**密级:**



**博士学位论文**

**分布式数据并行应用内存溢出错误分析与诊断**

**作者姓名:**  许利杰

**指导教师:**  魏峻 研究员

中国科学院软件研究所

**学位类别:**  工学博士

**学科专业:**  计算机软件与理论

**培养单位:**  中国科学院软件研究所

**2015 年 10 月**

**Understanding and Diagnosing Causes of Out of Memory Errors in Distributed Data-parallel Applications**

**By**

**Lijie Xu**

**A Dissertation Submitted to**

**University of Chinese Academy of Sciences**

**In partial fulfillment of the requirement**

**For the degree of**

**Doctor of Computer Science**

Institute of Software

Chinese Academy of Sciences

**October 2015**

**独创性声明**

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明。

签名：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 日期：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**关于论文使用授权的说明**

本人完全了解中国科学院软件研究所有关保留、使用学位论文的规定，即：中国科学院软件研究所有权保留送交论文的复印件，允许论文被查阅和借阅；中国科学院软件研究所可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。

(保密的论文在解密后应遵守此规定)

签名：\_\_\_\_\_\_\_\_\_\_ 导师签名：\_\_\_\_\_\_\_\_\_\_ 日期：\_\_\_\_\_\_\_\_\_\_

摘要

大数据以数据量大、数据类型多样、产生与处理速度要求快、价值高的4V特性迅速成为工业和学术界关注的焦点。由于传统软件系统难以在可接受的时间范围内处理大数据，因此分布式数据并行处理框架如Hadoop，Spark等，成为了解决该问题的核心软件系统，并在工业界得到广泛应用。分布式数据并行应用指然而在框架处理大数据时面临诸多内存管理方面的问题： 如内存溢出频繁却不知道原因，内存消耗影响因素多且难以预知，大内存使用和回收效率低。本文旨在研究和解决分布式处理框架在内存分配、使用、回收方面的问题，分别从内存分析与预测、错误诊断与修复、内存管理机制改进方面开展研究。具体包括采用统计、模拟和规则方法对处理框架中的内存使用进行建模与预测，提出框架与应用的内存溢出错误诊断与修复方法，改进JVM内存分配，改进框架中间数据存储与处理方式，最终从可用性、容错性、效率方面提高框架处理数据性能。

分布式数据并行应用广泛内存溢出错误常见。用户不知道错误原因，本论文实证分析了内存溢出错误原因，具有良好扩展性的分布式处理框架现已被广泛应用于企业和科研机构的大数据处理与分析当中。然而，目前的主流框架如Hadoop、Spark等，在内存管理方面均存在性能和可靠性问题。具体表现在任务执行时内存消耗量大且难以预知，内存溢出频繁却难以定位发生原因，存在内存膨胀现象且内存回收效率低等问题。本文针对这几个问题，分别从内存消耗分析与预测、内存溢出错误诊断与内存管理机制改进三个方面开展具体研究。当前，已经完成内存消耗模型的建立与消耗量预测；已经对内存溢出错误进行了分析，总结出了诊断方法；在内存管理机制方面有一定分析，有待进一步深入研究。

本论文针对分布式数据并行应用，主要贡献如下：

**1) 内存溢出错误的实证分析：**分析了内存溢出错误原因，修复方法，及潜在的可以提高框架错误容忍能力，方便用户进行错误诊断的方法。

**2) 内存用量模型构建与用量估算：**建立应用内存用量模型，可以估算内存用量。

**3) 内存溢出错误的诊断方法：**提出一种半自动化诊断方法帮助用户诊断内存溢出错误。

本论文的研究成果对用户和框架设计者均有参考意义，可进一步应用于。综合上述研究成果，论文设计并实现了。该系统主要由。

**关键词：**分布式数据并行应用，MapReduce，内存溢出，分析诊断

**Unnderstanding and Diagnosing Causes of Out of Memory Errors in Distributed Data-parallel Application**

Lijie Xu (Computer Software and Theory)

Directed by Professor Jun Wei

Abstract

Out of memory (OOM) errors occur frequently in MapReduce applications, where memory space is shared by the MapReduce framework and user code. However, current memory profilers and memory leak detectors can only figure out what objects are (unnecessarily) persisted in memory, but cannot figure out where the objects come from and why they become so large.

In this paper, we observe that improper job configurations, data skew, and memory-consuming user code can lead to the excessive memory usage in MapReduce applications. Based on this observation, we propose a semi-automatic dataflow- centric memory backtracking approach to diagnose OOM errors in MapReduce applications. Our approach first automatically quantifies the relationship among memory usage, runtime data, and the configurations. Then, it uses quantitative rules to trace the OOM error back to the problematic user code, data, or configurations.

The evaluation on 20 real-world OOM errors in diverse Hadoop MapReduce applications shows that our approach can accurately identify the root causes of 15 OOM errors, and partly identify the root causes of 5 OOM errors where the user code is written by high-level languages.

**Keywords:** Data-parallel applications, MapReduce, Out of memory, Error diagnosis

目录

[摘要 I](#_Toc369296357)

[Abstract III](#_Toc369296358)

[目录 VII](#_Toc369296359)

[第一章 绪论 1](#_Toc369296360)

[1.1 研究背景 1](#_Toc369296361)

[1.2 问题分析 2](#_Toc369296362)

[1.3 论文工作 3](#_Toc369296363)

[1.3.1 研究思路 3](#_Toc369296364)

[1.3.2 研究内容 4](#_Toc369296365)

[1.4 论文组织 5](#_Toc369296366)

[第二章 分布式缓存技术研究综述 7](#_Toc369296367)

[2.1 分布式缓存概述 7](#_Toc369296368)

[2.1.1 分布式缓存的特性 7](#_Toc369296369)

[2.1.2 典型应用场景 8](#_Toc369296370)

[2.1.3 分布式缓存的发展 8](#_Toc369296371)

[2.1.4 分布式缓存与NoSQL 9](#_Toc369296372)

[2.1.5 分布式缓存与极限事务处理 9](#_Toc369296373)

[2.1.6 相关标准规范 10](#_Toc369296374)

[2.2 分布式缓存技术分析 10](#_Toc369296375)

[2.2.1 弹性资源供给 12](#_Toc369296376)

[2.2.2 可用性与可靠性 12](#_Toc369296377)

[2.2.3 敏捷性与自适应性 13](#_Toc369296378)

[2.2.4 多租户 14](#_Toc369296379)

[2.2.5 数据管理 15](#_Toc369296380)

[2.2.6 数据安全与隐私防护 19](#_Toc369296381)

[2.3 典型系统 19](#_Toc369296382)

[2.3.1 Oracle Coherence 19](#_Toc369296383)

[2.3.2 Memcached 20](#_Toc369296384)

[2.3.3 Terracotta 20](#_Toc369296385)

[2.3.4 Microsoft AppFabric Caching 20](#_Toc369296386)

[2.3.5 典型系统比较 21](#_Toc369296387)

[2.4 云计算环境下分布式缓存面临的新挑战 22](#_Toc369296388)

[2.4.1 缓存策略的自适应与自管理 22](#_Toc369296389)

[2.4.2 虚拟化环境下缓存数据迁移的优化 23](#_Toc369296390)

[2.4.3 缓存服务的性能隔离 23](#_Toc369296391)

[2.5 研究趋势 23](#_Toc369296392)

[2.6 本章小结 24](#_Toc369296393)

[第三章 基于机器学习的缓存策略自适应重配方法 25](#_Toc369296394)

[3.1 引言 25](#_Toc369296395)

[3.2 问题分析 26](#_Toc369296396)

[3.2.1 缓存策略的评价与分析 26](#_Toc369296397)

[3.2.2 缓存策略自适应面临的挑战 28](#_Toc369296398)

[3.3 关键机制 28](#_Toc369296399)

[3.3.1 总体架构 28](#_Toc369296400)

[3.3.2 数据预处理 29](#_Toc369296401)

[3.3.3 模型的参数选择 31](#_Toc369296402)

[3.3.4 性能模型的训练 33](#_Toc369296403)

[3.3.5 性能模型的持续优化 33](#_Toc369296404)

[3.3.6 缓存策略重配置 35](#_Toc369296405)

[3.3.7 缓存策略重配开销的优化 37](#_Toc369296406)

[3.4 实验与评价 37](#_Toc369296407)

[3.4.1 实验设置和评价方法 38](#_Toc369296408)

[3.4.2 缓存策略重配的有效性 39](#_Toc369296409)

[3.4.3 缓存策略重配周期的调整 40](#_Toc369296410)

[3.4.4 监测开销评价 42](#_Toc369296411)

[3.4.5 性能模型的优化 42](#_Toc369296412)

[3.4.6 缓存策略重配开销的优化 43](#_Toc369296413)

[3.4.7 扩展性评价 44](#_Toc369296414)

[3.4.8 进一步的讨论 45](#_Toc369296415)

[3.5 相关工作比较 45](#_Toc369296416)

[3.5.1 基于模拟的方法 46](#_Toc369296417)

[3.5.2 基于规则的方法 46](#_Toc369296418)

[3.5.3 基于机器学习的方法 47](#_Toc369296419)

[3.5.4 其他方法 47](#_Toc369296420)

[3.6 本章小结 48](#_Toc369296421)

[第四章 性能干扰与开销敏感的缓存数据迁移方法 49](#_Toc369296422)

[4.1 引言 49](#_Toc369296423)

[4.2 问题分析 50](#_Toc369296424)

[4.2.1 Xen 的I/O架构 50](#_Toc369296425)

[4.2.2 虚拟化环境对缓存数据迁移的影响 51](#_Toc369296426)

[4.3 关键机制 53](#_Toc369296427)

[4.3.1 性能干扰感知的预测模型 54](#_Toc369296428)

[4.3.2 数据迁移开销模型 56](#_Toc369296429)

[4.3.3 开销敏感的数据迁移算法 57](#_Toc369296430)

[4.4 系统实现 59](#_Toc369296431)

[4.4.1 负载均衡度检测 59](#_Toc369296432)

[4.4.2 预测模型的训练 60](#_Toc369296433)

[4.4.3 预测模型的优化 60](#_Toc369296434)

[4.5 实验与评价 61](#_Toc369296435)

[4.5.1 实验环境和设置 61](#_Toc369296436)

[4.5.2 预测模型的性能 62](#_Toc369296437)

[4.5.3 预测模型的鲁棒性 62](#_Toc369296438)

[4.5.4 数据迁移算法的有效性 63](#_Toc369296439)

[4.5.5 数据迁移算法的扩展性 65](#_Toc369296440)

[4.5.6 迁移开销模型的有效性 65](#_Toc369296441)

[4.5.7 进一步的讨论 66](#_Toc369296442)

[4.6 相关工作比较 66](#_Toc369296443)

[4.7 本章小结 67](#_Toc369296444)

[第五章 面向多租户的云缓存服务性能隔离方法 69](#_Toc369296445)

[5.1 引言 69](#_Toc369296446)

[5.2 问题分析 70](#_Toc369296447)

[5.2.1 研究背景 70](#_Toc369296448)

[5.2.2 内存资源竞争 71](#_Toc369296449)

[5.2.3 网络资源竞争 72](#_Toc369296450)

[5.3 关键机制 73](#_Toc369296451)

[5.3.1 总体架构 73](#_Toc369296452)

[5.3.2 缓存空间划分机制 74](#_Toc369296453)

[5.3.3 租户敏感的缓存替换算法 76](#_Toc369296454)

[5.3.4 租户带宽分配机制 76](#_Toc369296455)

[5.4 系统实现 78](#_Toc369296456)

[5.5 实验与评价 78](#_Toc369296457)

[5.5.1 实验设置和评价方法 79](#_Toc369296458)

[5.5.2 性能隔离框架的有效性 80](#_Toc369296459)

[5.5.3 缓存空间划分评价 81](#_Toc369296460)

[5.5.4 租户带宽分配评价 81](#_Toc369296461)

[5.5.5 性能开销 82](#_Toc369296462)

[5.5.6 进一步的讨论 82](#_Toc369296463)

[5.6 相关工作比较 83](#_Toc369296464)

[5.7 本章小结 84](#_Toc369296465)

[第六章 系统原型实现 85](#_Toc369296466)

[6.1 系统总体架构 85](#_Toc369296467)

[6.2 缓存服务器 86](#_Toc369296468)

[6.2.1 请求处理 86](#_Toc369296469)

[6.2.2 数据迁移管理 87](#_Toc369296470)

[6.2.3 性能监测与统计 87](#_Toc369296471)

[6.3 节点管理器 88](#_Toc369296472)

[6.3.1 缓存节点管理 89](#_Toc369296473)

[6.3.2 资源监测与统计 90](#_Toc369296474)

[6.3.3 缓存节点状态管理 91](#_Toc369296475)

[6.4 主节点 91](#_Toc369296476)

[6.4.1 缓存集群管理 91](#_Toc369296477)

[6.4.2 节点部署 93](#_Toc369296478)

[6.4.3 管理控制台 94](#_Toc369296479)

[6.5 缓存客户端 94](#_Toc369296480)

[6.5.1 命令处理 95](#_Toc369296481)

[6.5.2 配置管理 96](#_Toc369296482)

[6.6 关键技术 96](#_Toc369296483)

[6.7 本章小结 98](#_Toc369296484)

[第七章 结束语 99](#_Toc369296485)

[7.1 论文工作总结 99](#_Toc369296486)

[7.2 进一步的工作 100](#_Toc369296487)

[参考文献 103](#_Toc369296488)

[发表文章目录 113](#_Toc369296489)

