中国科学: 信息科学 2020 年 第 50 卷 第 8 期: 1178–1196
SCIENTIA SINICA Informationis
论文

基于程序分析的大数据应用内存预估方法

胡振宇, 石宣化*, 柯志祥, 金海, 王斐

华中科技大学计算机科学与技术学院, 大数据技术与系统国家地方联合工程研究中心, 服务计算技术与系统教育部重点实验室, 武汉 430074

* 通信作者. E-mail: xhshi@hust.edu.cn

收稿日期: 2018–04–28; 修回日期: 2019–02–10; 接受日期: 2019–04–29; 网络出版日期: 2020–07–31

国家重点研发计划 (批准号: 2017YFC0803700) 和国家自然科学基金 (批准号: 61772218, 61433019) 资助项目

摘要 基于内存计算模型的分布式数据处理系统, 如 Flink 和 Spark, 经常遭受严重的内存压力, 尤其在平台内存资源紧张, 且被多个用户或组织共享情况下, 内存资源竞争进一步加剧. 用户应用被分配的内存空间不足, 会在运行期间产生严重的垃圾回收 (garbage collect, GC) 开销, 而分配过量的内存会导致平台资源的浪费. 因此平台中如何为用户应用配置合适的内存成为关键问题. 通过研究分析发现, 平台上的多个应用会多次共同处理某个特定的数据集, 且应用对数据的处理逻辑具有相似性, 如图计算和机器学习应用; 大数据应用框架的算子 API 和用户自定义方法 (UDF) 与数据处理逻辑有着密切的关系, 继而影响运行时内存的使用. 基于该发现, 本文提出了一种预估新提交应用的合理内存阈值的方法. 该方法利用程序分析与历史应用处理数据特征分析, 基于 kTree 判断与历史应用的数据路径的相似性来预估新应用的合理内存阈值, 并在 Spark 系统上实现该方法. 通过一系列实验评估预估的准确性和性能收益, 实验结果表明本方法预估大数据应用的结果与真实合理内存阈值的误差比例低至 4%, 预估过程所产生的开销与应用运行时间相比可以忽略不计, 平台上数据处理应用整体执行时间减少至 56%.

关键词 内存管理, 生命周期, 代码诊断, 数据处理系统, 分布式系统

1 引言

基于内存计算模型的大数据系统如 Spark [1], Flink [2] 等, 大部分是采用 Java, C# 和 Scala 这类高级语言开发, 应用处理的数据封装成对象驻留在内存中, 同时运行在受管理的运行平台上. 这些系统最重要的特性是它们内置一套自动管理内存的机制, 且具有自动垃圾回收的功能 [3]. 在大数据平台上, 大数据处理系统通常作为一个独立的批处理系统使用, 但也有很多情况作为服务面向多个不同的用户和组织, 比如 Hive [4] 与 Spark SQL [5], 因此相较于单独运行应用的批处理模式, 基于服务的数据处理

引用格式: 胡振宇, 石宣化, 柯志祥, 等. 基于程序分析的大数据应用内存预估方法. 中国科学: 信息科学, 2020, 50: 1178–1196, doi: 10.1360/N112018-00108
Hu Z Y, Shi X H, Ke Z X, et al. Estimating the memory consumption of big data applications based on program analysis (in Chinese). Sci Sin Inform, 2020, 50: 1178–1196, doi: 10.1360/N112018-00108

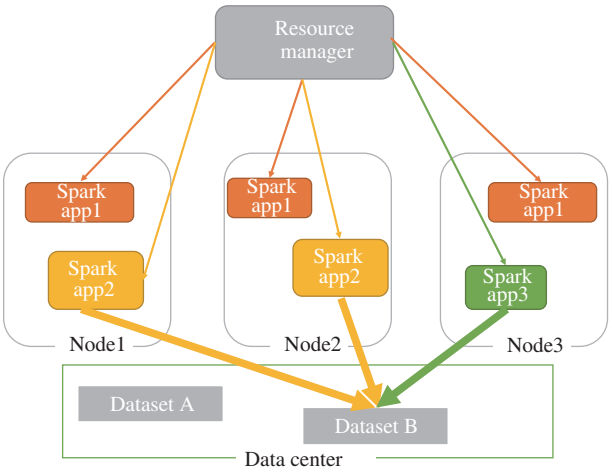

图 1 (网络版彩图) 大数据平台上资源管理器、应用与数据集的场景关系

Figure 1 (Color online) The scenario in big data platform about the relationship among ResourceManager, applications and datasets

系统会有大量应用同时在平台中运行. 由于平台中内存与 CPU 资源的共享, 并且基于内存计算的大数据框架会大量使用内存资源, 平台中的内存压力将变得更加严重, 内存资源成为整个平台性能的关键因素. 现有的一些研究工作尝试去优化内存管理或者设计特定的 GC 策略消除 GC 问题和内存压力 [6~8]. 在给出一个已知大小的数据集的情况下, 应用开发者通常凭经验值来配置应用的最大内存, 而大量的数据对象与复杂的对象结果会直接导致内存的膨胀, 并且大数据系统框架的算子 API 和 UDFs 提供了很多操作数据对象的模式, 这样会造成运行期间数据对象的总大小无法预测, 因此配置的内存大小经常会和应用运行期间所需要的内存空间有较大出入. 而且应用内存一旦配置, 运行之后便无法再改变. 综上所述, 大数据平台的使用者很难合理地为某个应用分配合适大小的内存空间. 若一个应用所配置的内存很小, 内存中缓存的大量数据对象会导致巨大的内存压力与 GC 问题. 尽管 Spark 等其他数据处理系统提供的内存数据交换 (swap) 机制会在内存不足时将内存中部分缓存数据溢出到磁盘. 然而溢出的操作会带来额外的磁盘 I/O 开销. 另一方面, 平台资源的总量是恒定的, 一旦应用配置了过量的内存空间, 会导致平台剩下更少的内存资源去分配给其他应用. 因此, 为应用配置合适的内存大小显得至关重要.

大数据平台中属于同一个使用者或组织的不同应用会经常访问共同的数据集 [9], 比如工业界与科研界的机器学习应用与图计算应用 [10,11]. 这里有 3 种情况应用会处理一份特定的数据集多次, 一是为了验证结果的准确性对应用进行多次测试, 二是在不改变应用逻辑代码的情况下, 仅仅变更其参数来获得不同的结果, 如在机器学习应用中调节模型的参数; 第三种更普遍的处理情况是开发者利用不同的机器学习, 图处理算法应用或者简单的数据处理应用来处理相同的离线数据, 以此计算并获得数据中不同的价值. 可以用一个简单的模型来说明本文预估方法适用的场景, 图 1 中显示资源管理、应用、数据集之间的交互, 使用基于 Spark 系统的数据服务. 资源管理器 (ResourceManager) 负责分配定量的内存资源给工作节点 (worker nodes) 上运行的应用, 应用一旦获取到静态配置的内存资源, 之后内存使用的上限是固定的, 这适用于 Spark 各种部署模式. 多个节点上运行若干个应用, 其中不同的应用会使用共同的数据集.

数据平台中存在多个应用处理同一数据集的特性, 这项事实提供了优化这些应用的机会, 本文的

研究进一步揭示出应用对数据集的处理逻辑主要由框架定义的算子与用户自定义的方法 (UDF) 组成, 可以通过程序分析的方法去分析数据的处理逻辑. 针对问题与发现, 本文提出一种基于程序分析技术预估新提交应用合理内存阈值的方法, 合理内存阈值能保证应用足以容纳所有缓存的数据, 同时不会发生数据 swap 到磁盘和严重的 GC 问题[1)]. 由于每个应用业务逻辑的不同, GC 情况以及合理内存阈值都是不同的, 无法用统一的标准值去衡量. 但是对于每个应用程序而言, 在其他参数不变的情况下, 我们以一个比较细粒度的方式逐渐增加内存的容量 (本文实验是以 1 GB 为粒度), 程序的 GC 时间占整个运行时间的比例会随着内存的增加而减少, 随后会逐渐稳定保持不变. 这是因为当内存由少变多的过程中, JVM 中有更多的内存来容纳对象, 减少了垃圾回收的次数, 从而减少了垃圾回收的时间占比, 我们称使程序 GC 时间占比开始趋于稳定的内存值为合理阈值. 该预估方法通过应用程序分析跟踪长生命周期对象, 通过一种数据结构 kTree 记录和描述内存计算系统中的缓存数据对象, 如 Spark 中的缓存数据 (cached data) 和洗牌缓冲区 (shuffle buffer). 然后在用户提交新应用时, 通过 kTree 的相似性匹配来预估新应用的内存需求, 给出其合理内存阈值, 最终在 Spark 系统上实现该方法. 应用本方法后, 在大数据平台上运行的应用会更合理地使用总内存资源, 并且不损害自身的性能.

