器放置以及减少任务剩余时间的估计精度对于减少Spark作业执行时间和执行器放置等待时间至关重要。

**5.4资源调整的好处**

图10描述了iKayak和库存YARN在实验中实现的MapReduce任务的吞吐量。 除了资源调整机制，我们将关闭iKayak的其他功能。 结果表明，iKayak的资源调整机制有效地提高了地图任务的吞吐量，

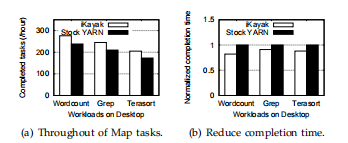


图 10 减少调整的效力

略微减少了reduce任务的完成时间。 图10（a）显示，与桌面YARN相比，台式机上的映射任务的吞吐量显着提高了约17％，这是因为依赖关系感知的资源调整机制为映射执行提供了更多资源。 当没有足够的中间数据要处理时，资源调整机制会定期挂起reduce任务，并将冗余资源释放给相关的映射任务。

减少挂起可为相应的映射任务提供更多的群集资源。 它还可以为它们的归约任务生成更多的中间数据。 因此，在这种情况下，挂起化简任务确实会加速这些化简任务的执行。 图10（b）表明，与YARN在台式机上的库存相比，挂起策略已将减少的完成时间减少了8％。 这是由于以下事实：分配给减少任务的资源是过度配置的，如果没有足够的中间数据要提供，则该资源将处于空闲状态。

**5.5地方协调的好处**

图11表明，所提出的位置感知任务分配协调模块有效地提高了MapReduce任务的本地数据访问率，从而减少了作业完成时间。图11（a）比较了在实验中通过iKayak和YARN-cluster模型的位置感知协调实现的地图任务的本地数据访问率。结果表明，与YARN集群模型相比，iKayak将工作负载Wordcount，Grep和Terasort的本地数据访问量提高了27％。这是由于以下事实：当MapReduce与Spark应用程序竞争本地性意识时，协调机制会强制使用本地数据访问权在托管节点上分配要分配的地图任务。因此，协调机制比YARN群集部署允许更多的本地映射执行。

如前所述，缩小地图任务的资源分配以实现本地数据访问会降低单个地图任务的性能。图11（b）显示，应用位置感知协调器后，平均地图任务完成时间略微增加了9％。但是，单个任务的性能下降会带来更多机会并行运行地图任务（即，增加地图“位置”）。缩小某些任务以偏爱数据局部性时，会有更多资源

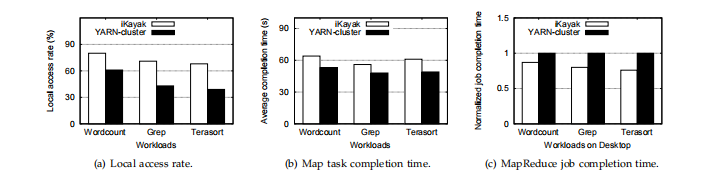


图 11 跨平台位置感知分配协调（LAC）的影响

群集中其他从属计算机上的磁盘将被释放。 这些可用资源可用于并行运行更多其他任务，而无需缩减任务。 映射执行的并行性增加，不仅可以补偿缩小的映射任务的性能损失，而且可以显着提高作业级别的性能。 整个作业的执行进度将相应地加快。 图11（c）展示了在应用本地感知任务分配协调机制时，Wordcount，Grep和Terasort的工作完成时间得到有效减少。

**5.6参数和配置的影响**

我们更改悬架重量参数v和开销B的值，以研究它们对工作完成时间对性能改进的影响。图12（a）显示，随着悬架参数v的增加，作业完成时间最初会减少。但是，增加v会进一步导致性能下降。这表明很大的悬挂参数v可能导致作业完成时间变差。大的开销B（例如2）由于悬架控制的不稳定性而导致显着的性能下降。因此，我们在实验中根据经验将悬架参数v设置为1。图12（b）显示，调整悬挂开销B与调整参数v具有相似的现象。我们根据经验将开销B设置为0.15，而不会影响系统稳定性。这是在减少任务暂停时间成本和完成工作时间之间进行权衡的方法。