[参与科研项目目录 115](#_Toc369296490)

[获得发明专利目录 117](#_Toc369296491)

[获得软件著作权目录 119](#_Toc369296492)

[致 谢 121](#_Toc369296493)

# 绪论

本章简要介绍了论文的研究背景，并分析了论文研究的三个主要问题，接下来给出了论文的研究思路和研究内容。在本章的结尾，列出了论文的组织方式。

* 1. 研究背景

### 大数据与大数据处理

数据一直是工业界和学术界的重要研究对象。从数据中获益，从数据中获取知识成为数据处理的主要目的。例如，搜索引擎每天都在收集、处理、分析海量的网页及多媒体数据，并对外提供数据查询服务。社交网站每天记录大量的用户数据，组织形成虚拟的人际网络。商业智能公司依靠分析企业生产和销售的数据，来为企业提供商务决策支持。学术研究机构也在天文、物理、化学、地理、生命科学等方面不断积累大量的实验数据，从数据中分析挖掘各种科学知识。

互联网、云计算、移动计算、物联网等技术的发展使得数据的产生速度越来越快、数据规模越来越大、数据类型越来越多。诸如社交网站Facebook每天要处理25亿条消息，500+TB的新数据，用户每天上传32亿张照片，每半个小时扫描的数据大约为105TB [FB]。早在2008年，Google每天就要处理约20,000 TB（20PB）的数据。Youtube网站每分钟用户会上传越48小时的视频。Twitter每天大约有1.75亿条新微博 [Youtube]。

为了描述数据的新特性，“大数据”的概念被提出。大数据具有数据量大、数据类型多样、产生与处理速度要求快、价值高的4V特性。这些特性也使得传统软件系统难以在可接受的时间范围内对大数据进行获取、管理和处理。例如，出现于70年代的关系数据库很好地解决了关系型数据的存储、管理以及OLTP（在线事务处理）。之后出现的数据仓库将数据按主题组织，解决了数据建模以及OLAP（在线数据分析）。然而由于传统数据库和数据仓库在扩展性方面的问题，使得大数据的处理与分析变得困难。为了解决这个问题，工业届和学术界设计了可扩展的分布式数据并行处理框架，这些被提出并得到广泛应用。

### 分布式数据并行处理框架

为了解决大数据的有效高效处理，工业界和学术界提出了很多分布式并行处理框架。分布式数据并行框架采用“分治－聚合”策略来对大数据的分布处理。以Google MapReduce为代表，包含非开源框架Microsoft Dryad，Apache Spark等。框架的运行模式是MapReduce-like。拥有共同的编程范型。

2004年Google在OSDI会议上提出了基于分治、归并、函数式编程思想的MapReduce分布式计算框架，获得巨大成功。之后模仿GFS和MapReduce实现的开源系统Apache Hadoop将大数据的分布式存储与处理技术推广到了工业界和学术界，并迅速成为大数据的主流存储与处理平台。2007年微软公司提出了Dryad分布式计算框架，虽然是基于MapReduce，但与MapReduce固定数据处理流程不同的是Dryad允许用户将任务组织成DAG（有向无环图），任务结构更加灵活，表达能力也更强。2012年UC Berkeley的AMPLab提出了基于内存，适合迭代计算的Spark[[1]](#endnote-1)分布式大数据处理框架。该框架允许用户将公共或中间处理数据缓存到内存，极大缩短了数据处理的时间。

### 分布式数据并行应用

运行在分布式处理框架之上的应用被成为分布式数据并行应用。

大数据分布式处理的框架的产生使得上层的大数据分析应用成为可能。Yahoo开发了从SQL语言到MapReduce job的转换框架。Facebook开发了建立在Hadoop之上的分布式数据仓库Hive，核心也是将类SQL查询分析语言转变成MapReduce job。Mahout是建立Hadoop之上的机器学习框架。HBase是模仿Google BigTable，建立在Hadoop分布式文件系统HDFS上的Key-Value数据库。应用被广泛应用于网页索引构建。分布式数据并行应用指的是运行在分布式数据并行框架之上，对大数据进行分布处理的作业（Job）。下面本文简称为“数据并行应用”。数据并行应用广泛应用于网页索引构建、日志挖掘、大数据SQL语句执行、机器学习、社交网络图分析等。

### 应用运行的三层结构

整个应用从开发到运行产生结果可以拥有三层结构，用户层、框架执行层、物理内存层。

用户层负责撰写应用，应用（Job）的组成元素是 <data，configurations，user code>。数据（data）指的是输入的大数据，大数据一般分布存放在分布式文件系统（如Hadoop FileSystem上）。配置参数（configurations）用于指定框架运行时需要的信息，如partition个数。用户代码包含map(), reduce()以及可选的combine()。

框架执行层负责job的执行，一个job包含多个map/reduce 任务（task）。框架负责启动每个task，并按照分布式数据流的方式执行固定的处理步骤。如在Hadoop中的处理阶段可以分为map，shuffle，reduce等阶段。数据流指运行时的输入／输出／中间数据，以及数据以来关系。每个task是一个进程或线程，比如Hadoop MapReduc中每个task运行在一个JVM中。

物理内存层反映应用任务（task）的真实内存消耗，应用的内存使用情况是框架使用一部分内存，用户代码也消耗一部分内存。框架在内存暂存一些数据，用户代码在处理输入数据时也会消耗内存。

* 1. 论文工作

### 研究问题及研究动机

尽管框架为用户提供了简单的编程范型、编程接口，用户不需要具有分布式系统的知识也可以使用。这种设计方式方便用户只关注于处理逻辑本身，无需考虑应用是如何分布执行的及中间处理过程。这种方式方便用户撰写数据逻辑，但对用户来说，如果应用出现运行时错误，那么对错误的诊断和修复都是困难的。内存溢出错误是一种常见的的错误，由于应用会在内存中处理大数据。内存溢出错误目前还不能被框架的错误容忍机制处理（tolerate）。除了内存溢出问题，我们也发现了用户不明白应用的消耗。

**现象1: 内存溢出错误常见，不明白错误原因及修复方法**

例如，在程序员的公开论坛StackOverflow.com中有很多关于内存溢出的问题，比如

① 我很诧异为什么我的 job 会在 map 阶段出现内存溢出错误，内存 溢出的原因是什么？[1]

② 如何处理并修复内存溢出错误？我调了一些配置参数（如reducer数目），但不起作用[2]

③ 我的Spark程序出现了内存溢出错误，我不能找到

我们收集到 276 个 real-world OOM errors

发现各个运行阶段均存在内存溢出错误（map，shuffle，reduce）

发现各类应用均存在内存溢出（用户手写代码或高层语言产生的代码）

**现象2: 用户不明白内存用量，也不知道该为内存分配多少内存。**

① 为什么我的 map 任务会消耗如此多的内存？[1]

② 我能把任务的内存设为小于 2GB 吗？ [2]

③ 我想准确地预测出 job 的内存用量，因为内存设的不合理，我的 job 在 40 个节点的集群上跑了 5 天后失败了[3]

**现象3: 用户对框架运行不理解，手工诊断内存溢出错误困难。**

① 我的job总是内存溢出错误，这是Hadoop自己的bug吗？我不懂处理流程中哪些步骤会消耗过多内存，希望能够得到建议。

[http://stackoverflow.com/questions/17975335/hadoop-streaming-memory-usage]

② 我不是很理解MapReduce数据流的工作原理。我的一个很紧要的job在reduce阶段出现了内存溢出错误，希望大家帮我看一下为什么会出现这种错误。[http://stackoverflow.com/questions/19490723/detailed-dataflow-in-hadoops-mapreduce]。

内存溢出错误：

|  |
| --- |
| FATAL org.apache.hadoop.mapred.Child:  **Error running child : java.lang.OutOfMemoryError: Java heap space**  at java.util.Arrays.copyOf(Arrays.java:2882)  at java.lang.AbstractStringBuilder.expandCapacity(AbstractStringBuilder.java:100)  at it.unimi.dsi.fastutil.objects.AbstractObject2IntMap.toString(AbstractObject2IntMap.java:298)  ...  at org.apache.hadoop.mapred.ReduceTask$NewTrackingRecordWriter.write(ReduceTask.java:665)  at cloud9.ComputeCooccurrenceMatrixStripes$MyReducer.reduce(ComputeCooccurrenceMatrixStripes.java:136)  at org.apache.hadoop.mapreduce.Reducer.run(Reducer.java:216)  at org.apache.hadoop.mapred.Child.main(Child.java:404) |

问题本质及研究问题

**内存溢出错误原因**：影响内存用量的因素太多（有静态配置也有动态数据流），不知道哪个是内存溢出错误原因

研究问题1：有没有常见的错误原因类型及修复方法？

**用量预测**：预测难是因为难以确定静态因素（如配置，用户代码）与动态内存用量之间的关系

研究问题2：有没有内存用量模型来指导内存用量预测？

**手工诊断困难**：目前没有诊断方法，需要用户对框架，应用及内存使用都非常了解，有丰富的调试经验

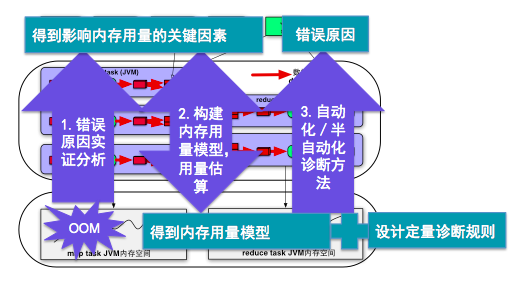
研究问题3：有没有可以直接或者辅助用户诊断的方法？

然而在应用使用分布式处理框架处理大数据时，经常面临着内存消耗未知、内存溢出、内存分配和回收效率低等内存管理问题。内存消耗方面：因为磁盘和内存的价格问题，内存容量一般要差磁盘一个数量级。内存容量有限而且所有任务的输入、输出和临时数据都要流经内存。因此内存消耗一般较大，变化较快，且受框架配置参数、数据流影响，难以分析和预测。内存溢出方面：框架不仅在内存中处理数据，而且将部分中间数据存储在内存中，加上配置参数、数据流和用户定义的程序影响，内存溢出难以诊断原因和修复。在著名的程序错误问答网站stackoverflow.com上有超过10000个question是关于“hadoop outofmemory” ，在Spark邮件列表上有111条（当前共917条）是关于“outofmemory”的。本人在实践实验过程中，也经常碰到“Heap Error”等内存错误，其中在使用Mahout的K-means算法分析20GB数据时，碰到的“Heap Error”至今也没找到出错原因。内存分配和回收方面：JVM在管理大对象和巨量对象时存在严重的GC（垃圾回收）效率问题，而框架在处理大数据时会产生大数组、生命周期不一的大量对象。由于JVM内部的分区管理问题，GC发生频繁且持续时间长。另外，JVM在管理超过130G的内存时会出现诸多问题 。

本论文主要研究大数据分布式处理框架的内存管理技术。以主流的开源分布式框架Hadoop、Spark为平台，解决框架和应用在内存分配、使用和回收方面的诸多问题。针对内存消耗影响因素多且难以预知的问题，给出内存消耗的分析方法，帮助用户和系统管理人员理解框架的内存消耗影响因素与规律；给出内存消耗建模与预测方法，帮助用户在运行job时能够提前预知内存消耗情况。针对内存溢出的问题，给出错误诊断方法，定位出错阶段和错误来源；给出错误修复方法，帮助用户或框架再次运行job时能够成功。针对内存分配和回收效率低的问题，一方面改进现有JVM的区域分配与GC机制，降低GC频率，提高大内存管理能力；另一方面改进框架的中间数据处理方式，更合理地使用内存，提升job的执行性能。总体为了提高分布式处理框架的可用性、容错性和执行效率。

### 研究思路

通过实证分析来总结内存溢出的常见原因及修复方法，得出影响内存用量的关键影响因素。通过构建内存用量模型，对内存用量进行预测。有了内存用量模型以后，设计定量规则，根据job的静态和动态信息，自底向上定位出错原因。



### 研究内容

错误原因实证分析：分析内存溢出错误原因，修复方法，及潜在可以避免内存溢出错误的方法，RQ1、RQ2、RQ3。

内存用量模型构建：建立静态因素 <data, configurations, user code> 与动态内存用量之间的关系。

内存溢出错误诊断：给定job，如何诊断出其静态因素中的错误原因，比如有问题的配置，内存消耗量高的代码段，数据流异常以及有问题的配置参数。

基于上述研究路线，论文借鉴 思想， 具有直接的意义，，也可为相关关键技术的研究提供一定的借鉴和指导。论文研究内容具体包括以下几个部分：

1. **内存溢出错误原因及修复方法实证分析**

论文研究三个问题。

1. **内存用量模型构建与内存用量估算**

论文首先研究了虚拟化环境对缓存数据迁移的影响。基于上述模型，提出了一种开销敏感的数据迁移算法。

1. **缓存服务性能隔离方法研究**

论文围绕问题展开研究。在内存资源分配方面，针对已有工作的不足，设计了一种新的缓存空间划分机制。在此基础上，提出。

### 创新点

本文针对分布式并行应用的内存用量及内存溢出错误开展研究，属于性能与可靠性问题。以前的性能问题主要研究应用的执行时间以及网络传输量优化，我们研究了内存使用问题。以前的可靠性问题主要研究一般类型的错误，我们专门研究内存溢出错误。内存溢出错误在一般的应用程序中很常见，我们相信研究也结果对未来问题有借鉴意义。