综上所述, 本文提出的预估方法包含以下 3 项贡献:

- 分析了在大数据平台下处理共同数据集的应用之间的关系, 发现框架的算子 API 与用户编写的 UDF 共同构建了应用处理数据集的流程逻辑.
- 利用静态程序分析, 得到应用的数据路径, 继而设计出 kTree 来描述数据处理流程, 用于新提交应用的与历史应用相似性的匹配.
- 设计出用于应用运行期间合理内存阈值的预估模块. 通过匹配存储的历史 kTree 与新提交应用的 kTree 来判断是否可以预估内存, 进一步使用细粒度的算法预估出应用的合理内存阈值.

2 背景与动机

2.1 缓存数据对象

在基于内存计算模型的大数据框架中, 通常包括两部分重要的缓存数据: 一是重复使用的数据会被驻留到内存中, 避免重算的开销; 二是 Shuffle 过程在内存中持续聚合的数据. 这两类缓存数据占用应用中很大的内存比例, 且具有较长的生命周期. 为预估应用运行期间的合理内存阈值, 首先需要预估这两类缓存数据所占用的内存空间大小. 如 Spark 框架中的 Shuffle Buffer 和 Cached Data, 这两项缓存数据会几乎占用典型 Spark 作业的所有内存空间. 大数据框架通常包含一系列算子 API [12], 这些内置的算子 API 拥有自己独特的语义, 如 Spark 中 `filter` 的作用是筛选数据, 并且开发者需要将自定义的 UDF 注入到部分算子 API 中. 由算子 API 和 UDFs 编写的 Spark 程序在真正运行之前, 应用生成的作业 (job) 会在 driver 端构建一个具有依赖关系的 DAG (directed acyclic graph) 执行计划, DAG 的基本单位为 RDD (resilient distributed datasets). 通过遍历所有的 RDD, 根据 RDD 间的 Shuffle 操作, 把 job 对应的 DAG 图划分成一个或多个阶段 (stage), 每个 stage 基于数据分区生成一组任务集合. 对于 Spark 应用来说, 每个子阶段 stage 中常驻内存的数据对象主要包括以下两类:

Shuffle Buffer. 一个 stage 中 Shuffle Buffer 在开始的 Read 阶段与末尾的 Write 阶段形成, 用于存储每个 Key 和对应 Key 的 Value 聚合值. Spark 支持多个与 Shuffle 操作相关的算子 API, 比如

1) 业务逻辑和内存大小会影响 GC 严重程度, 当程序发生了 Full GC 时, 我们认为此时程序的 GC 比较严重. 在其余参数不变的情况下, 当内存容量较小时, 我们查看 GC 日志会发现, Full GC 发生比较频繁, 从而导致程序暂停时间长, 垃圾回收占整个运行时间的比例偏高. 当内存容量增大时, Full GC 的次数逐渐减少.

`groupByKey`, `reduceByKey`, 它们都是基于 Key 操作的算子, 处理由 Tuple 类封装的 Key-Value 二元组对象, 然后在内存中创建 Shuffle Buffers 并将二元组对象填充进去. `groupByKey` 为每个 Key 将所有的 Values 聚合成 Value 列表, Spark 采用类 HashMap 的结构存储 Key 与 Value 列表. 而 `reduceByKey` 负责将 Key 的所有 Value 聚合成单个值, 且同样以 Hash 的方式存储. 值得注意的是 Shuffle Buffer 中所有的对象在对应数据分区计算完成后会被释放, 不会再占用内存空间. 即 Shuffle Buffer 的生命周期会在 stage 结束时而终结.

Cached Data. Cached Data 中的对象通常具有很长的生命周期. 任意 Cache RDD 都包括了多个 Cache 存储块, 每个存储块包含了一组数据对象. 当用户对某个 RDD 调用 `cache()` 时, Spark 会将其标记为 Cache RDD. Cache RDD 中数据对象的生命周期直到调用 `unpersist()` 方法或程序结束才会终结. 因此 Cache RDD 中的数据对象会存活数个 stage 或数个 job, 其生命周期可以通过程序分析判定.

2.2 不同内存配置的影响

大数据处理系统如 Spark [1], Flink [2], 其应用的开发者只能一次性地设置其静态配置, 无法控制程序运行期间的具体 GC 情况. JVM 中的垃圾回收机制将堆中的对象分成多代, 一般包括年轻代和老年代. JVM 上大量的数据对象由框架和用户应用程序生成并缓存起来, 它们长久地存在于老年代中直到它们生命周期的结束, 而临时分配的对象都生存在 JVM 的年轻代中. 一次全局 GC (Full GC) 操作会触发直接回收被老年代对象所占用空间的操作. 对于平台上应用来说, 如果内存空间不足以容纳这些长生命周期对象或者剩余内存空间非常少, 应用的性能将会因为频繁的 Full GC 操作而明显恶化, 甚至有可能发生 Out-Of-Memory (OOM) 错误.

为了说明不同的最大内存配置对应用性能的影响, 本文使用 Spark 中 benchmark 应用 PageRank (PR) 设计出一个简单的性能测试实验, 测量 Spark PR 在不同 JVM 最大内存值下的 GC 时间与计算时间开销. JVM 的堆大小通过设置 Spark 的参数 spark.executor.memory 来决定, 堆空间主要用来缓存 PR 应用生成 Shuffle Buffer 和 Cached Data. 为消除其他因素对结果的影响, 如网络数据的传输, PR 应用以 local 模式运行, 只配备一个 executor 来运行任务. 实验环境的单个节点配备两个 8 核 Xeon-2670 CPU, 一个 SAS 硬盘和 64 GB 内存的机器节点. 输入数据为由 Hibench 工具 [13] 随机生成的图数据, executor 的执行内存大小范围设置为 10 GB 到 35 GB. 设置总内存大小的 50% 与 40% 用于 Cache 数据与 Shuffle 操作. 实验结果在图 2 中显示, 当最大内存值设为 10 GB 时, PR 的性能主要受大量的磁盘 I/O 影响, 因为大量的数据会溢出 (spill) 到磁盘. 当内存大小为 20 GB 时, 虽然 Shuffle Buffer 和 Cached Data 刚好被容纳, 没有数据溢出到磁盘, 但是临时对象没有更多的内存空间和内存的紧张导致 Full GC 频繁被触发, GC 操作所花费的时间开销是计算时间的将近 3 到 4 倍. 而 30 GB 的总运行时间和 35 GB 基本相同, 这是因为内存已经足够容纳程序产生的对象, 必要的 GC 操作是无法消除的, 因此 30 GB 是该应用处理该数据集的合理内存阈值.

3 方法设计

3.1 程序分析

我们基于 Soot 工具[2)]以及迭代融合 [14] 在 Spark 上实现了程序分析器, 它在每个 job 提交到 driver 时, 分析整个 job 的数据路径. 该过程主要分为两个阶段, 第 1 阶段为预处理阶段, 该阶段通过融合每

2) Soot framework. 2016. http://sable.github.io/soot/.

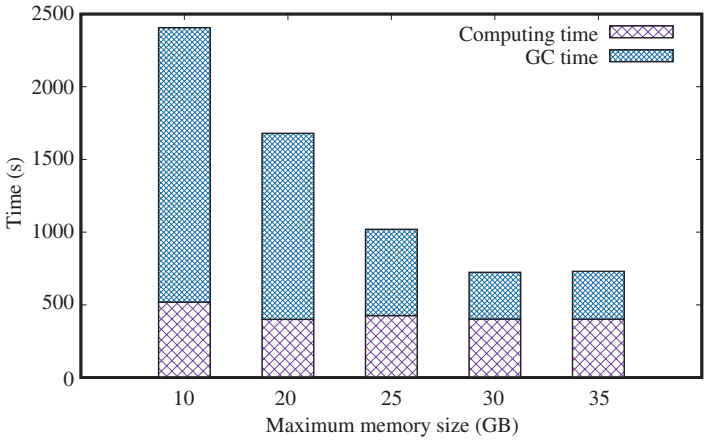

图 2 (网络版彩图) local 模式运行的 PR 应用在不同内存大小下的性能表现

Figure 2 (Color online) The performance of Spark PR with different maximum memory sizes, in Spark local mode (one executor)

个作业的 stage 中的链式调用的 UDF 以避免分析 Spark 框架代码; 第 2 阶段为指针分析阶段, 该阶段为后续的数据路径分析提供 UDT 的详细信息.