我们将在实验中的各种配置下（即应用提前启动和延迟调度）进一步探讨所提出的iKayak的有效性。我们首先比较了不同的早期启动百分比下的性能改进。图13（a）显示了在不同的方法（即iKayak和原始YARN）下，随着我们更改了提前开始百分比的值而改变了工作完成时间。结果表明，在调整早期百分比时，iKayak在工作完成时间方面优于原始YARN约7％。它表明，紧张工作负载（例如terasort）中的IO倾向于使用较小的早期启动百分比，因为可以直接整理来自映射任务的中间数据以节省带宽。非IO密集型工作负载（例如，字数统计）倾向于较大的提早启动百分比，因为这样可以防止还原任务占用还原插槽而不执行。然后我们比较了本地数据访问率，同时分别在MapReduce和Spark上应用了不同的调度策略（即延迟（D）和无延迟（N））。图13（b）中的结果表明，如果仅采用一个应用程序（即Spark或MapReduce），则延迟调度会将本地访问率提高12％-15％。但是，当Spark和MapReduce都应用延迟调度时，本地访问速率与它们都不应用延迟调度的情况类似。

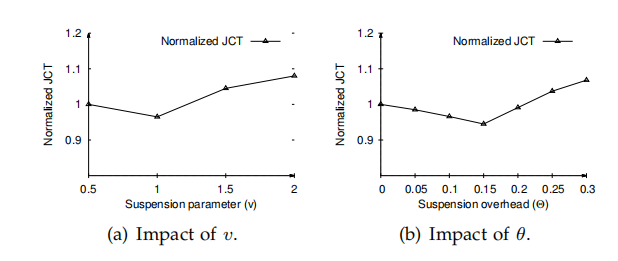


图12 悬架参数调整的灵敏度

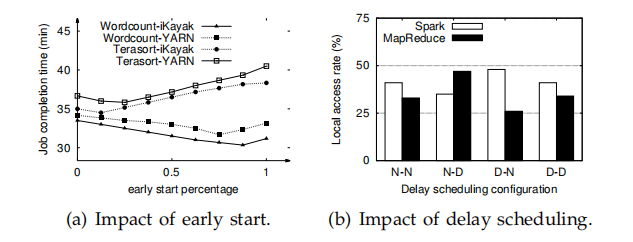


图 13 提前启动和延迟调度的影响

**5.7 iKayak的开销和可伸缩性**

iKayak的开销主要来自执行执行程序放置算法所需的时间，挂起和恢复reduce任务所需的时间以及重新配置各个Map任务所需的时间。 实验结果表明，执行器放置算法的开销在120毫秒至150毫秒之间ms，与Spark作业执行时间相比非常小。 我们还测量每个控制间隔中的暂停开销。 iKayak平均花费2.7秒暂停和恢复还原任务。 与5分钟的控制间隔相比，开销相对可以忽略不计。 最后，任务级重新配置的开销非常稳定（即45-55 ms），并且与测试平台的大小无关。 iKayak具有可伸缩性，适用于大规模集群。 YARN组件的修订不会影响其他计算系统。

**6相关工作**

大数据资源管理：现代大数据集群应用了各种资源管理器组合，例如YARN，Corona，Omega和Mesos。这些不同的计算框架具有内在的调度需求。 YARN [3]，第二代Hadoop，在Hadoop中添加了资源管理层。它允许为不同的应用程序分配不同数量的任务容器。 Corona [18]由Facebook开发，并根据工作负载的不同资源需求提供了更大的灵活性来管理集群资源。 Omega [19]，一种围绕共享状态构建的新并行调度程序体系结构，使用无锁乐观并发控制，以实现实现的可扩展性和性能可扩展性。 Mesos [5]从计算机中提取CPU，内存，存储和其他计算资源，从而使容错和弹性的分布式系统易于构建和运行。在这些大数据资源管理器中，YARN是唯一支持安全性并同时老化现有HDFS数据集的管理器。