Contributions。

* 1. 已有研究工作不足

A. 大数据应用错误的实证分析

很多研究人员已经实证研究了大数据应用或大数据系统的故障错误。Li [] 等人研究了250个SCOPE job（运行在微软的Dryad框架之上）的故障错误，发现错误主要原因是未定义的列，错误的数据模式，不正确的行格式等等。他们也发现3个内存溢出错误，错误原因是在内存累积了大量的数据（比如一个大表的所有行被存放到内存中）。Kavulya等人[] 分析了4100个执行失败的Hadoop jobs，这些jobs运行在Yahoo!管理的M45集群。他们发现36%的故障是数组访问越界错误，还有23%的故障是IO异常。Xiao等人[]研究了MapReduce程序（同样是SCOPE程序）执行结果的正确性。他们研究发现5种非交换的reduce()如果运行多次会输出不一致的运行结果。

Zhou等人[]研究了微软大数据处理平台的质量问题。他们发现36%的故障是系统缺陷导致的，有2个故障（1%）与内存有关。Gunawi等人[]研究了3655个云计算系统（比如Hadoop，HBase）中出现的开发和部署故障。他们发现有一个HBse中的内存溢出错误，错误原因是用户提交的查询请求需要在大规模的数据上进行。他们也发现一个HDFS中出现的内存溢出错误，原因是用户并行地创建了上千个小文件。

B. 减少大数据应用的内存消耗

一些研究者提出了一些减少大数据应用内存消耗的办法。Tachyon []提供了一个基于内存的分布式数据存储系统，主要用于存放不同应用（jobs）产生的重用数据。用户可以将原本直接缓存到框架的数据转移存放到Tachyon上，以降低框架的内存消耗。Façade[]提供了用于降低用户代码内存消耗的用户代码编译和执行环境。Façade设计目的是将数据存储和数据操作分开，方法是将数据存放到JVM的堆外内存，将对堆内对象的数据操作转换为对Façade的函数调用。对于Java对象本身产生的overhead（也就是Java对象自身所需的header和reference），Bu等人[Bu] 提出了两种减少overhead的方法，比如将大量数据对象（record object）合并少量的大对象。然而，他们没有研究内存溢出错误的诊断问题。

C. 内存泄漏错误分析

内存泄漏可以直接导致内存溢出错误，所以之前的研究者提出了很多可以检测内存溢出错误的方法和工具。方法主要有静态检测方法和动态检测方法。内存泄漏的含义是用户忘记释放本应该释放的内存对象。在我们研究的应用中，我们还没有发现内存溢出错误。一方面原因是用户代码是由具有垃圾回收机制的语言（Java或Scala）写的。另一方面的原因是我们很难去判定用户代码持有的大数据和中间计算结果是被该释放还是不该释放（故意持有）。

D. 配置参数问题检测

研究人员提出了很多黑盒和白盒方法来检测配置参数错误。比如，PeerPressure [PeerPressure] 是一种黑盒方法，它使用统计方法来识别正常和异常节点的参数状态的变化。ConfAid [ConfAid]是一种白盒方法，可以识别出配置参数中有问题的标记（token），方法是去分析应用二进制代码的控制和数据流。这里，数据流指的是代码中的变量依赖关系（如val a = b + c中的数据流是b和c流动到a）。上面的检测方法不能被直接用于诊断内存溢出相关的配置参数，因为这些方法不能建立动态内存用量和静态配置参数之间的关系。我们的方法也是一种白盒方法：根据MapReduce数据流特点建立数据流模型，根据用户代码编程范型建立用户对象模型。

E. 数据倾斜

分布式数据并行应用存在诸多内存使用问题

1. **内存溢出错误常见**

拟解决的问题是，这其中面临两项关键挑战：一是；二是。已有研究工作，本文的研究对象为内存

1. **内存用量的建模与估算**

解决的问题。已有研究工作，如何在因此，需要找到一种方法。

1. **内存溢出错误原因诊断**

错误原因难以诊断。



图1.2论文整体研究路线

* 1. 论文组织

论文总共分为七章，其中，后续章节的组织结构和内容如下：

第二章首先分析介绍了的特性、典型应用场景、发展阶段、相关标准规范以及推动缓存技术发展的若干关键要素。在此基础上，基于该框架对当前缓存技术进行总结和分析，并对典型系统进行了比较，指出了存在的问题和不足，这些构成了本文研究的出发点。

第三章以为研究对象，关注当前问题。

第四章主要解决优化问题，提出一种性能干扰与开销敏感的数据迁移方法，重点关注VM性能干扰的刻画以及迁移时间与性能衰减值如何权衡的问题。

第五章针对 。

第六章介绍了 。

第七章总结论文工作，并对进一步的研究工作进行了展望。

# 背景知识

作为云计算环境下提升应用性能的一种重要手段，指出了存在的问题和不足，这些构成了本文研究的出发点。

* 1. 分布式数据并行应用概述

### 编程范型

什么是分布式数据并行应用？

### 用户代码

什么是分布式数据并行应用？

### 配置

什么是分布式数据并行应用？

### 数据流

### 典型系统Hadoop和Spark

# 数据并行应用内存溢出错误实证研究

摘要。

* 1. 概述

该部分实证研究试图回答以下三个研究问题：

**RQ1**：应用内存溢出的错误原因是什么？有没有一些常见的错误类型（cause pattern）？

**RQ2**：用户是怎么修复内存溢出错误的，有没有一些常见的修复策略（pattern）？

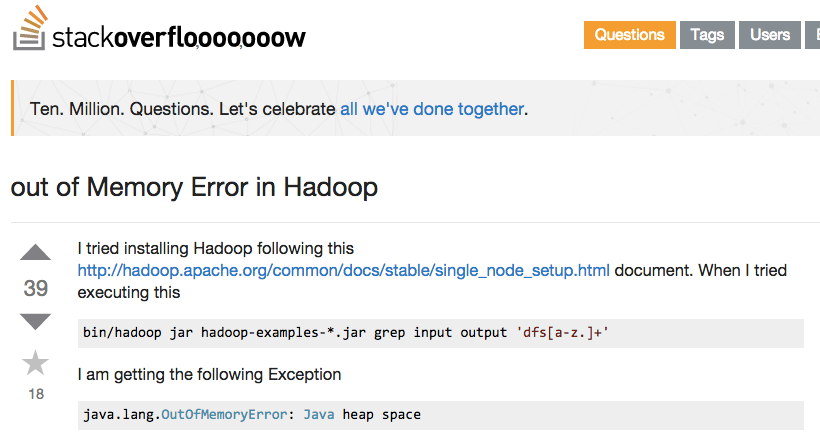
**RQ3**：有没有可以提高框架错误容忍能力的方法，有没有可以方便错误诊断的方法？

* 1. 研究方法

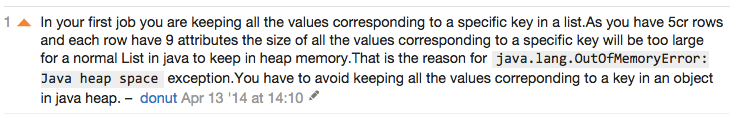
### 研究对象

我们选择运行在Apache Hadoop和Apache Spark上的分布式数据并行应用作为我们的研究对象。因为目前没有专门针对内存溢出错误的缺陷库（bug repository），用户在遇到内存溢出错误时会选择公开论坛或者Hadoop/Spark的邮件列表等公开途径来询问错误原因及修复方法。因此，我们以这些公开应用为研究对象，获得应用的方法是，我们先在公开渠搜索内存溢出错误的关键字，比如“Hadoop out of memory”，“Hadoop outofmemory”，“Spark OOM”，在StackOverflow.com，Hadoop邮件列表[HadoopMail]，Spark users/dev邮件列表[SparkMail]，开发者的博客，和两本关于MapReduce的畅销书籍中的例子[MBook]。

错误描述



专家回答



通过关键词搜索，我们一共发现1151个例子（issue），我们手工检测每个例子看是会否满足下列三个条件：

1. 例子是Hadoop/Spark的内存溢出错误。

其中有786例子不是内存溢出错误，比如只包含部分的关键字“Hadoop Memory”，只讲Hadoop Memory的配置方法。

1. 内存溢出错误发生在Hadoop/Spark应用里

有些内存溢出错误发生在Hadoop/Spark系统本身，比如发生在Hadoop调度器或Hadoop的资源管理器（TaskTracker）里。

经过筛选，最后留下276个错误例子，我们发现其中123个例子的错误原因已经下面的方法诊断出来，所以我们的研究对象是这123个内存溢出错误例子。这些例子既有用户手工撰写代码生成的应用，也有依赖高层语言（如Apache Pig）和高层库（如MLlib）生成的应用，应用的详细分布如下表

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **框架** | **应用来源** | **手写** | **Pig** | **Hive** | **Mahout** | **Cloud9** | **GraphX** | **MLlib** | **总数** | **重现数** |
| Hadoop | StackOverflow.com | 20 | 4 | 2 | 4 | 0 | 0 | 0 | 30 | 16 |
| Hadoop mailing list | 5 | 5 | 1 | 0 | 1 | 0 | 0 | 12 | 6 |
| Developers’ blogs | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 3 | 2 |
| MapReduce books | 8 | 3 | 0 | 0 | 0 | 0 | 0 | 11 | 2 |
| Spark | Spark mailing list | 16 | 0 | 0 | 0 | 0 | 1 | 2 | 19 | 3 |
| StackOverflow.com | 42 | 0 | 0 | 0 | 0 | 1 | 5 | 48 | 14 |
| 总数 |  | 93 | 13 | 3 | 4 | 1 | 2 | 7 | 123 | 43 |

### 错误原因与修复方法搜集

对于每一个错误例子，我们手工分析用户的错误描述信息以及专家的诊断信息。专家包含Hadoop/Spark的框架设计开发者，有经验的应用开发者或者MapReduce书籍作者，这些专家有来自Hadoop/Spark框架提供商，比如cloudera.com和databricks.com，也有来自框架用户比如ebay.com，huawei.com。我们通过下面的方法来确定错误原因：

1. 专家诊断出了错误原因，用户也接受了专家了回答，一共66个。
2. 用户自己诊断出了错误原因。用户详细原因，包括数据异常信息，不正确的配置，或者异常的用户代码逻辑。一共45个。
3. 我们自己重现了错误（重现方法见下节），手工分析诊断出了错误原因。一共12个。

修复方法类型的确定：我们发现这123错误中有42个错误有专家的修复建议或者用户修复所使用的方法。我们首先抽取相应的修复方法，然后将类似的修复归并在一起，最终得到11个修复方法类型（pattern）。

### 错误重现方法

错误重现方法：为了更清楚地研究内存溢出错误的原因，我们重现了43个内存溢出的错误（35%），这43个错误详细的可以重现的信息（比如数据类型，数据大小，可重现的用户代码和内存溢出错误的trace）。因为我们无法得到用户自己的数据集，我们使用公开数据（如英文维基百科）或者合成数据（一个公用的benchmark [BrownBench]。重现实验在11个节点上的集群进行，框架版本是Hadoop-1.2和Spark-1.2。每个节点包含16GB的内存，map和reduce任务的内存大小设为1GB。重现出来的错误栈信息和用户报告的一致。

### 研究方法的局限性

应用的代表性：我们只选取了在开源框架（也就是Apache Hadoop和Apache Spark）上运行的应用。尽管很多公司（比如Facebook和Yahoo！）都在使用这两个框架[PowerBy]，一些大公司 建立了自己的分布式数据并行框架（比如Microsoft的Dryad），我们还没有研究运行在这些私有框架上的应用。

错误类型的完整性：在276个错误中，我们只研究了123个，还有153个的错误原因是未知的。这153个错误中可能还存在新的错误原因或者修复方法。另外，当一个应用的内存溢出错误存在多个错误原因时，用户的错误描述和专家的诊断信息可能只覆盖了主要的错误原因。比如，专家可能只诊断出来错误原因是用户代码缺陷，但另外一个原因可能是框架在内存中暂存了大量的数据。

分类的主观性：尽管我们全力去理解、判断和归类错误原因，仍然有不准确的分类。比如，对于一个内存溢出错误，专家可能诊断出来多个错误，但我们只选取被用户接受的错误原因。

* 1. RQ1（常见错误原因）的研究结果

尽管这123个内存溢出的错误原因多种多样，我们可以根据其与数据存储、数据流和用户代码的关系将错误原因分成3大类（共10个小类），详见下表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **错误类别** | **错误原因** | **原因描述** | **Hadoop** | **Spark** | **总数** | **比例** |
| 框架暂存了大量数据 | 框架缓冲（buffer）了大量数据 | 大量的中间数据被存放到了框架的缓冲区里 | 6 | 2 | 8 | 6% |
| 框架缓存（cache）了大量数据 | 用户显式地将大量数据缓存到内存用（用于数据重用） | 0 | 7 | 7 | 6% |
| **子类别总数** |  | **6** | **9** | **15** | **12%** |
| 数据流异常 | 数据划分不恰当 | 一些数据块（partition）变的太大 | 3 | 13 | 16 | 13% |
| 热点key | 运行时一些<k, list(v)>变得太大 | 15 | 8 | 23 | 18% |
| 单个键值对太大 | 运行时单个<k, v>太大 | 6 | 1 | 7 | 6% |
| **子类别总数** |  | **24** | **22** | **46** | **37%** |
| 内存使用密集的用户代码 | 用户代码加载了大量外部数据 | 用户代码在未处理键值对之前，加载了大量的外部数据 | 8 | 0 | 8 | 6% |
| 单个中间计算结果过大 | 用户代码在处理单个键值对的时候产生了大量的中间计算结果 | 4(3)\* | 2 | 6(3) | 5% |
| 累积中间计算结果过大 | 用户代码在内存中累积了大量的中间计算结果 | 30[13]\* | 10[1] | 40[14] | 33% |
| Driver生成了大量的数据 | Driver程序本身产生了大量的数据 | 0 | 9 | 9 | 7% |
| Driver收集了的计算结果太大 | Task输出的计算结果太大，被driver全部收集 | 0 | 16 | 16 | 13% |
| **子类别总数** |  | **42** | **37** | **79** | **64%** |
| **总数** |  |  | **72** | **68** | **123+17** | **113%** |