在预处理阶段中 Spark 的 job 可以根据宽窄依赖分为不同的 stage, 每个 stage 之间又是由不同的转换算子组合而成, 用户的 UDF 嵌入在这些转换算子之中, 不同转换算子的计算过程需要通过 Spark 的框架代码连接, 因此当我们直接进行程序分析时, 会涉及到复杂的 Spark 框架代码, 这是非常耗时的. 然而这些 UDF 是相互隔离的, 每个 UDF 完成特定的功能, Spark 框架代码只是起到将数据从一个 UDF 的输出转移到下游 UDF 的输入的作用, 因此我们可以将一个 stage 中转换算子包含的 UDF 抽取出来, 将各个 UDF 的逻辑融合到一个独立的代码区域中, 从而跳过 Spark 框架代码. 如图 3 所示, 是我们融合 LR 作业的例子. 对于 Spark 作业的代码, 程序分析器首先根据 RDD 的血缘关系构建出一个有向无环图 (DAG), 利用 Spark 中的 DAGScheduler, 根据作业中的 shuffle 操作将 DAG 划分为 stage. 然后我们通过观察 stage 的特征, 决定循环体的层次. 一般地, 一个 stage 生成一个循环体用于放置融合的代码, 当 stage 中有缓存操作时, 我们需要将循环体一分为二, 第 1 个循环体用于存放缓存之前的操作, 第 2 个循环体用于存放从缓存中读取数据然后处理缓存数据的操作, 这样便于我们分析缓存的数据路径. 在决定了循环体的层次之后, 程序分析器利用 Java 反射提取转换算子中的 UDF, 并结合算子语义如 filter, flatmap, mapValues 等, 将 UDF 融入到循环体中. 最后程序分析器利用 Soot 生成一个合成类, 将循环体作为一个方法置于该类中.

指针分析阶段给数据路径分析提供 UDT 的信息, 程序分析器用 Soot 来对上一阶段生成的合成类中的循环体进行指针分析. 指针分析可以确定数据变量在循环体中的赋值位置, 以及赋值表达式, 从而确定数据变量的具体类型. 因为迭代融合之后的合成类是一个独立的类, 无法用 soot-SPARK 直接进行指针分析, 为了解决这个问题, 程序分析器生成一个入口方法和入口类链接到之前生成的合成类, 以让 Soot-SPARK 可以进入到循环体.

3.2 数据路径分析

为预估应用的合理内存阈值, 预估方法首先需要分析缓存数据对象的处理流程. 大数据应用中包括框架的 API 算子和用户定义的方法 UDF, 对于 Spark 应用来说, 数据路径由 UDF 和算子 API 构

```
public final class StageFunc {
    ...
    public DenseVector$Double compute(TaskContext, int, DataSourceIterator) {
        ...
        r0 := @this: StageFunc;
        r3 := @parameter2: DataSourceIterator;
        ...
        z0 = staticinvoke <BlockManager: boolean isContainBlock(String)>(blockId);
        if z0 == 0 goto label2;
        r14 = staticinvoke <BlockManager: CacheIterator getIter(String)>(blockId);
      label1:
        z2 = virtualinvoke r14.<CacheIterator: boolean hasNext()>();
        if z2 == 0 goto label5;
        r18 = virtualinvoke r14.<CacheIterator: LabeledPoint next()>();
        d0 = (double) 1;
        d1 = (double) 1;
        d2 = virtualinvoke r18.<LabeledPoint: double get_label()>();
        d3 = neg d2;
        r22 = r0.<StageFunc: DenseVector$Double w>;
        r23 = virtualinvoke r18.<LabeledPoint: DenseVector$Double get_features()>();
        d4 = virtualinvoke r22.<DenseVector$Double: double dot(DenseVector$Double)>(r23);
        d5 = d3 * d4;
        d6 = staticinvoke <math: double exp(double)>(d5);
        d7 = d1 + d6;
        d8 = d0 / d7;
        d9 = (double) 1;
        d10 = d8 - d9;
        r31 = virtualinvoke r23.<DenseVector$Double: DenseVector$Double
         times(double)>(d10);
        r36 = virtualinvoke r31.<DenseVector$Double: DenseVector$Double
        times(double)>(d2);
        r37 = virtualinvoke r0.<StageFunc: DenseVector$Double
        reduce(DenseVector$Double,DenseVector$Double)>(r37, r36);
        goto label1;
      label2:
        staticinvoke <BlockManager: void createBlock(String)>(blockId);
      label3:
        z3 = virtualinvoke r3.<DataSourceIterator: boolean hasNext()>();
        if z3 == 0 goto label3;
        r39 = virtualinvoke r3.<DataSourceIterator: String next()>();
        virtualinvoke r0.<StageFunc: LabeledPoint parsePoint(String)>(r39);
        goto label5
      label4:
        r44 = staticinvoke <BlockManager: CacheIterator getIter(String)>(blockId);
      label5:
        z4 = virtualinvoke r44.<CacheIterator: boolean hasNext()>();
       ...  //the same code with label1
      label6:
        return r37;
   }
}
```

图 3 (网络版彩图) 应用 LR 迭代融合融合之后的代码

Figure 3 (Color online) Code of LR after iterative fusion

成, 如 `filter`, `map`, `join`, `flatMap` 等, 数据单元经过数据路径封装进 RDD 中. 应用中的处理可以简洁地归纳为控制路径 (control path) 与数据路径 (data path)[7,15], 控制路径负责在主节点中创建任务, 对程序本身实现优化, 属于控制路径中的代码只会在主节点端执行. 而数据路径真正涉及到数据集中的数据对象、数据路径在计算节点中执行, 其处理逻辑较为简单, 且代码规模 (code size) 远小于控制路径. 因此对于大数据应用数据路径的分析具有较低的开销.

通常, Spark 应用数据单元由 Hadoop, 文件系统迭代地读取到内存中, 然后经过 UDF 和算子 API 处理, 在 stage 的开始和末尾生成聚合的 Key-Value 对象并存储到 Shuffle Buffer 中. 类似地, 属于 Cache RDD 的数据对象会进入缓存块中, 如图 4 所示, 该流程就是上文提及的数据路径. 本文进一步

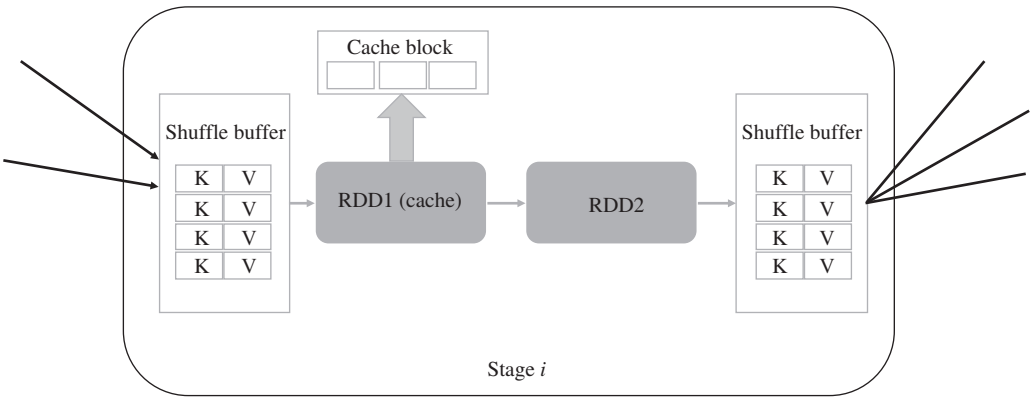

图 4　Spark 应用中 stage 的数据路径

Figure 4　The data path in a stage of the Spark application

表 1　3 个典型应用中缓存数据的分析

Table 1　The analysis of each container in three typical applications

Application	Data container	Data structure	Data type
PR	Cached data	(K1, Array(V1))	Int, Int
	Shuffle buffer	(K2, V2)	Int, Double
CC	Cached data	(K1, Array(V1))	Int, Int
	Shuffle buffer	(K2, V3)	Int, Double
WC	Shuffle buffer	(K1, V4)	Int, Int

地发现由于不同应用中缓存的处理流程相似, 且数据路径的 code size 很小, 结果是不同应用缓存对象的数据路径相似甚至完全一致. 为说明这个问题, 本小节展示出一个典型且简单的应用场景. 3 个应用分别处理同一份图数据集以获取不同的需求结果, 应用分别为 PageRank (PR), ConnectedComponent (CC) 和 WordCount (WC), 数据集为两列整型数据构成 [16], 整型值代表顶点. 其中 PR 是具有代表性的连接分析算法3). CC 目的在于计算所有顶点的连通分量归属. WC 原用于聚合每个 key 的 value, 这里是统计每个顶点的入边个数. 为了说明以上 3 个应用中缓存数据的数据路径相似性, 可以用简略的表达式来描述 Shuffle Buffer 和 Cache Data 的数据形式. 表 1 中显示 3 个应用处理同份数据集时缓存的数据形式. PR 通过 `groupByKey` 算子将数据源中读取的 Key 对应所有的 Value 聚合成 Value 列表, 然后以 (K, Array(V)) 的形式缓存到内存中, 形成 Cache RDD. 由于 `groupByKey` 生成的 Shuffle Buffer 与 Cache Data 中的数据引用的是内存中同一份数据对象, 所以两者占用的内存空间仅需要计算一次. 表 1 中描述的 Shuffle Buffer 由位于第 3 个 stage 的 `reduceByKey` 算子产生, 其数据对象在内存中以 (K, V) 形式存在, 但是 Key 自身的值因为后续的算子与 UDF, 相较于 Cache Data 来说已经发生变化. 通过观察 3 个应用缓存的数据形式, 相似性可以很简易地判定出来. 例如, PR 与 CC 中 Cache Data 所占用的内存空间大小基本相同. 尽管 3 个应用 Shuffle Buffer 中的数据形式不一致, 但本文的预估方法可以利用其中一个数据形式的内存大小推断出其他形式的内存大小, 具体策略在 3.3 小节中介绍.