MapReduce优化：使用各种技术，例如资源调配[20]，作业计划[21]和自整定配置[12]，对MapReduce性能优化的兴趣与日俱增。 Rao等。提出了一个新的MapReduce框架Sailfish [22]，以通过汇总中间数据来提高性能。金达等。 [23]提出了一种新的数据布局，称为“木马布局”，该内部将数据块内部组织为属性组，以缩短数据访问时间。 Dittrich等。提出了Hadoop ++ [24]，这是一种新的索引和联接技术，可以改善MapReduce作业的运行时间。他们能够使用最合适的Trojan Layout将传入的MapReduce作业调度到数据块副本。最近，一些研究开始探索如何优化Hadoop配置以提高MapReduce性能。 Herodotou等。提出了Starfish [25]，这是一种优化框架，通过搜索良好的参数配置对MapReduce作业到工作流程进行分层优化。这些方法都没有考虑通过Spark-on-YARN环境中的动态作业调度来优化MapRe duce性能。

任务调度：许多先前的研究表明，可以通过各种调度技术显着提高MapReduce性能[21]。 Hadoop实施中的默认FIFO调度程序可能无法正常工作，因为长时间的工作会排他性地占用群集上的计算资源，并导致其他工作的大量延迟。这就是许多调度程序（例如，容量调度程序，公平调度程序）可以在多个作业之间共享资源的原因。最近，一些研究[20]开始针对其性能目标优化MapReduce作业的性能。沃尔夫等。描述了FLEX [21]，一种针对MapReduce工作负载的灵活智能分配方案。它被建议作为与Fair Scheduler协同工作以提供性能保证的附加模块。 Curino等。设计了一个基于预留的计划[20]，该计划为具有各种截止日期的作业的优先MapReduce计划提供支持。我们的工作不同于这些工作，因为我们考虑在YARN群集上一起运行Spark和MapRe duce应用程序

**7 结论与未来工作**

我们注意到，由于YARN的基于预留的资源分配方案与动态应用程序需求之间存在语义上的差距，因此在YARN群集上运行Spark和MapReduce会导致性能显着下降。因此，我们设计和开发了跨平台资源调度中间软件iKayak，旨在提高群集资源的利用率以及YARN上Spark部署的应用程序性能。它依赖于三个关键机制，即，预留感知的执行器放置，感知依赖的资源调整和本地感知的分配协调。 iKayak利用了不同应用程序的时变资源需求，地图任务和约简任务之间的任务间依赖性以及跨平台的位置感知来应对上述挑战。实验结果表明，与YARN部署模型中的两种流行的YARN客户端模型和YARN集群模型相比，iKayak的Spark应用程序性能提高了50％，MapReduce应用程序性能提高了19％。在未来的工作中，我们将探索更多跨平台资源调度方法，同时在Hadoop YARN上部署其他处理范例（例如Shark，Pig，Storm和Hive）。

**致谢**

This research was supported in part by U.S. NSF research grant CNS-1422119.

**参考文献**

1. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A faulttolerant abstraction for inmemory cluster com­puting,” in *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI),* 2012.
2. Yahoo!, “Let spark fly: Advantages and use cases for spark on hadoop,” <https://spark-summit.org/2014/>
3. V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, “Apache hadoop yarn: Yet another resource negotiator,” in *Proc. ACM Symposium on Cloud Computing (SoCC),* 2013.
4. V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, “Apache hadoop yarn: Yet another resource negotiator,” in *Proc. ACM Symposium on Cloud Computing (SoCC),* 2013.
5. P. C. Foundry, “Yarn resource management,”

[http://pivotalhd.docs.pivotal.io/docs/yarn-resource- management.html](http://pivotalhd.docs.pivotal.io/docs/yarn-resource-management.html).