\*注：4(3)表示4个内存溢出错误中有3个的错误原因不仅包含“单个中间计算结果太大”，同时也包含“单个key/value键值对太大”。30[13]表示30个内存溢出错误中有13个的错误原因不仅包含“中间计算结果累积量太大”，同时也包含“热点key”。113%的意思是30个内存溢出错误中有13个有两个错误原因类型。

上表说明最主要的错误原因（占64%）是用户代码消耗了过多的内存。第二大的错误原因（占37%）是数据流异常，13%的错误既是由于内存消耗量太大的用户代码也是由于数据流异常导致的。

### 错误类别1:框架暂存了大量的中间数据

这个类别包含两种错误原因：

A. 框架缓冲（buffer）了大量数据

为了降低磁盘的读写I/O，框架一般会在内存中先对中间数据（map()的输出和reduce()的输入数据）进行缓冲（buffer）。缓冲区有两种：固定缓冲区和虚拟缓冲区。固定缓冲区直接占内存的大块空间，比如Hadoop的map buffer是一个大的byte[]数组。虚拟缓冲区只是一个内存界限，用来限定内存中最多有多少空间可以用来存储中间数据。Hadoop和Spark都有一个虚拟缓冲区叫做shuffle buffer，用来暂存从map任务shuffle过来的的数据。

有8个内存溢出错误（6%）可以被划分到这个小类里面。4个错误是因为Hadoop的map buffer分配的过大（通过io.sort.mb）设置。比如，一个错误配置了300MB的map buffer，但给mapper分配的内存只有200MB [e01]。另外4个错误是由于shuffle buffer开的过大。比如，一个用户将shuffle buffer设为0.7（也就是shuffle buffer可以占用70%的reducer内存空间），这个值太大（设置为0.3可以成功执行），导致了内存溢出错误。

B. 框架缓存（cache）了大量重用数据

缓冲在buffer中的数据是由框架自动完成的，在框架中缓存（cache）的数据是有用户显式地制定的，而且是用于数据重用的。在一些应用，特别是迭代型的机器学习应用或者图计算应用（比如PageRank和Kmeans聚类）中，输入数据（比如输入的原始图或者训练数据）或者一些中间计算结果（比如权重参数）可以在不同的jobs中进行重用。Hadoop的重用方法是将当前job的输出数据写入分布式文件系统（HDFS），然后下一个job可以从HDFS中读取。Spark提供了内存缓存机制，用户可以显式地指定那些中间数据要缓存到内存中。如果在Spark中缓存的数据量太大，会导致内存溢出的错误。

这个子类别包含7个内存溢出错误（占6%）。这些错误都是由于Spark job在内存中缓存了大量的重用数据（RDD）。比如，一个应用在试图缓存最后job输出的大的RDD时产生了内存溢出错误[e03]。有3个错误的原因是job不断在内存中缓存数据。比如，一个机器学习应用SVDPlusPlus在每次迭代中均在内存中缓存图数据，最后因为数据量太大导致内存溢出错误[e04]。最后一个错误因为用户显式地缓存了driver程序广播过来的大的数据[e05]。

通过这两个错误子类别的实证分析，我们可以研究发现：

|  |
| --- |
| Finding 1：尽管用户可以通过参数设置来限制框架的内存使用量，但还是由12%的错误是由于框架暂存大量的中间数据导致的  Implication：用户很难设置正确的配置参数来调配框架和用户代码的内存占用比。 |

### 错误类别2:数据异常

因为在框架中数据是被分布处理的，在每个处理步骤中的数据量大小（比如key/value键值对大小和key分布）是运行时动态决定的，所以框架不能避免产生数据流异常，比如产生大的数据分块（large data partition），大的<k, list(v)>组，大的单一key/value键值对。数据流异常可以直接导致内存溢出错误。该类别包含三种错误类型。

A. 不合适的数据划分

为了实现“分治－聚合”的并行处理模式，框架将mapper的输出数据进行划分（partition），然后不同的数据分块（partition）会被汇集（shuffle）到不同的reducer进行处理。在Hadoop/Spark中，map()会根据输出的key/value records的partition id（比如id = hash(key)%partitionNumber）将records输出到不同的数据块。比如在下图中，和有相同的partition id。在相同partition中的records会被相同的用户代码（reduce()或combine()）处理。不合适的数据划分包含两种子类别：(1) 当partition number比较小的时候，所有的数据分块都可能很大。(2)不均衡的partition函数可以导致某些数据分块远远大于其他数据分块。常用的partition函数（比如hash partition和range partition）都不能避免产生不平衡的数据分块。不合适的数据划分可以导致存放到内存中的中间数据量过大。比如在下图中，*P1* 包含的key/value键值对个数远远大于*P2*和*P3*，那么在shuffle *P1*的时候reducer需要在内存缓冲更多的数据。更严重的是，如果把Spark的spill参数设置为false，那么Spark会将整个*P1*都缓冲到内存中，可以直接导致内存溢出错误。不合适的数据划分同样可以导致接下来的用户代码的输入数据量过大，由于用户代码的内存消耗量通常与处理的数据量有关，那么输入数据量过大可以导致用户代码产生内存溢出错误。比如在下图中，如果*P1* 包含的key/value键值对个数远远大于*P2*和*P3*，那么reduce()可能在处理*P1*时产生内存溢出错误，而在处理*P2*和*P3*的时候不产生内存溢出错误。

这个子类比包含16个内存溢出错误。其中7个错误是因为partition number过小。比如，将partition number增加到1000后，用户报告说每个数据分块变的很小，也没有内存溢出错误了[e06]。有4个错误是因为partition函数不平衡导致的。比如，一个用户报告说大多数中间数据都聚集到了2个reducer那里，其中的1个reducer的输入数据超过了总数据量的一半[e07]。在剩下的5个错误中，用户只报告说数据分块变的很大，但没有详细描述partition number和partition函数的情况。



B. 热点key

一个数据分块是一个粗粒度的key/value records集合，集合里面的records仍然可以有不同的key。尽管在一个partition里面的records要被同样的用户代码处理，这些records要先根据key被聚集成不同的<k, list(v)>组。然后用户代码（reduce()或者combine()）处理组里面的每一个record。热点key的意思是某些<k, list(v)>组包含的<k, v> records远远多于其他组。Partition个数可以影响每个partition的大小，但不能影响每个<k, list(v)>的大小，因为组的大小是运行时动态决定的（也就是多少个records具有相同的key）。比如在上图中，如果对应的组包含的records个数远远大于对应的，那么框架在聚合时可能会出现内存溢出错误，或者处理的用户代码也会产生内存溢出错误。

这个子类别包含23个内存溢出错误（占18%），这些错误均是由于某个key对应的的values个数太多。比如，一个用户报告说错误原因是一些key只有1个或2个values，但其他一些key对应10万个values [e08]。另外一个用户报告说某个key <custid, domain, level, device>严重倾斜，大约42%的records都包含这个key [e09]。另外6个错误是由于Spark的groupByKey(), reduceByKey() 和coGroup()等聚合类错误出现了热点key。比如，一个用户报告说某个key对应太多的values，这些values不能被存放到内存中[e10]。专家解释说当前的groupByKey()要求一个key对应的所有values都能够存放到内存中[e11]。

C. 单个key/value record过大

因为用户代码需要将整个record读入内存进行处理，所以如果一个key/value record过大，那么用户代码会直接出现内存溢出错误，或者在处理这个record的时候产生大量的中间计算结果，导致内存溢出错误。因为record的大小也是动态决定的，所以数据流相关的配置也不能控制record的大小。

这个子类别包含7个内存溢出错误，都是因为单个record过大。比如，一个用户报告说内存只有200MB的空间，但应用产生了一个350MB的record（这个record只包含一行字符都是a的文本）[e12]。另外一个用户报告说一些record只有1MB，但另外一些有100MB，而且不能分割[e13]。更惊奇的是，一个用户报告说因为数据格式变化而分割符没有变化，全部的100GB数据都进入了同一个record [e14]。

|  |
| --- |
| Finding 2: 数据流异常很常见（占37%的错误），异常数据流可以导致运行时数据量过大，比如 large data partition，hotspot key 和 large single key/value record  Implication: 框架在进行数据划分时没有考虑运行时数据的特征（比如key 分布），导致中间数据量和用户代码的输入数据量不可控 |

### 错误类别3:内存使用密集的用户代码

不同与传统的单机程序，分布式数据并行应用的用户代码有一个重要特征就是streaming-style。Key/value records被用户代码一个个按顺序读取，处理，最后输出。所以，当一个record被处理完以后，该record及其对应的中间处理结果会被清空。除非这个record或其对应的中国结果会被以后用到，而被用户显式地存储。基于这个特征，我们总结出两种错误子类型：单个中间计算结果太大，累积中间计算结果过大。另外一个子类型是用户代码加载了大量的外部数据。另外，driver程序也能触发内存溢出错误，相应的错误子类型一是driver程序本身产生了大量的数据，而是driver收集了过大的task的处理结果。

此处要有图（带buffer的）

A. 用户代码加载了大量的外部数据

不同于被用户暂存（buffer/cache）的数据，外部数据指的是被用户代码直接加载的数据（不包含input records）。一些应用在处理input records之前，需要从本地文件系统、分布式文件系统、数据库或者其他地方加载一些数据用来辅助处理records。比如，为了查询input records的key是否在一部字典里，用户代码会加载一部字典到内存中（比如HashMap里面）。如果加载的外部数据太大，那么会直接导致内存溢出错误。

这个子类别包含8个内存溢出错误（占6%）。有3个错误出现在Mahout应用中，这些应用中的mapper从分布式文件系统上加载了大量的训练模型数据用来做分类[e15]和聚类[e16]。另外一个错误发生在Hive应用里，这个应用加载了一个大的外部表[e17]。还有一个错误出现在Pig脚本的UDF（用户定义函数）里面，这个UDF加载了一个很大的外部文件[e18]。

B. 单个中间计算结果过大

中间计算结果指的是用户代码在处理单个<k, v> record过程中产生的计算结果。这个子类型包含两种情况：(1) 输入的单个<k, v> record很大，那么产生的中间结果也很大。比如，如果一个record是一个64MB的句子，那么这个句子被分词后，也会占用64MB左右的空间。(2) 一个很小的input record仍然可以产生较大的中间处理结果。比如，一个record包含两个集合，那么这两个集合的笛卡尔集要比这个record大的多。

这个子类别包含6个内存溢出错误（占5%）。其中3个错误的原因是input record过大。另外一个错误是由于reduce()生成了一个非常长的record [e19]。在另外一个错误中，一个机器学习应用在内存中建立了一个大的训练模型用于矩阵分解[e20]。最后一个错误发生在文本处理应用[e21] 中，这个文本处理程序使用了第三方的库StanfordLemmatizer来处理文本。在处理文本中的每一行（line）的时候，lematize()会分配一个大的数据结构用于动态规划，库的作者解释说这个数据结构可以扩张到是输入line的3倍。当line过大时，lematize()会产生内存溢出错误。

|  |
| --- |
| **public class Mapper** {  StanfordLemmatizer slem = **new** StanfordLemmatizer();  **public void** map(Long key, Text value) {  String line = value.toString();  **for**(String word: slem.lemmatize(line))  emit(word, 1);  }  } |

C. 累积的中间计算结果过大

如果当前record对应的中间计算结果被显式地缓存到内存中，那么这些计算结果就会成为累积计算结果。这样，records被处理的越多，内存中就会累积越多的中间计算结果。比如，为了对input records进行去重，map()会分配一个Set用来持有不同的records。如果不同的records个数很多，那么Set也会变得很大。、reduce()在处理一个大的<k, list(v)> 的时候也会产生大量的累积中间结果，这个大的累积中间结果可能时由于热点key导致的。

这个子类别包含40个内存溢出错误（占33%）。在11个错误中，用户分配了内存数据结构用来持有input recods。比如，一个用户使用ArrayList来持有一个key对应的所有的values，而value的个数可能达到1000万个[e22]。在其他的错误中，用户试图去累积中间结果，比如word的次数[e23]和训练参数[e24]。用户代码也会累积中间计算结果，比如去计算不重复的元组[e25]，进行基于内存的排序，计算中值，或者做笛卡尔集[e26]。下面的错误发生在 reduce()，reduce()分配了一个类似HashMap的数据结构（HMapStIW）用来持有每个词（word）的邻接词。因为文档中的word过多，而且每个word有很多邻接词，具有累积逻辑的plus()将太多的词存放到了map数据结构中，导致了内存溢出错误。

|  |
| --- |
| **public class Reducer** {  **void** reduce(Text key, Iterable<HMapStIW> values) {  Iterator<HMapStIW> iter = values.iterator();  HMapStIW map = **new** HMapStIW();  **while** (iter.hasNext()) {  map.plus(iter.next());  }  emit(key, map);  }  } |

D. Driver产生或收集了大量的数据

尽管driver程序不直接处理<k, v> records，driver程序可以出现内存溢出错误，而且有两个子类别：(1) driver程序本身在内存中产生了大量数据。(2) driver收集了大量的task处理结果。