为了持久化分析得到的缓存数据的数据路径, 且方便数据路径之间的匹配来预估应用的合理内存阈值, 需要将数据路径转化成抽象的数据结构. 本文提出一种特定的树状的数据结构 kTree 用于描述

3) The definition of pagerank. https://en.wikipedia.org/wiki/PageRank.

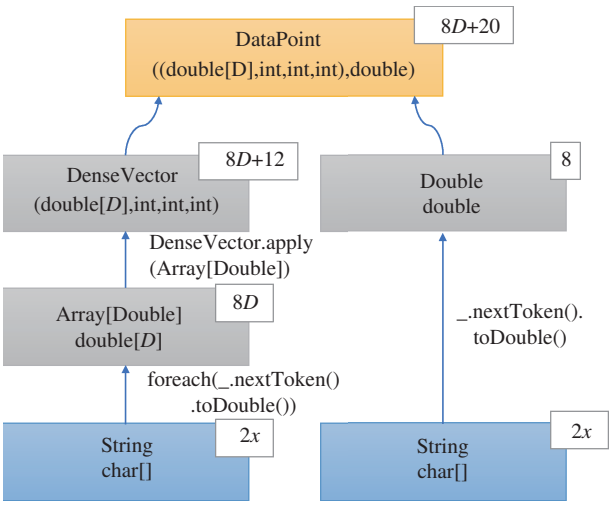

图 5 (网络版彩图) LR 应用的 kTree, 根节点代表 Cache Data 中的数据对象

Figure 5 (Color online) The kTree of logistic regression (LR). The root node represents the data objects in cache data

应用中缓存数据的数据路径, 从而基于 kTree 来预估新提交应用的合理内存阈值.

数据路径的提取需要采用程序分析技术, 而大数据应用本身的代码涉及到框架内部的代码和 API, 直接对其代码进行分析会生成大而复杂的调用关系图 (CallGraph), 造成大量时间与内存的开销. 为缩短分析时间, 本文首先使用 Soot 工具并利用迭代融合技术 [14], 将具有迭代性与隔离性的 UDF 划分到一块代码区中以实现过程内分析. 迭代融合之后, 每个 stage 生成一个代码循环体, UDF 与算子 API 的语义全都体现在循环体中. 基于迭代融合, 程序分析模块对代码循环体中 Shuffle Buffer 与 Cache Data 中的数据对象进行指针分析, 以确定长生命周期对象的数据路径. 技术上, 本方法选择使用 kTree 这种与抽象语法树 (AST) [17] 类似的树结构, 用于描述长生命周期对象如何生成和处理的过程. kTree 的节点代表对象实体, 对应的边记录了方法调用或表达式. 值得注意的是程序分析模块同样分析了每个节点的对象实体的层次结构, 将对象实体分解成原生类型对象或数组形式原生类型对象, 这样做的目的是为了表达数据对象大小 (data-size). 为了更清晰地说明 kTree 的构建和形式, 本文选择了数据路径较为复杂的 Logistic Regression (LR) 应用. 如图 5 所示, LR 应用 Cache Data 中的数据对象的数据路径以 kTree 的形式展现. LR 读取输入数据集中的每一行生成一个 DataPoint 对象, 数据路径的源头是一行 String 类型数据, 根节点是最终创建的 Cache Data 中的数据对象. 其他节点和边显示了从数据集到最终缓存对象的中间对象和方法调用或表达式, 并且每个对象都进行了拆解, 展示出由原生类型对象组成的层次化数据结构, 用数据值来表示该对象的对象大小, 若存在未知长度的数组对象, 则采用表达式来定义它的长度. 图中的 D 就是 double 数组的长度, 则 DenseVector 节点的对象大小是 $8D + 12$. 本文提到的预估方法会基于新应用与历史应用中数据路径的相似性或一致性, 具体地判定对应的 kTree 的相似性, 最后利用简单的算法来预估新提交应用的合理内存阈值.

3.3 预估合理内存阈值

为了以最少的资源让应用拥有最好的性能, 本文提出了一种基于程序分析的预估方法, 通过分析和计算得到新提交的应用的合理内存阈值. 上节所说的程序分析模块主要是生成缓存对象数据路径对应的 kTree, 而这里的预估模块包含的功能有: (1) 获取与存储历史应用运行时的重要数据特征; (2) 通过匹配 kTree 来计算新提交应用的合理内存阈值.

本方法以较细粒度的一种方式, 以 Spark 的 stage 为单位来分析每个 stage 的合理内存阈值, 再决定整个应用的合理内存阈值. 对于每个 stage, 合理内存阈值意味着在不浪费内存的情况下, 使用最小的内存保证 stage 在运行期间不会产生数据 spill 现象, 且不发生严重的 GC. 具体的方法是统计这个 stage 的 shuffle buffer 对象与 cache 对象的内存大小的和, 而临时对象生命周期很短, 不会长时间占据内存空间, 以及 Spark 框架代码本身运行的内置对象所占内存大小比例很低, 所以将之前计算的和除以一个比例 Ratio, 该比例为 shuffle buffer 对象与 cache 对象的内存大小占 spark 执行总内存的比值, 除之后的结果即一个 stage 的合理内存阈值. Spark 应用会基于 action 算子划分成多个 job, 所以一个应用可能包含一个甚至更多 job, 这种情况在迭代性应用中很常见. 每个 job 以动态的方式提交到 Spark 平台. 预估模块会对 job 进行拦截, 然后分析 job 中所有 stage 的数据路径. 然而, 第一个 job 分析完之后, Spark 会首先将任务下发到各 executor 直接运行这个 job, 且此时的 JVM 堆大小已经固定, 无法在后面 job 提交时改变. 所以需要一次性分析完所有 job, 解决方法是采样出很小的数据集在 Spark 中进行试跑, 获得所有 job 中的 stage 的 kTree, 从而预估出每个 stage 的合理内存阈值, 将同时提交执行的 stage 划分到同一组求和, 然后在其中取出最大值作为应用的合理内存阈值. 预估完内存后会重新提交应用 jar 包. 在具体分析每个 stage 之前, 存在一个重要流程是识别出迭代性应用的循环体中的 stage, 这部分 stage 无需重复分析其数据路径, 因为它们的代码段即数据路径完全一致. 方法是通过静态识别循环体并跟踪到对应形成的 stage, 这样可以大大减少迭代性应用程序分析的时间开销.

上述情况讲述了如何根据 kTree 来预估一个 stage 的内存, 这个过程发生在 job 真正执行前. 但在 stage 形成任务集运行之后, 预估模块需要统计运行时重要的数据特征, 包括迭代过程中数据项数 Item_Num, 运行期间占用缓存内存 Cached_Data_Size 和 Origin_Shuffle_Size, Origin_Shuffle_Size 包含 Shuffle Read 和 Shuffle Write 阶段生成 buffer 的大小. 它们都可以直接通过 Spark 内部采样的接口获得, Shuffle Buffer 会在一个数据分区聚合完成之后将缓存释放掉, 数据分区数等于总任务数 (partitions), 并发线程数为 threads_num, 最终的 Shuffle_Buffer_Size 通过式 (1) 计算得到. 当一个 stage 运行结束, 会为每个 kTree 存储这些对应的数据特征, Item_Num, Shuffle_Buffer_Size, Cached_Data_Size. 这些数据特征是计算出新 stage 的合理内存阈值的重要条件. 在实践中, Cache 的数据和 Shuffle 缓存数据可能都会溢出到磁盘中, 但不会妨碍在程序运行期间获取这些数据特征. 算法 1 中显示了预估单个 job 合理内存阈值的具体过程, 包含对每个 stage 分别进行处理 (行 4, 10).

$$\text{Shuffle\_Buffer\_Size} = \frac{\text{Origin\_Shuffle\_Size}}{\text{partitions}} \times \text{thread\_num}. \tag{1}$$

如何基于 kTree 与数据特征预估任意新 stage 在运行期间的合理内存阈值成为一个根本性的问题. 算法 1 描述了如何通过 kTree 匹配获取缓存数据大小的方法 (行 13, 33). 本方法需要去获取新 stage 中长生命周期对象的 kTree, 与已存储的历史应用的数据路径, 即 kTree 进行匹配与比较, 直到匹配成功为止, 若匹配全部失败则放弃预估, 按照原配置对应用进行提交. 具体的匹配方法分为两种情形:

- 通常, 匹配两个 kTree 需要依次比较对象大小, 原生类型对象层次结构和 kTree 的具体结构. 匹配成功, 则说明该 stage 与某个历史 stage 的数据路径相同, 预估模块会调取该历史 stage 的数据特征 Shuffle_Buffer_Size, Cached_Data_Size, Ratio 为 Spark 默认的第 3 部分内存配置比例, 通过式 (2) 计算得出新 stage 的合理内存阈值. 值得注意的是, 不同应用中 stage 中缓存数据 kTree 相同是比较普遍的现象, 比如机器学习应用、图计算应用. 可以在图 6 中看到 PR 和 CC 中缓存数据拥有相同的数

算法 1 Process of estimating a reasonable memory threshold for a single job

Input : The submitted job from an application Job;

Output: The optimal memory size M;

acquire all stages Stages from Job; get the Manager KM for kTrees; create the estimating memory size set M_Set for all stages;

foreach stage **do**

  /*Estimate the appropriate memory threshold for each stage, including Shuffle Buffer and Cached Data*/;

  kTree1 ← KM.generate(stage.CachedData); kTree2 ← KM.generate(stage.ShuffleBuffer);

  cache_memory ← `MatchKTree(kTree1)`; shuffle_memory ← `MatchKTree(kTree2)`;

  total_memory ← (cache_memory + shuffle_memory)/ratio;

  M_Set.add(total_memory);

end

M ← M_Set.getMax();

return M;

/*Matching the cached kTree in the new application*/;

Function `MatchKTree(`kTree$_{\text{arg}}$`)`

  top_object ← kTree$_{\text{arg}}$.TopNode;

  /*If the top-level object is the primitive type*/;

  if top_object is a primitive type **then**

    old_kTree ← KM.querySimilar(kTree$_{\text{arg}}$);

    old_top ← old_kTree.top_object;

    memory ← KM.getMemorySize(old_kTree);

    if top_object is proportional to old_top **then**

      if top_object isSameType(old_top) **then** memory **else**

        return memory/old_top.DataType * top_object.DataType;

      end

    end

    else

      KM.storeKTree(kTree$_{\text{arg}}$);

      return NULL;

    end

end

  else

    old_kTree ← KM.querySame(kTree$_{\text{arg}}$);

    if top_object is not null **then**

      memory ← KM.getMemorySize(old_kTree);

      return memory;

    end

    else

      KM.storeKTree(kTree$_{\text{arg}}$);

      return NULL;

    end

end

据路径. 进一步地, 对于以二元组形式的 Shuffle Buffer 来说, 虽然 kTree 相同, 然而若一个 stage 由 `reduceByKey` 生成 (K, V) 形式的 Shuffle Buffer, 另一个 stage 是 `groupByKey` 聚合数据生成 (K, Array(V)) 形式的缓存数据, 可以在图 7 中看到 WC 中 Shuffle Buffer 与 PR, CC 的区别. 技术上, (K, V) 形式的缓存数据所占内存大小依然可以通过 (K, Array(V)) 形式数据占用的内存大小计算. 且 `groupByKey` 聚合成 (K, Array(V)) 可以获得 Key 的数量即记录的数据特征 Item_Num, 所以 (K, V) 形式的 Shuffle

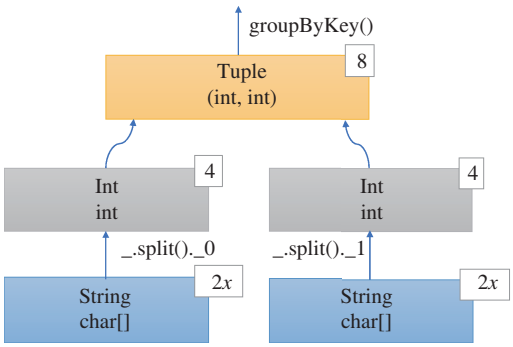

图 6 (网络版彩图) PR 与 CC 中 groupByKey 算子在 stage2 中产生的 kTree

Figure 6 (Color online) The kTree of the second stage produced by **groupByKey** in PR and CC

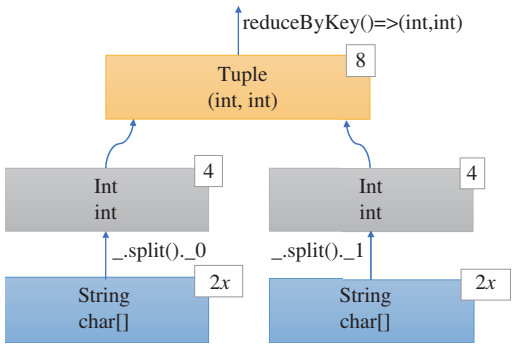

图 7 (网络版彩图) WC 中 reduceByKey 算子在 stage2 中产生的 kTree

Figure 7 (Color online) The kTree of the second stage produced by **reduceByKey** in WC

Buffer 的大小为 Item_Num* (Key_Size + Value_Size).

$$\text{Proper\_Memory\_Threshold} = \frac{\text{Shuffle\_Buffer\_Size} + \text{Cached\_Data\_Size}}{\text{Ratio}}. \tag{2}$$

• 存在这样一种特殊情况, Shuffle Buffer 中的二元组 key 为原生类型对象, 包括 Int, Double, Float, Char, Long. 这种情况在大数据应用中是很常见的. 当符合这种条件的新的 stage, 即它的 shuffle buffer 的 key 是原生类型, 该 kTree 与历史 kTrees 都无法匹配, 但存在一个历史 kTree, 且新的 kTree 与历史 kTree 的顶层原生对象呈一次线性关系, 使得历史 kTree 中任意一个 key, 新 kTree 都会有唯一的新 key 存在, 使得最后 Shuffle Buffer 中的 key 个数仍然不变, 即新 kTree 对应缓存数据的 Item_Num 是没变的. 若数据类型 (DataType) 不同, 但都是原生类型时, 可以根据大小的比例来换算总数据对象大小. 因此, 尽管 kTree 不完全一致, 但新 kTree 对应的缓存数据大小仍然可以预估出来.

以上都是基于 kTree 的相似或者一致性来判定缓存数据中 items 数是相同的情况. 若新 kTree 遍历了所有历史 kTree, 在与某历史 kTree 结构基本一致的情况下, 但缓存数据生成的过程额外地经过筛选操作如 `filter` 算子, Item_Num 会减少, 但本方法仍然参照历史 kTree 对应的内存大小来配置. 这是一种保守的做法, 使得在已有的历史信息下浪费尽量少的内存来保证程序的高性能.

4 实验

4.1 配置

本节主要通过实验来评估本工作的有效性和性能. 实验配置使用 4 个节点为用于计算的 worker, 1 个节点为 master. 每个节点配备 2 个 8 核 Xeon-2670 CPU, 64 GB 的内存, 和一块 500 GB 大小的 SAS 硬盘, 操作系统为 RedHat Enterprise Linux 5 (kernel 为 2.6.18), JDK 版本为 1.7.0. 运行的 Spark 版本为 1.6.3. 为了能精确配置 Shuffle 操作和 Cache 的内存大小, 防止两者之间的内存调度, 所有实验都关闭了统一内存管理模式. 所有的实验都会重复三次, 并采用三次的结果取平均值来作为结果, 以此保证正确性, 实验结果的误差控制在 10% 以内.

4.2 预估内存阈值的准确度

我们测试了 PR, CC, WC, LR 和 KMeans 预测精度, 与手动测试的合理阈值相比, 最大误差比不

超过 11%. 下面我们选择 PR, CC 和 WC 3 个 benchmark 应用来进行具体说明. 它们在执行任务时会在内存中驻留数据对象, 其中 PR 和 CC 相似度最高, 尽管代码逻辑不一致, 都具有 Cache 操作和 Shuffle 操作, 而 WC 仅包含简单的 Shuffle 聚合操作, 只需要预估 Shuffle 操作所消耗的内存. 本小节实验中, PR 和 CC 的迭代次数均设置为 10 次.

实验中使用的数据集由开源工具 Hibench [13] 随机产生, 数据集由两列数值构成, 分别代表点和边, 整个数据集代表着一张由点与边构成的大图. 事先利用手动实验来测试得出 3 个 benchmark 应用在某个共用数据集下的合理内存阈值, 具体方法是通过不断调整 JVM 堆的大小来找出它们运行的合理内存阈值, 测试的内存变化粒度为 1 GB, 并且记录它们在各个内存配置下程序执行的整体时间. 测试应用所处理的数据均为 Hibench 所生成的 40 GB 数据, 单个节点内存大小区间为 5∼20 GB 来配置 PR 和 CC, 内存区间为 1∼10 GB 来配置 WC 应用, 如图 8 中显示 3 个应用在不同内存大小下的执行时间, 同时记录了 GC 时间的变化, 体现出不同内存下 GC 的严重性. 可以看到当配置内存增长到一定大小时, 应用执行时间和 GC 次数不会继续降低, 性能变化在图中出现明显的拐点, 这也是本文所提出的合理内存阈值. 而当内存小于合理内存阈值时, 应用执行时间呈倍数增长, 如 PR 各节点设置为 6 GB 内存时, 是合理内存阈值下运行时间的近 2 倍, 同时在图 8(c) 可以看到 WC 应用在 1 GB 时由于内存严重不足出现了 OOM 错误. 图 8 中同样标记了本方案预估的内存大小, 可以发现预估的结果和拐点都很接近, 手动测得的合理内存阈值 PR 和 CC 的预估结果与手动测试得到的合理内存阈值的误差比例不超过 6%, WC 误差比例不超过 11%.