1. B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph,

R. Katz, S. Shenker, and I. Stoica, “Mesos: A platform for fine-grained resource sharing in the data center,” in *Proc. USENIX Symposium on Networked Systems Design and Implemen­tation (NSDI)*, 2011.

1. “PUMA: Purdue mapreduce benchmark suite,” <http://web.ics>. purdue.edu/ ~ fahmad/benchmarks.htm.
2. L. Wang, C. Luo, Y. He, J. Zhan, Z. K., X. Li, Y. Zhu, S. Zhang, Q. Yang, B. Qiu, and Z. Jia, “Bigdatabench: a big data benchmark suite from internet services,” in *Proc. of the IEEE Int'l Symposium on High-Performance Computer Architecture (HPCA)*, 2014.
3. M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling,” in *Proc. ACM European conference on Computer Systems (EuroSys)*, 2010.
4. S. . Documentation, “spark.deploy.spreadout,” <https://spark.apache.Org/docs/l.4.0/spark-standalone.html>.
5. Y. Chen, B. Yang, A. Abraham, and L. Peng, “Automatic design of hierarchical takagisugeno type fuzzy systems using evolutionary algorithms,” in *IEEE Trans. on Fuzzy Systems*, 2007.
6. Y. Wang, C. Xu, X. Que, X. Li, and W. Yu, “Jvm-bypass shuffling for hadoop acceleration,” in *Proc. of IEEE IPDPS*, 2013.
7. D. Cheng, J. Rao, Y. Guo, and X. Zhou, “Improving mapreduce performance in heterogeneous environments with adaptive task tuning,” in *Proc. of the ACM/IFIP/USENIX Int'l Middleware Confer­ence (Middleware)*, 2014.
8. R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, “Scale-up vs scale-out for hadoop: Time to rethink?” in *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2013.
9. eBay, “Using spark to ignite data analytics,” [http://www.ebaytechblog.com/2014/05/28/using-spark-to- ignite-data-analytics/U-qUSPldUbw](http://www.ebaytechblog.com/2014/05/28/using-spark-to-ignite-data-analytics/U-qUSPldUbw).
10. Cloudera, “Configuration parameters,”

<http://blog.cloudera.com/blog/author/aaron/>.

1. M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, N. Fuller, and A. R. Butt, “mronline: Mapreduce online performance tuning,” in *Proc. of ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2014.
2. Cloudera, “Apache spark resource management and yarn app models,” [http://blog.cloudera.com/blog/2014/05/apache- spark-resource-management-and-yarn-app-models/](http://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/).
3. Facebook, “Hadoop corona: the next version of mapreduce,” [https://github.com/facebookarchive/hadoop- 20/tree/master/src/contrib/corona](https://github.com/facebookarchive/hadoop-20/tree/master/src/contrib/corona).
4. M. Schwarzkopf, A. Konwinski, M. Abd-el Malek, and J. Wilkes, “Omega: flexible, scalable schedulers for large compute clusters,” in *Proc. ACM EuroSys*, 2013.
5. C. C., D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, “Reservation-based scheduling: If youre late dont blame us!” in *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2014.
6. J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar,

S. Parekh, K. Wu, and A. Balmin, “Flex: A slot allocation scheduling optimizer for mapreduce workloads,” in *Proc. of the ACM/IFIP/USENIX Int'l Middleware Conference (Middleware)*, 2010.

1. S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, “Sailfish: A framework for large scale data processing,” in *Proc. of ACM Symposium on Cloud Computing (SoCC)*, 2012.
2. A. Jinda, J. Quian?Ruiz, and J. Dittrich, “Trojan data layouts: Right shoes for a running elephant,” in *Proc. of ACM Symposium on Cloud Computing (SoCC)*, 2011.
3. J. Dittrich, J.-A. Quiane-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, “Hadoop++: making a yellow elephant run like a cheetah (without it even noticing),” *Proc. VLDB Endow.,* vol. 3, 2010.