第一个子类别包含9个内存溢出错误（占7%）。Driver产生了大量数据，这些数据是为了广播到其他节点用于本地计算。用于数据广播的情况：比如一个driver程序产生了1GB的的数组[e28]，另外一个产生了0.15GB的变量[e29]。用于本地计算的情况：一个driver程序产生了8000×8000的矩阵，大约256MB[e30]。另外一个driver为了计算条件概率，产生了4亿个double变量。

第二个子类别包含16个内存溢出错误（占13%）。比如，一个driver程序使用graph.edges.collect()把4.5GB的图的所有边都收集并放到内存[e32]。另外一个driver将reduceByKey()的计算结果（包含2亿个word）收集起来放到了内存[e33]。更有意思的是，在一个迭代型应用里面，driver把每一轮迭代task产生的结果都收集起来放到内存，这样结果在内存慢慢积累，最后导致内存溢出。

|  |
| --- |
| Finding 3：内存使用密集的用户代码是最主要的错误原因（占64%的错误），复杂度高是由于用户代码不小心处理了大量数据或者本身的空间复杂度很高  Implication：用户很难在不知道运行时输入数据量的情况下设计出memory-efficient 的代码 |

* 1. RQ2（常见修复方法）的研究结果

我们可以看到内存溢出错误与数据存储、运行时数据流和用户代码都有关系，所以修复内存溢出错误不是一件容易的事。用户一般采取的方法就只是去提高内存界限，但是这种方法并不可靠，因为扩张后的内存空间仍然可以迅速填满。

为了找到可靠的修复方法，我们研究了42个包含修复信息的内存溢出错误，从修复信息（来自专家修复建议或者用户自己的修复方法）中总结出11中修复方法。其中8种是比较传统的方法，3种是有些tricky的方法，比如去重新设计key，跳过异常大的数据，更改应用相关的参数。下表总结了每种错误类型对应的修复方法（3种错误类型缺少修复方法）。标签“配置”表示该修复方法需要修改参数，标签“代码”表示该修复方法需要更改用户代码。Error(n)表示该修复方法包含相应的错误个数和被修复的错误个数（n）。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **修复方法类别** | **错误原因** | **修复方法类型** | **配置** | **代码** | **Error(n)** |
| 数据存储相关的修复方法 | 框架缓冲（buffer）了大量数据 | 把框架的缓冲区调小 | ✓ |  | 6 (6) |
| 框架缓存（cache）了大量数据 | 降低缓存界限，或者使用基于磁盘的缓存 | ✓ |  | 2 (2) |
| 数据流相关的修复方法 | 数据划分不恰当 | 增加partition个数，或者改变partition函数 | ✓ |  | 12 (6) |
| 热点key | 重新设计key（比如使用组合key） |  | ✓ | 3 (0) |
| 单个键值对太大 | 把大的record拆分成小的records |  | ✓ | 4 (1) |
| 用户代码相关的修复方法 | 累积中间计算结果过大 | 将累积操作变成流式操作 |  | ✓ | 2 (2) |
| 把累积操作拆分成多个轻量级操作 |  | ✓ | 3 (1) |
| 把部分中国计算结果spill到磁盘上 |  | ✓ | 3 (1) |
| 跳过异常大的数据 |  | ✓ | 2 (2) |
| Driver收集了的计算结果太大 | 使用tree聚集来替代直接收集 |  | ✓ | 3 (2) |
| 调整应用本身的参数 | ✓ |  | 2 (2) |
| **总数** |  |  |  |  | **42 (25)** |

### 数据存储相关的修复方法

该类别有两种修复方法可以减少框架暂存的数据。这两种修复方法都只需更改内存相关的配置参数。

A. 把框架缓冲区调小

降低缓冲区大小的意思是减小map buffer或者减小shuffle buffer。两者都可以直接减小框架暂存的数据量，但代价是会有更多的磁盘I/O。

这个修复方法修复了6个内存溢出错误。其中3个是通过减小Hadoop的map buffer（比如把io.sort.mb从128MB调小到64MB[e35]）修复的。有2个是通过降低Hadoop的shuffle buffer修复的。比如，一个错误修复方法是把shuffle.input.buffer.percent从70%调小到30%，但会导致更长（2个小时）的执行时间[e02]。最后一个错误的修复方法很极端，是把Spark的shuffle buffer设置为0，这样会导致执行时间变长了20%[e36]。

B. 降低框架的缓存界限

缓存界限是一个配置参数，用来限制可以用于数据缓存的内存空间大小。降低缓存界限可以直接减少缓存的数据量，代价就是大的数据块不能被缓存。为了解决这个问题，Spark提供了多种内存＋磁盘的cache机制，用户可以选择多种存储模式，memory-only， memory+disk，或者disk-only。

这个修复方法修复了2个内存溢出错误。其中一个是将缓存界限（spark.storage.memoryFraction）从0.66调小到0.1[e37]。另外一个是将存储模式从memory-only设为disk-only，代价就是执行速度会变慢。

### 数据流相关的修复方法

由于数据partition过大，热点key，单个key/value键值对过大都可以导致内存溢出错误，相应的修复方法就是要对运行时的数据进行进一步划分，具有包含3种修复方法。

A. 改变partition个数或partition函数

数据块（partition）的大小与partition个数和partition函数直接相关。一种直接减小partition的大小的方法就是增加partition个数，尽管不是100%可靠的方法。另外一种修复方法是尝试另一种partition函数来避免不平衡的partition，比如使用range partition，round-robin partition，或者自定义的partition函数。

有12个内存溢出错误用到了这种修复方法。其中6个错误可以通过增加partition个数来修复。在Hadoop种，partition个数的配置参数就是reduce number。在Spark种，partition个数的配置参数是spark.default.parallelism。比如，当把partition个数提高到1000后，用户报告说每个partition的数据量减少很多，内存溢出错误也消失了[e06]。一个错误被部分修复，因为用户报告说“我唯一的修复方法是去增加partition number，但这好像不是一种绝对可靠的方法”。专家解释说当key分布均匀时，增加partition个数才是一种可靠的修复方法[e39]。在另一个错误中，用户尝试对一个range partition函数调整range来避免不平衡的划分。在剩余的4个错误中，用户只是被建议增加partition个数。

B. 重新设计key

重新设计key的目的是当出现热点key的时候让key分布更加平均。一种简单的方法是使用组合key来替代单一key。比如map()可以输出一个key组(*k*1, *k*2)来替代单一key *k*，如下：

*map*(*k*, *v*) ⇒ *list*((*k1*, *k2*), *v*)

*reduce*((*k1*, *k2*), *list*(*v*)) ⇒ *list*(*v*)

这样，原来很大的<*k*, *list*(*v*)>组就被划分为多个小的<(*k*1, *k*2), *list*(*v*))> 组。

有3个内存溢出错误用到了这个修复方法。在一个错误中，key *vehicle*对应太多的values。专家建议用户使用(*vehicle*, *day*)作为key，这样可以减少一个key对应的values数据[e40]。在另外一个错误中，专家建议用户在原始key后面加上一个哈希值（可以是1-10）来划分大的<*k*, *list*(*v*)>组[e11]。在最后一个错误中，用户建议在key后面加上1或2[e26]。

C. 对单个键值对进行划分

这个修复方法是要将大的单一record划分为多个小的records。

有4个内存溢出错误使用了这个修复方法。在一个错误修复中，专家建议用户在map()中将100MB+的record划分为多个小的records，这样接下来的reduce()处理的record会小[e41]。在一个错误修复中，专家建议用户不要输出一个巨大的字符串，而要将字符串划分为多个小的字符串输出[e42]。在一个错误修复中，专家建议用户使用pos.maxlen来将过长的句子划分为多个短的句子[e21]。在最后一个错误修复中，修复方法是将大的record（倒排索引的一个链表）划分为小的record，然后对每一个key输出多个records。

### 用户代码相关的修复方法

下面列举的修复方法只针对用户代码中的map()和reduce()。

A. 将累积操作转换为流式操作

因为累积操作是造成内存溢出的重要原因，所以专家建议用户将累积操作转换为流式操作。

有两个内存溢出错误用到了这个修复方法，这两个错误都是发生在基于内存排序的时候。为了对<*k*, *list*(*v*)>中的*list*(*v*)进行排序，用户在内存分配一个List来持有每一个*v*，这个累积操作导致了内存溢出错误。专家建议用户利用框架的排序机制来将累积操作转换为流式操作。具体方法是在key中添加v，原来的k变成(k,v)。这样当框架在排序key的时候，就顺带把v给排了序。最后结果就是，用户代码不用分配任何数据结构，只需进行流式操作（即将读入的list(v)中的v一个个输出）即可[e44]。

B. 把累积操作拆分成多个轻量级操作

这个修复方法是将累积操作拆分成多个轻量级操作。轻量级操作是指该操作只处理部分数据或者具有较小的内存消耗。

有3个错误用到这个修复方法。在一个错误中，专家建议用户在map()之后和reduce()之前加上combine()（也就是一个迷你reduce()）来提前进行部分聚合。这样，reduce()就会收到较少的数据（更好的不重复的values）[e45]。在另一个错误中，专家建议用户使用使用aggregateByKey()来替代groupByKey()，因为前者可以进行本地聚合，这样可以使得shuffle和reduce()的输入数据量变小[e46]。最后一个错误[e47]的修复方法是将一个聚合操作拆分为两个轻量级操作。输入数据是一张包含多列的大表，应用想要统计column A中每个不同的元素在column B中对应的不同元素数。 用户先进行groupby(A)操作，然后进行count(distinct B) 的聚合操作。由于某些A包含过多不同的B元素，导致count(distinct B)操作出现内存溢出错误。专家建议用户将累积操作拆分为两个轻量级操作如下：

原始版本：groupby(A) ⇒ count(distinct B)

优化版本：groupby(A, B) ⇒ output(A, B) ⇒ (1)

groupby(A) ⇒ output(B) (2)

第一步先将A和B联合起来进行聚合，也就是使用groupby(A, B)，得到的结果是不同的(A, B)。第二步进行groupby(A)，然后直接输出groupby的结果（也就是不同的B）即可。

C. 将部分聚集的中间结果spill到磁盘上

这个修复方法不是去优化用户代码，而是在累积的中间过变得过大之前将这些中间结果spill到磁盘上。

有3个内存溢出错误用到了这个修复方法。在其中一个错误中，map()每处理n个records，就把中间计算结果存放到磁盘上[e23]。在一个Hive错误中，专家建议用户将hive.map.aggr.hash.percentmemory从0.5调小到0.25，这意味着map()分配的HashMap占到内存总量25%的时候会将存放的数据spill到磁盘上[e48]。同样，在一个Pig错误中，专家建议用户将pig.cachedbag.memusage设为0.1或者更低，当Pig在内存中存放的内部数据（bag）达到这个界限时会spill到磁盘[e25]。

D. 跳过异常大的数据

这个修复方法的意思是当遇到特别大的record或者<k, list(v)>组时，直接跳过这个数据而不处理。我们很惊奇地看到还有这样tricky的修复方法，原因可能是有时难以去优化用户代码，尤其是当用户代码使用了第三方包的时候。这个修复方法同样适用于那些某些不需要精确结果的应用，比如对大规模数据进行简单统计的应用。

有2个内存溢出错误用到了这个修复方法。第一个错误发生在一个视频推荐应用里面。出错地方是这个应用的用户代码处理了异常数据（有一个用户包含了10万个观影信息）。修复方法是跳过这些异常用户的数据，因为这些用户可能是机器人，而且这个应用最多只需要一个用户的100部观影信息来进行特征建模[e49]。在另一个错误中，热点key对应的values被直接跳过，因为它们对最终结果并没有产生影响[e38]。

### Driver程序相关的修复方法

针对driver收集大量task计算结果而导致内存溢出的情况，有两种相应的修复方法。

A. 使用树形聚合（Tree aggregation）

这个修复方法以多层树的方式来收集和聚合从task那里获得的输出结果。在第一层，这种方法对task的输出数据进行两两聚合，这样相当只剩一般的task的输出结果。接着，在第二层到第n层，仍然执行相同的两两聚合操作，最后的两两聚合由driver程序来完成，但此时存在于driver中的数据将会很少。

有2个MLlib应用出现的内存溢出错误用到了这个修复方法。用户使用树形聚合方法来避免直接去收集过多的task结果（比如向量[e50]和梯度[e51]）。

B. 调整应用本身的参数

为了修复内存溢出错误，专家和用户甚至尝试去调整应用本身的参数。在一些机器学习应用，比如SVD和ALS中，task的输出数据大小与应用本身的参数（比如SVD中的k和ALS中的rank）存在线性或多项式相关性。降低这些参数就一位置task的输出数据大小会变小。

有2个MLlib应用出现的内存溢出错误用到了这个修复方法。第一个是去调小SVD的参数k，第二个是把ALS的rank调小到40以下。

|  |
| --- |
| Finding 4：常见的修复方法包含：调整内存或者数据流相关的配置，对运行时数据进行动态划分，优化用户代码逻辑。我们也很惊奇地发现一些tricky的修复方法，比如使用组合key，跳过异常大的records等。  Implication：用户很难在不知道运行时输入数据量的情况下设计出memory-efficient 的代码。 |