应用所需要的合理内存阈值会随着数据集大小的变化而改变, 为了进一步验证预估方案的准确性, 实验针对 PR 和 WC 应用生成了不同大小的数据集, 分别为 10 GB, 30 GB, 60 GB, 数据依然由 Hibench 随机产生. 首先手动测试出 PR 在各数据集下的合理内存阈值, 然后采用本文的预估优化方案得出预估内存, 可以在图 9 中发现, 真实的合理内存阈值与预估内存大小误差比例范围为 4%∼14%. 且数据集越大时, 预估结果与真实结果的误差比例有减少的趋势, 在数据集大小为 60 GB 时, 应用 PR 与 WC 的误差比例均不到 5%, 这是因为 Spark 获得内存使用情况是通过采样得到的, 使用它的接口并计算来预估出结果. 当数据集越大时, 采样的误差对总数据的大小影响减小, 所以体现出误差减小的趋势. 同时, 另增加了一组实验数据, 手动测试历史应用运行时有 Shuffle 缓存数据溢出到磁盘的合理内存阈值, 验证这种情况下本预估方案的精确度. 如图 9 所示, 其与真实值的误差比例与无数据溢出情况下几乎一致, 这是由于溢出到磁盘的数据会首先在内存中缓存, 预估方案在这个时间段同样获取到数据所占用的内存大小和其他数据特征. 因此历史应用若出现数据溢出到磁盘, 也可以获取到完整的数据特征供预估算法所利用, 预估的结果也与无 Shuffle Spill 情况基本一致.

4.3 程序分析开销

本小节实验的目标是探究本预估方案的程序分析的时间开销. 对于应用运行时动态地获取数据特征这部分的时间开销, 由于只是在数据迭代过程中附加了累计的操作, 最后将收集和累计后的数据返回给 driver 端并持久化到磁盘上, 这些数据很小, 因此相较于应用真正的计算开销, 额外带来的网络传输开销和磁盘开销可以忽略不计. 而方案中程序分析的流程较多, 包括迭代融合、指针分析、kTree 的构建和匹配, 研究这部分操作的开销是必要的, 也是衡量本预估方案价值很重要的标准.

本小节实验中分别测试了 5 个 benchmark 应用, 其中增加了 LR 和 KMeans, 并且在 Spark 的开源机器学习库 Mllib [11] 中选取了部分真实应用. 应用以 stage 为单位进行时间开销的记录, 因为每个 stage 都会进行迭代融合操作, 构建和匹配 kTree 数据结构. 表 2 中统计了应用所有 job 的 stage 个数, 并显示各 stage 的耗费时间之和与分析中可达的方法数之和. 其中迭代性应用只存在一次迭代的

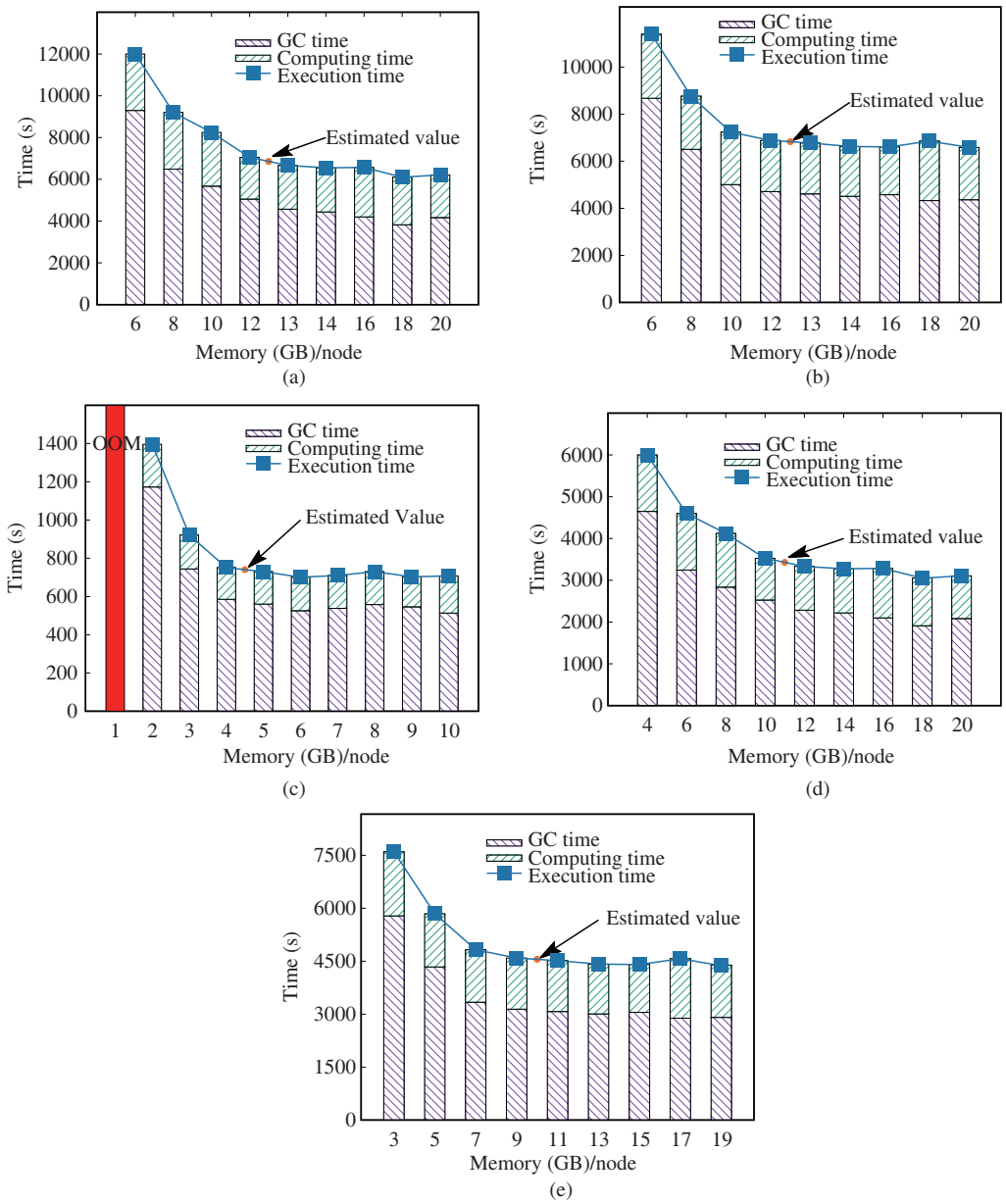

图 8 (网络版彩图) 应用 (a) PR, (b) CC, (c) WC, (d) LR, (e) KMeans 的预估合理内存阈值与真实值的误差

Figure 8 (Color online) The difference between the estimated proper memory threshold and the real one of (a) PR, (b) CC, (c) WC, (d) LR, (e) KMeans

开销, 由于预估方案会提前检测到迭代应用代码中循环体的存在, 其形成的 stage 的 kTree 都是一致的, 最后只会分析第一次迭代的 stage. 对于所有的应用来说, 程序分析所花费时间开销与大数据应用真正运行的时间相比较是可以忽略不计的, 以 PR 处理 20 G 数据的情况为例, 其分析的开销占总时间比例不足 1%. 究其原因, 大数据应用的逻辑较为简单, 用户自定义代码规模很小. kTree 的构建仅仅涉及到数据路径, 而且是细粒度地划分到每个执行 stage, 所以预估方案中程序分析的开销被尽可能降到了最低. 同时在表中可以发现, 可达的方法数目越多, 一定意义上体现了代码规模和数据路径的