* 1. RQ3（框架改进方法）的研究结果

因为内存溢出错误不能被现有的错误容忍机制解决，设计新的错误容忍机制和错误诊断方法很重要。

从我们总结出的错误原因和我们重现的错误诊断经验中发现两种可以提高框架错误容忍能力和一种辅助错误诊断的方法。

### 辅助错误诊断的方法

A. 提供详细的数据流统计信息

数据流异常是内存溢出错误的常见原因，但是当前的框架只提供了很少的数据流信息。比如，Hadoop只统计当前已经处理过的records个数，而Spark还需要用户自己写程序去统计当前已经处理的records个数。

如果框架能够提供详细的数据流统计信息，那么内存溢出错误诊断将会变得简单。数据统计信息包括已经处理过的records统计信息，比如最大／最小／平均record大小，每个<k, list(v)>组中包含的records最大／最小／平均个数，每个数据块中包含的records最大／最小／平均个数。我们在Hadoop 中实现了一个详细统计profiler用来统计数据流，profiler生成的数据流统计信息帮助我们诊断出7个数据流异常错误原因，包括2个不合适的数据划分，4个热点key，和一个单一key/value record过大。

### 提高框架的错误容忍能力方法

A. 设计动态内存管理方法

分析错误原因后可以发现，有12%的内存溢出错误的原因是框架暂存了大量的数据。因此，我们可以得知用户很难设置正确的配置参数来调配框架和用户代码的内存占用比。

动态内存管理目的是去动态调配框架和用户代码的内存占用比。直接实时计算用户代码的内存消耗量是困难的，但我们可以实时监控总的内存消耗量（total）和框架暂存的数据量（du）。可以将两者之差（total-du）简单地当作用户代码的消耗量。当内存总消耗量total达到一个界限时，框架可以暂停执行用户代码，将暂存的数据spill到磁盘，然后恢复执行用户代码。同样，当用户代码的内存消耗降低到一个界限时，可以将spill到磁盘的缓存数据读回到内存。

B. 提供内存磁盘混合数据结构

用户代码经常使用内存数据结构来对key/value records或者中间计算结果进行聚合。但这种操作很容易导致内存溢出错误，比如我们发现有18个内存溢出错误（15%）在内存数据结构中出现，这些数据结构包括List/ArrayList (9个错误)，Map/HashMap (5个错误)，Set/HashSet (3个错误)和PriorityQueue (1个错误)。

一个理想解决方案是为用户提供常用的内存磁盘混合数据结构。新的数据结构应该为用户提供与原始数据结构类似的API（比如Java的Collections和C++的STL）。与通用程序语言中的数据结构不同的是，新的数据结构应该能够根据当前的内存用量自动在内存和磁盘上进行swap。

Spark已经实现一个类似HashMap的内存磁盘数据结构叫做ExternalAppendOnlyMap。AppendOnly的意思是可以向这个数据结构添加或更改key，但不可以删除key。然而这个数据结构不像通用语言里面的数据结构可以支持增删改查，而且不能被用户代码使用（当前正在框架本身提供的操作，比如reduceByKey()使用）。其他的开源项目，比如STXXL [STXXL]和TPIE [TPIE]已经实现了全磁盘的数据结构和相应的算法，适用于数据量太大无法放入内存中的情况。如果能将这些数据结构进行改写，支持内存和磁盘混合存储，而且针对key/value records聚合进行优化，那么可以很好地降低内存溢出错误发生的概率。

|  |
| --- |
| Finding 5：当前框架为内存溢出错误的诊断和错误容忍提供了非常有限的支持  Implication：框架可以提供多种机制来辅助错误诊断和错误容忍：提供详细的数据流统计信息，提供动态内存管理策略，提供方便进行key aggregation 且可以在内存和磁盘进行 swap 的数据结构 |

* 1. 讨论

A. 研究结果的通用性

我们的研究对象是Apache Hadoop和Apache Spark的应用。大部分研究结果可以适用于其他类似MapReduce编程范型的框架，比如微软的Dryad [Dryad]和Naiad [Naiad]，Yahoo!早期的Map-Reduce-Merge [MRM]，和最近正在开发中Apache Flink。这些框架与MapReduce有类似的编程模型和分布式数据流。

B. 应用的覆盖情况

当前我们主要研究的是通用处理框架的MapReduce应用。尽管已经覆盖了文本处理，SQL处理，机器学习，图处理等应用，还有很多为专门应用设计的框架。比如Pregel [Pregel]和Powergraph [Powergraph]是专门用来进行大规模图处理和分析的框架，Apache Storm [Storm]是专门用于流处理的框架。我们把研究这些非MapReduce框架应用中的内存溢出错误作为我们未来的研究工作。

当前我们只研究了公开的内存溢出错误，但实际上我们也研究了工业界（包括阿里巴巴淘宝和腾讯）里面的Hadoop/Spark应用发生的内存溢出错误。只是考虑到商业机密原因，我们无法将应用本身及其错误原因进行详细分析。然而，我们发现工业届的内存溢出错误（比如[TencentOOM]）与我们研究的公开错误（比如[e47]）有相同的错误原因和修复方法。

* 1. 本章小结

本章详细研究分析了真实的数据并行应用的内存溢出的错误原因和修复方法。我们研究的对象是123个真实的Hadoop和Spark的应用。我们发现内存溢出的三类错误原因是内存使用密集的用户代码，数据流异常和框架暂存的数据量过大。我们也从42个包含修复信息的错误中总结出了常用的修复方法。另外，我们对内存溢出错误的研究分析经验帮助我们提供了潜在的可以提升框架错误容忍能力和辅助错误诊断的三种方法，我们认为研究结果不仅可以帮助用户和框架设计者更好地处理内存溢出问题，而且帮助以后的研究者发现新的问题。

# 应用内存用量模型建立及用量估算方法

摘要

* 1. 问题

给定应用 job (data, configurations, user code)，如何建立 MemoryUsage(task) = f (data, configurations, user code)？

作为一个简单通用的分布式数据并行框架，MapReduce和它的开源实现Hadoop目前广泛用于大规模的数据处理分析。MapReduce框架的最大优势是没有分布式或者并行系统经验的用户也能开发可扩展的数据密集型应用。然而，缺点是在提交应用（job）到框架之前，用户需要自己去配置资源需求量，特别是内存使用量需求。

然而问题是用户不知道要配置多大的内存需求，因为内存用量是动态的，而且用户通常对框架的分布式执行原理和流程并不清楚。因此，用户不知道怎么去分析内存用量，更不知道如何去估算应用的内存使用量，进而不知道如何去配置内存需求大小。

基于以上原因，用户一般根据他们的“想像”去配置内存大小，但这会带来两方面的问题：(1) 如果内存配置过大，会导致内存资源浪费，进而降低集群的并行度（同时能够运行的任务数）。比如任务资源管理与调度框架YARN和Mesos都需要用户事先指定内存需求，这样YARN和Mesos才能为任务寻找具有相应资源的节点，将任务调度到该节点运行。(2) 如果内存设置的过小，可能会导致内存溢出错误。

* 1. 问题描述

观察得出：

内存用量与数据流（中间数据量）紧密相关

内存对象有两种

① 框架暂存的中间数据 （与运行时中间数据量有关）

② 用户代码产生的中间计算结果 （与运行时输入数据和代码空间复杂度 有关）

应用消耗的内存空间由两部分构成：一是框架可以暂存一些中间数据到内存中，二是用户代码在内存中处理数据时也会产生中间处理结果。如图（此处需要内存使用图）所示，每一个map/reduce任务是一个独立进程（在Hadoop中，每个map/reduce task启动一个JVM运行）。Map()/reduce()先把数据输出到缓冲区，然后当缓冲区满的时候将数据spill到磁盘。用户对象（user objects）指的是用户代码产生的中间计算结果，这些结果的大小随着时间变化。在应用（job）运行之前，我们不知道多少数据要被缓存到内存中，也不知道用户代码会产生多少中间计算结果。

在背景里面要讲应用的内存使用量其实指的是它的任务（task）的内存用量。

挑战：

估计内存的使用量是一件有挑战性的事情，原因在于：(1) 内存用量受多个静态因素的影响，而且不是直接影响。比如input split size影响每个map task的输入数据大小，partition function定义map()如何对输出的<k, v> records进行划分。Partition number定义会有多少个数据分块（partition）产生及会有多少个reducer执行。Buffer size定义框架可以暂存的中间数据量。(2) 用户代码可以被任意书写，也可以由高层库或高层语言（比如类SQL的Pig和Hive脚本自动产生）。

当前MapReduce性能方面的研究工作集中于MapReduce应用的执行时间估计。然而，执行时间的估计方面很难被用于内存用量的估计。对于执行时间估计来说，基本假设是用户代码的执行时间与处理的数据量有关，而对于内存用量来说内存用量与处理的数据量不一定存在线性关系。

为了解决内存用量估计问题，我们主要解决三个子问题：

内存用量建模，用户代码空间度分析和模型参数估计。我们观察发现静态配置参数影响分布式数据流，而分布式数据流加上用户代码影响内存使用量。基于此我们提出了一种以数据流为中心的内存用量模型，该模型可以量化静态配置－数据流－内存用量的映射关系。该模型最主要的挑战点是在用户代码未知的情况下，如何去量化用户代码内存用量和输入数据量之间的关系。在分析了用户代码模版后，我们发现用户对象有固定的生命周期，而不同生命周期的用户对象受不同部分的输入数据影响。据此，我们提出了一种分段累积函数来量化用户对象大小和输入数据量之间的关系。因为我们的模型里面有一些参数（数据流参数和用户代码空间复杂度参数），我们通过在小数据集上模拟实验来得到参数，然后利用统计方法（比如线性回归）来得到用户代码的空间复杂度。实际上是一种白盒方法。我们的假设是在小数据集上运行时的中间数据量与在大数据集上运行产生的中间数据量之间存在线性关系。

给定一个MapReduce应用，给定数据，配置，和用户代码。我们的方法可以估计应用在大数据集上运行时的内存用量。评价显示我们的估计方法的平均误差率在20%以内。

我们的主要工作贡献是：

1. 我们分析了MapReduce应用代码的空间复杂度。
2. 我们提出了一种以数据流为中心构建应用内存用量模型的方法，该方法可以量化配置－数据流－内存用量之间的映射关系。
3. 我们基于抽样数据上的信息来估计大数据上的内存用量。

真实数据流可以抽样模拟

Job 处理大数据时（比如10GB）的数据流（中间数据量）与处理抽样数据（比如1GB）的数据流存在线性比例关系

* 1. 问题定义

给定应用 job (data, configurations, user code)，如何建立 MemoryUsage(task) = f (data, configurations, user code)？

Task包含map task和reduce task，需要分别估计map task和reduce task的内存用量（U），因为map和reduce task有多个，我们要估计mapper和reducer的最大内存用量也就是*Um*和*Ur*

m是mapper的个数，r是reducer的个数（也就是reducer number）。

* 1. 模型建立思路

以数据流为中心建立 静态因素－数据流－内存用量 定量关系

白盒方法（模拟＋统计）

子问题

① 怎么描述 静态因素－数据流 的关系？ ② 怎么描述 数据流－内存用量 的关系？

* 1. 数据流模型

数据流模型

* 1. 内存用量建模

应用内存消耗包含两部分：(1) 框架暂存的中间数据（buffered data）。(2) 用户代码产生的中间结果（会变成用户代码对象，user objects）。在不同的执行阶段task产生的对象不同，具有生命周期。

### 内存对象

根据内存对象类型，可以将对象分为三类：框架缓冲区（framework buffer），缓存的中间数据（in-memory intermediate data）和用户对象（user objects）。

### 框架对象模型

A. 框架缓冲区（framework buffer）

**对象定义：**固定buffer和虚拟buffer。固定buffer直接占据固定大小的内存空间（如Hadoop中的map buffer是一个byte[]数组）。虚拟buffer只是定义框架最多可以暂存多少数据到内存中（比如Hadoop中的shuffle buffer和reduce buffer）。

**在Hadoop中的情况：**在Hadoop中，每个mapper分配一个固定buffer（称为map buffer）来缓存map()输出的key/value records，等到buffer快满的时候，mapper会将缓存数据排序，然后将排序后的数据spill到磁盘上的不同的数据分块（partition ）中去。每个reducer 在shuffle阶段会分配一个虚拟buffer（shuffle buffer）来缓存从mapper那里shuffle过来的数据，当虚拟buffer被填满的时候，buffer中的数据会被排序、聚合、输出到磁盘上。用户也可以在reduce阶段分配一个虚拟buffer（称为reduce buffer）来暂存shuffle过来但没有被spill到磁盘的中间数据。

**生命周期：**缓冲区并不是一直都存在。以Hadoop为例，固定缓冲区map buffer只在map&spill阶段存在，在merge阶段就被回收了。虚拟缓冲区shuffle buffer在shuffle阶段存在，reduce buffer在reduce阶段存在。



**大小影响因素：**缓冲区大小由用户直接定义，比如使用io.sort.mb可以定义map buffer为300MB，使用 定义shuffle buffer是70%的内存空间，具体会消耗多少内存还要看缓冲了多少中间数据（见下一个item）。

**模型：**U(buffer) = *f* (configurations)

|  |  |  |  |
| --- | --- | --- | --- |
| **阶段** | **Buffer名称** | **类型** | **U(buffer) = f (configurations)** |
| Map | Map buffer | 固定buffer | io.sort.mb |
| Shuffle | Shuffle buffer | 虚拟buffer | HeapSize \* mapred.job.shuffle.input.buffer.percent |
| Reduce | Reduce buffer | 虚拟buffer | HeapSize \* mapred.job.reduce.input.buffer.percent |