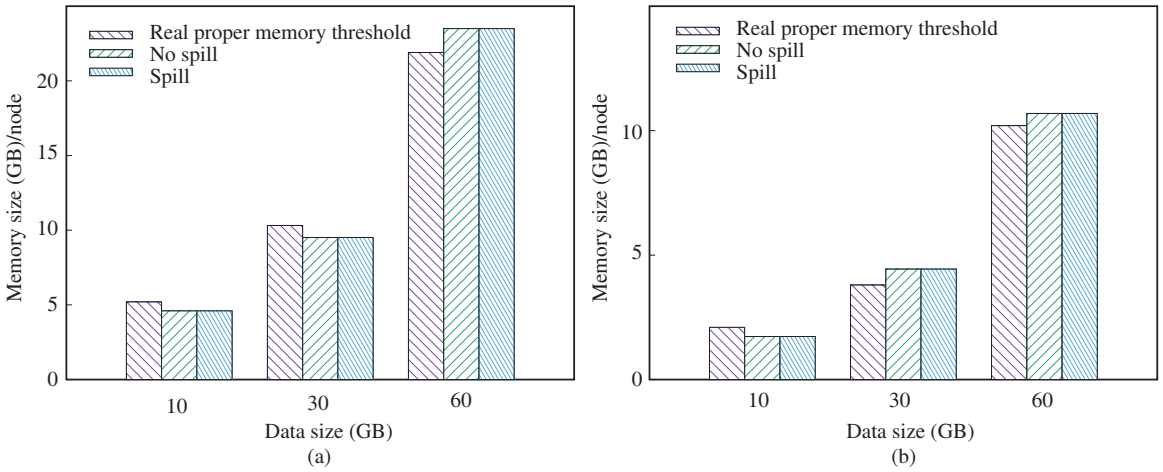

图 9 (网络版彩图) 不同数据集大小下, 应用 (a) PR, (b) WC 的预估值与合理内存阈值的误差

Figure 9 (Color online) The difference between the estimated proper memory threshold size and the real one of (a) PR, (b) WC with different datasets

表 2 程序分析开销

Table 2 The time cost of program analysis

App	Stage num	Reachable methods	Fusion time (ms)	Match time (ms)	Total time (ms)
SparkHdfsLR	1	5164	686	21369	22055
SparkKMeans	3	6352	963	21929	29244
SparkPR	3	1526	976	8643	9619
SparkCC	5	1390	1275	12756	14031
SparkWC	2	67	685	3773	4458
Normalizer	2	2440	834	11359	12193
LinearRegression	5	3003	1251	19677	20928
TallSkinnySVD	3	1998	1109	11642	12761
StandardScaler	4	3851	1245	19597	20842
ChiSqSelector	4	4301	1190	18841	24332

复杂度越大, 代码规模和数据路径的复杂越大, 时间开销也会越大.

4.4 多应用场景

为了验证预估和优化方案在多数据集、多应用同时运行的平台中优化性能的有效性, 本小节实验中模拟了在固定总内存的情况下, 有无实现预估优化内存配置方案对平台上多应用运行性能的影响. 本小节实验选取了更多的真实应用, 实验结果表明对于 MLlib 中的真实应用, 本方法依然可以成功预估合理内存阈值. 例如 Normalizer, StandScaler, ChiSqSelector 和 LogisticRegression 应用, 虽然这些机器学习应用逻辑较为复杂且各不相同, 但它们缓存的数据的数据路径都较为简单, 所以当数据路径出现一致和相似的情况, 预估优化方法在这些实例中也会有效.

平台中包含 4 个计算节点, 每个节点配置 60 GB 的最大内存, 共 240 GB 大小的总内存. 其中运行在上面的应用包括 PR, WC 和上段中提及的可预估合理内存阈值的真实应用 Normalizer, StandScaler.

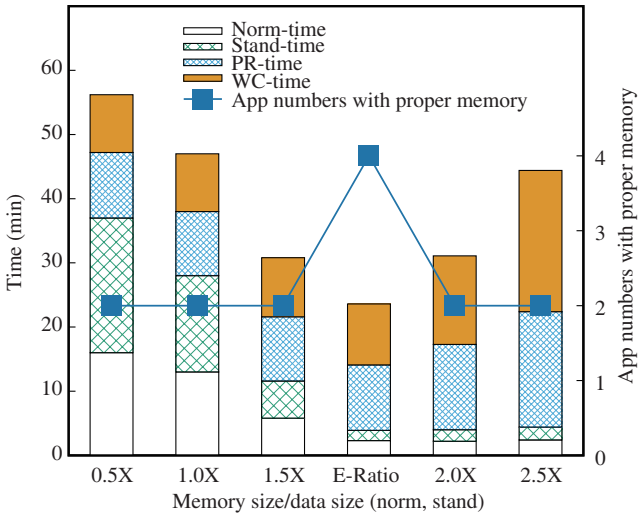

图 10 (网络版彩图) 平台上 PR, CC, Normalizer 和 StandScaler 应用在不同内存比例下的性能表现

Figure 10 (Color online) The performance of PR, CC, Normalizer and StandScaler with different memory ratio in Spark platform

PR 和 WC 运行的数据集大小为 40 GB, 为了均衡运行时间的差距, 将 PR 运行的迭代次数调整为 1 次. Normalizer, StandScaler 处理大小为 30 GB 的标签数据集, 数据均由 Hibench 随机产生. 将 4 个应用提交到 Spark 平台上, 实验将 Normalizer 和 StandScaler 的内存大小配置分别为其处理数据集大小的 0.5X, 1X, 1.5X, 2.0X 和 2.5X, 这几种比例反映了开发者会采取的内存配置方案, PR 和 CC 应用使用平台剩余的内存空间. 应用开启 Shuffle Spill, 将 Cache 的缓存级别设置为 MEMORY_AND_DISK. 如图 10 所示, 实验分别记录手动配置各个比例的内存值和在平台中运用预估方法时的所有应用运行时间的结果, 其中 E-Ratio 为应用预估方法所给出的分配方案, 换算成图中横坐标的比例约为 1.8X. 从图中可以明显发现, 平台应用预估和优化内存方案时, 整体性能明显优越于其他手动配置情况下的表现, 总时间分别是 0.5X 情况下的 44.2%, 2.5X 情况下的 56.7%, 且只有在 `E-Ratio` 下, 达到合适内存阈值的应用数量为 4 个, 其余内存比例下都存在应用性能恶化的情况. 其中 0.5X 的内存配置下, 由于 Normalizer, StandScaler 的内存空间严重不足, 运行时产生大量的 GC 和数据读写磁盘操作, 平台的整体性能受到严重恶化. 当比例逐渐升高时, PR 与 CC 可使用内存空间依旧充足, Normalizer, StandScaler 应用性能逐渐改善, 但当比例为 2.0X 和 2.5X 时, 它们的内存空间很充足, 而 PR 和 CC 可使用的内存已经不足, 所以 PR 与 CC 的性能迅速恶化. 实验结果表明, 本方法在多应用多数据集场景依然能够更准确, 更加细粒度地预估应用的内存合理阈值.

5 相关工作

本文提到的优化方案利用了程序分析的方法去解决大数据应用的内存利用问题, 业界在这两个方向上都提出过一些优化大数据应用性能的方案, 下面分别从这两个方面介绍研究进展:

内存利用. 内存利用垃圾回收策略的优化可以很好地解决内存利用不合理的问题, 尤其是内存不足的时候. Bruno 等 [18] 提出了一种针对大数据内存管理的 N 代垃圾回收器, 可以有效地减少垃圾回收过程中的对象拷贝, 也就是大幅度缩短 GC 时间. 但优化垃圾回收需要对大数据框架很深的理解, 并且了解大数据应用本身的特性. Nguyen 等 [15] 提出了一种基于生命周期来管理大数据中对象的方

式, 它通过对应用中的代码进行标记来表示对象生命周期的开始和结束, 通过框架来进行内存的分配和回收. 这样防止了无用的缓存对象长期占用内存空间, 使得内存的分配回收更加精准. 也有研究者和本文类似采用通过内存预估的方法来高效利用内存资源, 他们通过训练机器学习模型去预测应用运行期间使用的内存 [19], 然后更改 JVM 来解除内存上限的限制, 在多个应用运行期间动态地调整 JVM 的堆大小. 不过机器学习这种黑箱模型, 可能会出现严重的误差, 并且每一个应用都需要进行模型的训练才能进行预测.

程序分析与优化. Ackermann 等 [20] 通过使用语言嵌入, 多阶段编程来分析用户程序的结构. 分析的结果被用来应用投影注入, 消除无用的数据, 以及操作融合, 可以高度优化程序的关键路径. 这种利用传统编译优化的方式应用到大数据应用中比普通的大数据框架具有更优越的性能. Garbervetsky 等 [21] 和 Zhang 等 [22] 在大数据处理系统中利用程序分析来减少昂贵的冗余操作. Deca [6,23] 基于 Spark 进行二次开发, 通过指针分析来获得程序中用户自定义对象的类型和层次结构, 然后将长生命周期对象拆解转换成字节数组形式, 并根据对象生命的周期来进行特定的内存管理. 结果是大数据应用运行期间使用更少的内存空间执行任务, 并且消除大部分的 GC. 而本文的预估方法不需要做具体的代码转换.

参数调优. Herodotou 等 [24] 针对长期运行的 Hadoop 作业, 收集作业运行期间 Hadoop 系统的各种运行时特征如 Map 的 Record 数目、所用的 buffer 大小, 基于 Hadoop 框架设计出一套时间预测引擎, 最后通过这套时间预测引擎为相同的作业预测最优的 Hadoop 参数组合. Yu 等 [25] 针对长期运行的 Spark 作业, 且输入的数据集大小相似的情况, 首先使用不同的参数组合来运行目标程序以获取运行时间, 然后根据参数组合以及对应的运行时间作为训练集, 利用机器学习训练出时间预测模型, 最后使用遗传算法找出最优参数组合, 这些工作都是针对长期运行的相同作业, 只是数据集大小不同或者数据集大小相似的场景.