B. 存放到内存的中间数据（in-memory intermediate data）

**对象定义：**指的就是暂存在虚拟buffer中的数据。

**在Hadoop中的情况：**在shuffle阶段，当虚拟buffer被填满的时候，buffer中的数据会被排序、聚合、输出到磁盘上。如果配置了reduce buffer，那么未被merge到磁盘的中间数据会被存放到reduce buffer中。

**生命周期：**尽管存放在buffer中的数据不定时会被spill到磁盘，但根据阶段来看，存放到内存中的中间数据与虚拟buffer的生命周期相同。

**大小影响因素：**暂存在虚拟buffer的数据与用户配置的虚拟buffer大小和shuffle得到的数据量有关。存放在shuffle buffer中的数据量直接有shuffle buffer size和shuffled data的数据量决定，是两者中的较小者。经过shuffle阶段后，没有被merge到磁盘上的数据量为 Size(shuffled data) % shuffle buffer size，而存放到reduce buffer中的数据量还收到reduce buffer限制，因此最后在reduce buffer的数据量大小是两者的较小值。

**模型：**U(in-memory intermediate data) = *f* (configurations, intermediate data)

|  |  |  |
| --- | --- | --- |
| **阶段** | **内存中的中间数据** | **U(in-memory data) = f (confs, intermediate data)** |
| Shuffle | Data in shuffle buffer | min (shuffle buffer size, shuffled data) |
| Reduce | Data in reduce buffer | min (reduce buffer size, shuffled data % shuffle buffer size) |

### 用户对象模型

由于用户代码可以被任意书写也可以被高层语言或高层库自动产生（自动产生map() 和reduce()），用户代码对我们来说相当于一个黑盒，很难依靠静态分析方法去确定的空间复杂度。

然而，我们观察发现MapReduce的用户代码有一个重要特点，可以帮助我们使用动态模拟＋统计的方法去确定用户代码的空间复杂度。这个特点就是streaming-style，也就是用户代码一个个读入、处理、输出records。这个特点可以使得在不同代码段分配／生成的用户对象有不同的生命周期，而不同的生命周期的对象大小有不同部分的input records决定。

根据这个特点，我们设计了分段累积函数来描述用户代码的空间复杂度。

**用户对象的生命周期：**

MapReduce编程范型的最大特点是用户以streaming-style来处理输入的key/value records。这样，当下一个输入record进入用户代码时，当前record及其相关的中间处理结果会在内存中被清除，除非他们被用户故意缓存到内存用于后面的records的处理。这个特性导致用户对象具有固定的生命周期，下面通过用户代码模版来分析用户对象的固定生命周期及其相对应的输入records。

**用户代码模版：**

用户代码来自现实世界的Google/Hadoop MapReduce来自，包括MapReduce Design Pattern书中提到的例子。

**map()产生的用户代码对象生命周期**

下面代码显示了map() 的代码结构及其里面的用户代码对象。map()的编程范型很简单。map(K, V) 每读入一个<k, v> record 就处理一个，所以每新来一个record，map(K, V)就会被调用一次。所以map()中的用户对象生命周期有两种：(1) map-level（用①标示），定义在map(K,V)之外的的对象，如mapLevelBuffer，会一直在内存，只到所有的records处理完后，mapper才会将其回收。(2) record-level（用②标示），定义在map(K,V)里面的用户对象（比如iResults）只在当前record被处理时存在，在下一个record读入前，当前的iResults会被清除，除非被cache到map-level buffer里。

|  |
| --- |
| **public class Mapper** {  private Object mapLevelBuffer; // ①  **public void** **map(K key, V value)** {  // record-level中间计算结果，可以被存放在mapLevelBuffer  Object iResults = **process**(key, value); // ②  emit(newKey, newValue); // using iResults  }  } |

所以record-level用户对象与当前被处理的单个record有关，而map-level的用户对象与map()读入的全部records有关。下图显示了这两种关系。



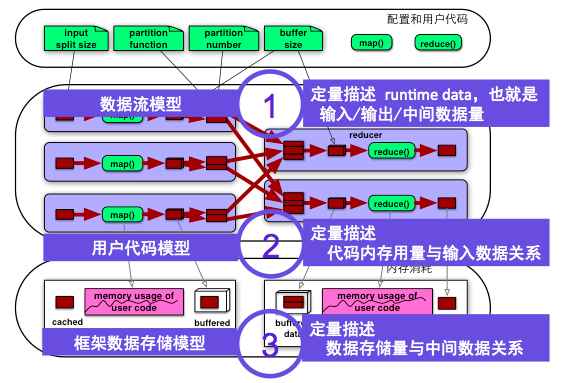
**reduce()产生的用户代码对象生命周期**

下面代码显示了reduce() 的代码结构及其里面的用户代码对象。Reduce()的编程范型略复杂。Reduce(K,list(V))负责处理<k, list(v)>，每新来一个<k, list(v)>，reduce(K, list(V))就被调用一次，<k, list(v)>中的values被一个个读入并处理。Reduce(K,list(V))中的用户对象生命周期有三种：(1) reduce-level（用③标示），定义在reduce(K, list(V))之外的对象，如reduceLevelBuffer，会在内存中一直存在，直到所有的<k, list(v)>组被处理完。所以reduce-level的对象大小与所有的<k,list(v)>组有关。(2) group-level（用②标示），定义在reduce(K, list(V))里面，但在for()外面的对象会在<k, list(v)>被处理的时候一直存在，直到当前的<k, list(v)>被处理完后回收。所以group-level的用户对象与当前<k,list(v)>中的所有records相关。(3) record-level（用①标示）定义在for()内部的用户对象与当前被处理的record有关，当下一个record（也就是下一个value）被读入前会被回收，除非被cache到高层（group-level或reduce-level）的buffer里面。Group-level和record-level用户对象的关系与map()里面map-level与record-level用户对象关系类似。Combine()中用户对象的生命周期与reduce()的情况相同，因为他们具有相同的编程范型。

|  |
| --- |
| **public class Reducer** {  private Object reduceLevelBuffer; // ③  **public void reduce**(K key, Iterable<V> values) {  Object grouplevelBuffer; // ①  **for (V value : values)** {  // record-level 中间计算结果，  // 可以被 cache 到 group-/reduce-level 的 buffer 里  Object iResults = process(key, value); // ②  emit(newKey, newValue); // may be here  }  emit(newKey, newValue); // may be here too  }  } |

* 1. 模型建立方法

方法：以数据流为中心构建静态因素-数据流-内存用量定量关系



* 1. 内存用量模型

### 建立数据流模型

数据流特点：数据依赖关系固定（Partition-aggregation）

在 t 时刻，数据量(当前处理阶段) = f (已经执行完的阶段数据量)

Job数据流



方法：根据基本数据依赖关系建立模型

定量描述输入/输出数据关系

函数内嵌配置参数，如

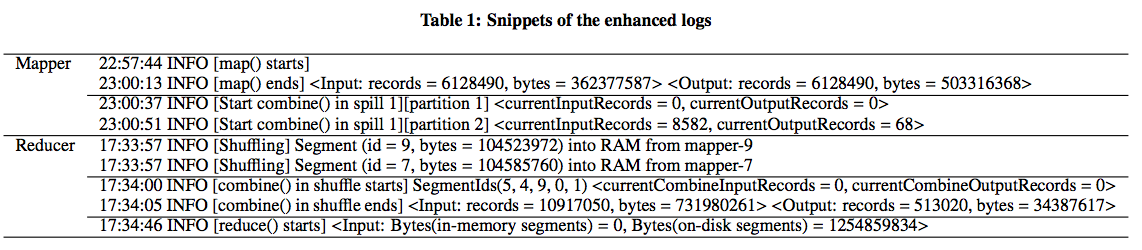
M 表示 mapper 个数

Np 表示 partition number

具体数据量可以通过小数据集上运行的 job 的数据流按比例得到

|  |  |  |
| --- | --- | --- |
| **基本的数据依赖关系** | **输入数据** | **输出数据** |
|  |  |  |
|  |  |  |
|  |  |  |

数据流模型加上partiton比例描述。加上log和dataflow counters

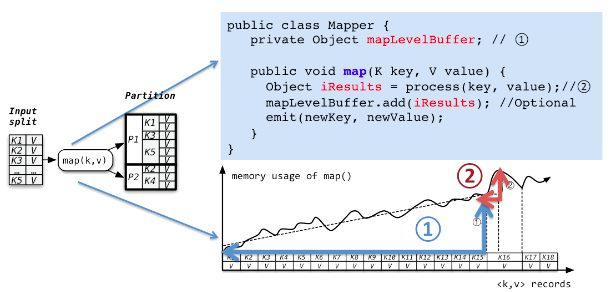


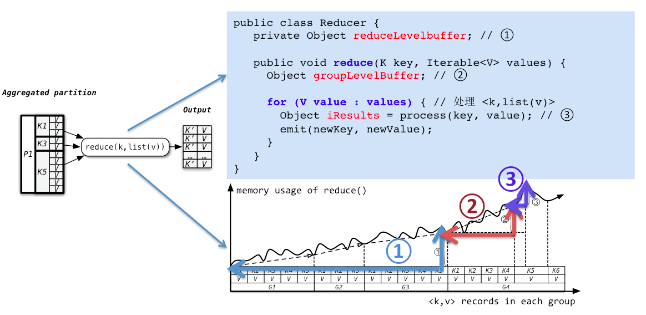
### 建立用户代码模型

观察得到：Map 代码特点（Streaming style）

对象具有固定的生命周期 ① code-level ② record-level

对象与特定区域的数据有关





用户代码模型建立方法

对于生命周期的对象建立不同函数关系

Size(code-level objs) = f (records in all the group)

Size(group-level objs) = g (records in current group)

Size(record-level objs) = h (current record)

O(user objects) = O(code-level objects) + O(group-level objects) + O(record-level objects)

用户代码模型建立方法

对于生命周期的对象建立不同函数关系

Size(code-level objs) = f (records in all the group)

Size(group-level objs) = g (records in current group)

Size(record-level objs) = h (current record)

通过动态监控＋曲线拟合方法得到空间复杂度 f, g, h

动态监控方法 🡪 内存使用趋势曲线

使用曲线拟合（如 linear regression）方法得到 f, g, h

内存模型

MemoryUsage(task) = O(user objects) + O(buffered data)

用户代码模型

O(user objects) = f,g,h (input data, user code)

数据流模型

O(buffered data) = f (intermediate data, buffer size)

O(input / output / intermediate data) = f (upstream data, confs)

* 1. 内存用量预测方法

离线训练阶段

实验验证方法：给定 job <data, configurations, user code>，预测其 task 的内存用量

具体实验步骤

① 先在小数据集上运行，计算中间数据量及空间复杂度 f, g, h

② 估算在大数据上运行时 task 的内存用量

③ 运行真实的 job，对比实际内存用量与估算用量的误差

* 1. 实验与评价

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **应用** | **数据集大小** | **配置参数** | | |
| **Input split size (MB)** | **Reduce number** | **Buffer size (MB)** |
| WikiWordCount | 9.4 GB | 64, 128, 256 | 9, 18, 36 | 200, 400 |
| BuildInvertedIndex | 9.4 GB | 64, 128, 256 | 9, 18, 36 | 200, 400 |
| Pig-UserVisitAggr | 75 GB | 64, 128, 256 | 9, 18, 36 | 200, 400 |
| TwitterBiEdgeCount | 24.4 GB | 64, 128, 256 | 9, 18, 36 | 200, 400 |
| TeraSort | 36 GB | 64, 128, 256 | 9, 18, 36 | 200, 400 |

评价方法

### 实验环境和设置

实验环境

### 实验对象与数据

图4.11中 。

### 实验结果

误差来源

抽样比例：10GB => 1GB

数据倾斜：WordCount 中某些 Word 出现的频率比其他的 Word 高很多

### 进一步的讨论

上述 。

* 1. 相关工作比较

指导意义。

* 1. 本章小结

是本章重点解决的问题

# 应用内存溢出诊断方法

本文。

* 1. 引言

内存溢出错误已经在第三章介绍过，本章希望设计出一种内存溢出错误的诊断方法。当前的内存分析工具和内存泄漏检测方法只能分析出内存中有哪些对象，哪些对象本应该释放却没有释放。但是，这些工具和方法不能分析出这些对象的来源及对象变大的原因。

在这本章，我们研究发现不合适的job的配置参数，数据倾斜，和内存使用密集的用户代码都可以导致应用的内存消耗量过大。基于此观察，我们提出一个半自动化的方法，称为以数据流为中心的内存用量回溯dataflow-centric memory backtracking。我们的方法首先自动化地建立内存用量模型来定量描述内存用量－数据流－配置参数的关系，然后通过设计的定量规则定位内存溢出错误相关的代码，错误相关的数据，和有问题的配置参数。

* 1. 问题分析

当map/reduce任务的内存空间不够用的时候，会产生内存溢出错误。下图显示了两个分别发生在map和reduce任务的内存溢出错误。当内存溢出错误发生时，用户只能从错误栈中发现正在运行的函数（方法）， 当时无法直接看出错误原因。

诊断溢出错误的挑战性：(1) 静态配置不直接影响分布式运行的map/reduce任务的内存用量。(2) 用户代码可以被任意书写，也可以由高层语言（比如类SQL的Pig脚本）或高层库（比如Mahout）产生。这样，对诊断来说，用户代码相当于黑盒。

当前的内存分析工具，比如Eclipse MAT [MAT]，JProfiler [JProfiler]，只能分析当前内存（堆）里面有哪些对象及对象的特征（如大小，引用关系）。静态和动态的内存泄漏诊断方法可以诊断出哪些对象（可能）没有被释放。但不管是内存分析工具还是内存泄漏诊断工具都不能分析出内存对象的来源及对象为什么会变得如此大。另外，内存溢出错误的主要原因时内存消耗过大，而非内存泄漏。