6 讨论

本文提出的方法利用不同的作业之间, 数据集的相似性, 以及数据处理路径的相似性来对占绝大部分内存总量的 ShuffleBuffer 以及 Cache 内存阈值进行一个合理预估. 这样的场景多出现在用户对同一个作业多次运行以保证准确性, 调整作业运行参数获取作业最佳运行结果, 或者对同一份数据应用不同的算法从多个维度挖掘数据的价值. 在用户作业提交后, 作业运行前截获用户代码, 利用 Soot 等工具自动进行代码分析, 与历史的作业进行匹配, 计算得出当前作业的合理内存阈值, 整个过程不需要人工干预. 对于一些更复杂的用户场景, 比如不同的作业数据集差异较大, 用户的业务逻辑也各不相同, 在这样的情况下, 我们无法根据之前的历史数据预估作业的合理内存阈值, 但是在一个作业的执行期间, 内存的增长或减少的幅度是可以监控的, 在未来的研究里, 如何根据作业执行期间的内存增长或减少速率来动态伸缩内存资源也是一个很好的切入点.

7 结论

本文的主要目标是消除大数据平台中批处理应用内存配置不合理的问题. 本文分析了大数据应用每个子阶段关键缓存数据的数据路径, 大数据处理应用中数据路径通常较为简单, 接着将数据路径以 kTree 这种数据结构持久化到内存和磁盘中, 便于不同大数据应用和子阶段的数据路径进行匹配和比较. 并在应用运行期间获取关键的数据特征并持久化到磁盘中. 通过截获应用匹配新应用中 kTree 和

历史的 kTree 来预估应用每个子阶段的合理内存阈值, 最后得出整个应用的合理内存阈值, 以使得平台上各应用都尽可能使用最合理的内存资源来提升平台的资源利用率和性能表现.

参考文献

1 Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, 2012. 2

2 Carbone P, Katsifodimos A, Ewen S, et al. Apache Flink: stream and batch processing in a single engine. In: Proceedings of Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015. 38: 28–38

3 Deutsch L P, Bobrow D G. An efficient, incremental, automatic garbage collector. Commun ACM, 1976, 19: 522–526

4 Thusoo A, Sarma J S, Jain N, et al. Hive: a warehousing solution over a map-reduce framework. In: Proceedings of VLDB, Endow, 2009. 1626–1629

5 Armbrust M, Xin R S, Lian C, et al. Spark SQL: relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, New York, 2015. 1383–1394

6 Lu L, Shi X, Zhou Y, et al. Lifetime-based memory management for distributed data processing systems. In: Proceedings of the VLDB Endowment, New Delhi, 2016. 936–947

7 Nguyen K, Fang L, Xu G, et al. Yak: a high-performance big-data-friendly garbage collector. In: Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16), Savannah, 2016. 349–365

8 Shi X H, Chen M, He L G, et al. Mammoth: gearing hadoop towards memory-intensive mapreduce applications. IEEE Trans Parall Distrib Syst, 2015, 26: 2300–2315

9 Ananthanarayanan G, Agarwal S, Kandula S, et al. Scarlett: coping with skewed content popularity in mapreduce clusters. In: Proceedings of the 6th Conference on Computer Systems, Salzburg, 2011. 287–300

10 Gonzalez J E, Xin R S, Dave A, et al. GraphX: graph processing in a distributed dataflow framework. In: Proceedings of Symposium on Operating Systems Design and Implementation, Broomfield, 2014. 599–613

11 Meng X R, Bradley J, Yavuz B, et al. Mllib: machine learning in apache spark. J Machine Learn Res, 2016, 17: 1235–1241

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

13 Huang S, Huang J, Liu Y, et al. Hibench: a representative and comprehensive hadoop benchmark suite. In: Proceedings of ICDE Workshops, 2010. 41–51

14 Murray D G, Isard M, Yu Y. Steno: automatic optimization of declarative queries. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, California, 2011. 121–131

15 Nguyen K, Wang K, Bu Y, et al. FACADE: a compiler and runtime for (almost) object-bounded big data applications. In: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems, Istanbul, 2015. 675–690

16 Cheng R, Hong J, Kyrola A, et al. Kineograph: taking the pulse of a fast-changing and connected world. In: Proceedings of the 7th ACM European Conference on Computer Systems, Bern, 2012. 85–98

17 Dietrich C, Rothberg V, Füracker L, et al. cHash: detection of redundant compilations via AST hashing. In: Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference, 2017. 527–538

18 Bruno R, Ferreira P. Alma: GC-assisted JVM live migration for Java server applications. In: Proceedings of the 17th International Middleware Conference, Trento, 2016. 19–20

19 Wang J, Balazinska M. Elastic memory management for cloud data analytics. In: Proceedings of 2017 Annual Technical Conference, Santa Clara, 2017. 745–758

20 Ackermann S, Jovanovic V, Rompf T et al. Jet: an embedded DSL for high performance big data processing. In: Proceedings of International Workshop on End-to-end Management of Big Dat, Raleigh, 2012. 206–220

21 Garbervetsky D, Pavlinovic Z, Barnett M, et al. Static analysis for optimizing big data queries. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, New York, 2017. 932–937

22 Zhang J, Zhou H, Chen R, et al. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In: Proceedings of NSDI, San Jose, 2012. 12: 22

23 Shi X H, Ke Z X, Zhou Y L, et al. Deca: a garbage collection optimizer for in-memory data processing. ACM Trans Comput Syst, 2019, 36: 1–47
24 Herodotou H, Dong F, Babu S. Mapreduce programming and costbased optimization? Crossing this chasm with starfish. In: Proceedings of the Vldb Endowment, 2012. 4: 1446–1449
25 Yu Z, Bei Z, Qian X. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In: Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems, Williamsburg, 2018. 564–577

Estimating the memory consumption of big data applications based on program analysis

Zhenyu HU, Xuanhua SHI*, Zhixiang KE, Hai JIN & Fei WANG

National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074*, China*
* Corresponding author. E-mail: xhshi@hust.edu.cn

Abstract Many distributed in-memory data-processing systems such as Flink and Spark suffer from serious memory issues, including limited memory resources shared by many users or groups, which aggravates the competition for memory resources. If a user application is allocated with insufficient memory, significant garbage collection overheads will occur at runtime. In contrast, if the user application is provided with a memory space larger than it actually requires, memory resources will be wasted. Therefore, it is important to ensure that a user application is allocated with an appropriate memory size. In a general case, multiple applications process one specific set of data repeatedly. Often, the process logic of applications working on the same dataset is similar; for example, they can perform machine learning or graph computing tasks. This study further reveals that the process logic reflected in application programming interface and user-defined functions also affects the memory usage at runtime. Based on this observation, this paper presents a method for estimating the optimal memory size for newly submitted applications. The proposed approach was implemented on Spark. It utilizes the information of program analysis and historical applications produced when processing data to estimate a proper memory threshold for a newly submitted application based on the similarity of the data path between the new application and a historical one. The results of the experiments conducted to evaluate the method's accuracy of estimating the memory threshold and performance profit demonstrated that the proposed method is able to (1) estimate the required memory threshold with an error of 4% compared to the actual proper memory threshold, (2) guarantee the overall time overhead of estimating to be negligible compared to the execution time of a submitted job, and (3) reduce the execution time of submitted applications by up to 56% compared to when the proposed method is not applied.

Keywords memory management, lifetime, program diagnostics, data-processing system, distributed system

Zhenyu HU received his B.A. degree in computer science from Huazhong University of Science and Technology, Wuhan, China, in 2017. Currently, he is a master student in the National Engineering Research Center for Big Data Technology and System/Service Computing Technology and System Lab at Huazhong University of Science and Technology, China. His research interests include in-memory computing and big data processing.

Zhixiang KE received his B.A. degree in computer science from Huazhong University of Science and Technology, Wuhan, China, in 2015. Currently, he is a master student in the National Engineering Research Center for Big Data Technology and Service Computing Technology and System Lab at Huazhong University of Science and Technology, China. His current research interests include in-memory computing and big data processing.

Xuanhua SHI received his Ph.D. degree in computer engineering from Huazhong University of Science and Technology, China, in 2005. He is a professor in the National Engineering Research Center for Big Data Technology and Service Computing Technology and System Lab at Huazhong University of Science and Technology, China. From 2006, he worked as an INRIA post-doc in the PARIS Team with Rennes for one year. His current research interests focus on cloud computing and big data processing.

Hai JIN received his Ph.D. degree in computer engineering from Huazhong University of Science and Technology in 1994. He is a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technol- ogy (HUST) in China. He is the director of the National Engineering Research Center for Big Data Technology and Service Computing Technology and System Lab at Huazhong University of Science and Technology, China. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.