不同于单机程序，MapReduce有三大特点，这些特点可以帮助我们诊断内存溢出错误。三个特点：(1) 用户需要配置合适的框架缓冲区大小来平衡框架和用户代码的内存消耗。(2) MapReduce框架分布处理<k, v> records，在运行时对<k, v> records进行划分（partition）和聚合（aggregation）。(3) 用户代码以streaming-style的方式读入、处理、输出key/value records。

根据第三章的研究发现，我们总结出三个内存溢出的原因：(1) 不合适的配置参数可以导致框架暂存过多的中间数据。(2) 数据流异常（数据倾斜）可以导致运行时产生异常大的中间数据。(3) 内存使用密集的用户代码可以导致用户代码在处理数据时产生大量的内存消耗。基于以上的研究发现，我们诊断内存溢出错误的想法是先对数据流进行建模，量化数据之间的依赖关系，然后建立内存用量模型来量化内存用量－数据流－配置参数之间的关系。基于这个量化关系，我们可以设计量化规则来将内存溢出错误归因到错误相关的代码段、错误相关的数据，及有问题的配置参数。

在本章中，我们提出一种半自动化的错误诊断方法，叫做以数据流为中心的内存错误归因（dataflow-centric memory backtracking）。该方法首先建立一个数据流模型来量化各个处理步骤的数据依赖关系，然后建立用户代码模型来量化用户对象和代码输入数据之间的关系。数据流模型建立方法是去量化MapReduce数据处理过程中的基本数据依赖关系。建模过程中的一个最重要挑战是很难去量化用户代码产生的中间结果（用户对象）与用户代码输入数据之间的关系。我们发现用户对象有不同但固定的生命周期，基于此，我们设计了一个生命周期敏感的内存监控策略（lifecycle-aware memory monitoring strategy）。通过使用这个策略，我们可以直接量化用户代码产生的不同生命周期对象与其相关数据的关系。最后，我们设计出两种不同的量化策略来定位错误相关的代码、数据和配置参数。这两种规则是：(1) 面向用户代码的规则（rules for user code），可以定位出用户代码中的错误类型和错误相关的输入数据。(2) 面向数据流的规则（rules for dataflow），该类规则基于数据流模型可以定位出数据流异常的地方和有问题的配置参数。

我们在20个真实MapReduce应用出现的内存溢出错误上进行了实验，这些应用包括手写的MapReduce代码，Apache Pig，Apache Mahout，Hadoop Cloud9工具，等等。这些实验表明我们的方法可以正确诊断出15个内存溢出错误的原因，可以诊断出5个错误的部分原因（这5个应用的用户代码有高层语言书写）。

我们的主要贡献是

我们定性分析了内存溢出的错误原因

我们提出一个以数据流为中心的错误诊断方法，该方法可以将内存溢出错误归因到用户代码段、错误相关的数据和有问题的配置参数。

我们设计了一个基于对象生命周期的内存用量监控方法，该方法在用户代码是黑盒的情况下量化用户对象与代码输入数据之间的关系。

* 1. 内存溢出错误诊断

内存溢出错误诊断的目的是：(1) 诊断出内存消耗量大的用户代码，包含用户代码的错误类型及内存消耗高的代码段。(2) 诊断出内存溢出错误相关数据。(3) 诊断出不合适的配置参数。

* 1. 诊断工具

数据流分析工具

给出数据流异常的规则（dataflow counters），怎么fit model

生命周期敏感的内存用量监控策略

在内存溢出错误发生的时候，用户对象由中间计算结果（record-level）和累积计算结果（map-level，group-level或reduce-level）组成。因为这些结果有固定的生命周期，而且不同的生命周期的结果与输入数据中的不同部分紧密相关，我们将大问题（量化用户对象与输入数据之间的关系）化成小问题（分别量化累积计算结果和中间计算结果与其相应的输入数据之间的关系）。

**量化累积计算结果与其输入数据之间的关系：**

为了还原出累积计算结果的是如何被积累的，以及其与输入数据之间的关系，我们选择在一些特殊的时间点进行heap dump（将当前时间点的JVM heap进行快照，并将快照存放到磁盘）。对于map-level的累积计算结果，可选的时间点是当用户代码刚刚完成当前的record处理（R*i*），并将去处理下一个record（R*i*+1）。在这些时间点，record-level的中间结果已经被清理，所以当前heap中的用户对象大小即是当前累积结果的大小。对于reduce-level的累积结果来说，可选的时间点是用户刚刚处理当前的<k,list(v)>组（G*i*），正准备去处理下一个<k,list(v)>组(G*i+*1)。为了效率，我们并不是在每个R*i*和G*i*处都进行heap dump，而是隔一些R*i*和G*i*进行heap dump（如下图中的竖线所示）。对于map-level的累积结果，我们选择每隔 (n-1)/m个records进行heap dump，n是内存溢出发生时已经被处理过的records个数，而m是想要的heap dump个数（由用户设置）。对于reduce-level的累积结果来说，n是内存溢出错误发生时已经被处理过的<k, list(v)>组个数。对于group-level累积结果来说（我们只关心在最后一个<k,list(v)>组里面的group-level累积结果），内存用量监控策略与map()中的策略是一样的。最后我们抽取并计算每个heap中的用户对象大小，然后将这些对象大小连接起来作出累积结果的内存使用趋势线。图中Mi和Si分别表示在第i个record/group间隔的积累率（accumulation rate）。另外，在R1和G1时的用户对象大小表示被用户加载到用户代码的外部数据大小。



**量化当前中间结果与当前输入record的关系：**

我们只关心在当前（也就是内存溢出错误发生时正在处理的record）Rn对应的。所以，要选择的heap dump时间点是：当Rn要被处理，当处理Rn发生内存溢出错误。这个heap之间的对象大小差（比如上图中的Mo）可以被当作当前中间结果大小。然而，我们仍然需要去检测这个 record-level中间计算结果是只与Rn有关还是与之前的records也有关。检测方法：我们重新运行用户代码，并让它只处理Rn（跳过其他的records，Hadoop框架支持跳过skip机制），然后重新计算Rn要被处理和处理后的对象大小差，如果这个差与Mo相同，那么我们可以断定当前的record-level中国计算结果只与Rn有关。

**实施监控策略：**

当内存溢出错误发生的时候，我们重新运行job，然后同时将在job上应用上面的监控策略。因为重新运行的job的输入数据、配置参数和用户代码均未改变，内存溢出错误还会像上次一样发生。唯一非确定性（non-deterministic）的地方是如果reducer运行两次的话，reducer可能shuffle到不同次序的数据分块（data partition）。为了避免这个问题，我们在第一次job失效时记录shuffle到的数据分块顺序，然后在重新运行的job 上重放（replay）这个顺序。

* 1. 诊断规则



上图显示了内存溢出错误的诊断过程。诊断过程先定位内存消耗较大的代码段，然后将内存消耗量大的用户对象归因到错误相关的数据和配置参数。具体包含四个步骤。

### 识别内存消耗量大的代码段

这一步首先从内存溢出时的heap dump抽取大的用户对象，然后识别出引用这些对象的代码段（方法）。一个heap dump就是一个object图，图中每个节点代表一个对象，每条边代表对象的引用关系。为了抽取出内存消耗量的对象，首先将对象图转换为dominator tree [DominatorTree]。然后通过识别对象被引用的方法来获得大的用户对象（比如3MB+）。比如在上图中，如果ArrayList被method2()引用，也就是被reduce()间接引用，那么这个对象被认为是reduce()产生的用户对象。通过计算每个method引用的用户对象大小，可以定位内存消耗量大的代码段。比如在图4中，80%（比如80MB）的用户对象被method1()引用，50%的用户对象被method2()引用，那么内存消耗量大的代码段是method1()和method2()。下表显示一个411MB的用户对象wcMap:HashMap被map()引用，，内存消耗量大的代码段是Mapper.map(InMemWordCount.java:49)。

|  |
| --- |
| at java.util.HashMap.put(HashMap.java:372)  at mapper.InMemWordCount$Mapper.map(InMemWordCount.java:49)  **=> wcMap java.util.HashMap @ 0xdbc5e0c0 (430,875,440 B)**  at org.apache.hadoop.mapreduce.Mapper.run(Mapper.java:203)  ...  at org.apache.hadoop.mapred.Child.main(Child.java:404) |

框架对象（也就是框架缓冲区和在内存中暂存的中间数据）可以通过直接识别对象名称来从heap dump抽取出来。比如在Hadoop MapReduce中，org.apache.hadoop.mapred.Merger.Segment表示的就是暂存的数据，kvbuffer表示的是map buffer。

### 识别用户代码中的错误类型和错误相关的数据

基于用户对象的固定生命周期以及还原出的用户代码内存消耗趋势，我们设计除了5个统计规则来识别用户代码中的错误类型以及错误相关的数据。如果某个规则r对应的错误现象出现，那么该规则生效。比如用户对象在[R1, Rn)上如果出现线性增长，那么我们可以判定map()出现了map-level的累积计算结果，其对应的错误相关的数据是map()所有的输入records。当前的rules虽然不完整，但是具有扩展性。比如，除了线性增长规则，多项式和指数增长方式也可以反应出积累效果。然而，这些复杂的增长趋势需要使用复杂的函数来拟合。为了简单，我们当前只考虑线性增长情况。对于其他的不能匹配到情况，我们的方法会输出最陡峭的N个内存增长区间及其相关的数据。然后，用户需要手工去断定哪些区域是最主要的错误相关的数据。

Rule 1,4,5被用于map()，而rule 2,3和4被用于reduce()。在Rule 1中，如果用户对象的大小与[R1, Rn)与线性相关，那么错误类型被断定是map-level的累积结果。下一个执行步骤（这里是检查input split size）会识别出input split size是不合适的配置参数。其他rules（2到5）具有同样的工作方式。如果rule 2被匹配到，那么错误相关的数据是aggregated partition，下一个动作就是依据数据流模型，调用rule 6-1和6-2去检测data partition是否异常。Rule 3的下一步是去调用rule 7去检测热点key是否存在。Rule 4的下一步是调用rule 8去检测当前的Rn是否是一个异常大的record。

用户代码规则：用于诊断用户代码的错误类型和错误相关的数据

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | 现象 | 错误原因 | 错误相关数据 | 下一步操作 |
| 1 | [R1,Rn]内存用量是线性／多项式／指数增长 | Map-level中间结果累积太大 | Map()所有的输入数据 | 检测input split size是否过大 |
| 2 | [G1,Gn]内存用量是线性／多项式／指数增长 | Reduce-level中间结果累积太大 | Reduce()的所有输入数据 | 检测数据划分异常 (Rule 6) |
| 3 | Gn组内存用量是线性／多项式／指数增长 | Group-level中间结果累积太大 | Gn组里的所有records | 检测热点key (Rule 7) |
| 4 | Rn处内存用量陡峭增长 | Record-level中间结果太大 | 当前的输入record | 检测单一record是否过大 |
| 5 | R1或G1处的内存用量是外部数据量的m倍 | 用户加载了大量外部数据 |  | 输出m |

什么是陡峭增长，线性／非线性增长？

### 识别数据倾斜和不合适的配置参数

基于识别出的错误相关的数据和已经量化的数据流，我们设计了三个统计规则（6，7和8）来识别数据倾斜和不合适的配置参数。比如在Rule 7中，如果在当前<k, list(v)>组Gn中出现了陡峭增长（相比其他Gi中的内存增长斜率是outlier [Outlier]），而且Gn比其他groups包含更多的records。最后，Rule 9-1和9-2被用于判定错误原因是否是框架暂存了大量数据。

数据流规则：用于诊断数据流异常和不合适的配置参数

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **现象** | **错误原因** | **不合适的配置参数** |
| 6-1 | Rule 2生效且data partition是平衡的 | 不合适的数据划分 | Reduce number太小 |
| 6-2 | Rule 2生效但data partition不平衡 | 不合适的数据划分 | 非平衡的partition函数 |
| 7 | Gn组出现陡峭增长，且Gn具有比他组更多的records | 热点key | Partition函数中key的选择 |
| 8 | Rule 4生效且Rn比其他Ri都大 | 大的单一record | 无 |
| 9-1 | 固定buffer存在且map()将要完成 | 框架缓冲区太大 | 固定buffer太大 |
| 9-2 | 虚拟buffer被填满，且combine()/reduce()快要完成 | 框架缓冲区太大 | 虚拟buffer太大 |

什么是outlier？

* 1. 实验评价

实验目的是去验证我们的方法诊断出来的错误原因是不是真正的错误原因。

### 实验设置

我们重现了20个现实世界的内存溢出错误，这些错误发生在不同类型的MapReduce应用里面，用户代码包含用户手写的MapReduce代码，Apache Pig，Hive，Mahout，Cloud9等高层语言或高层库产生的代码。这些错误来自公开论坛，例如StackOverflow.com，Hadoop mailing list，和开发者的博客，这些错误有详细的输入数据信息，可重现的用户代码，内存溢出错误栈等等。因为我们没有用户原始的输入数据，我们使用公开数据集（英文维基百科）和合成数据集（随机文本和一个知名的用于对比MapReduce和并行数据库性能的benchmark）。我们确认我们重现的内存溢出的错误栈与用户报告的相同。所有的应用都跑在我们实验室11个节点的Hadoop集群上，使用的Hadoop版本是我们修改过的Hadoop-1.2，这个版本可以支持自动在第i个input record/group上面进行heap dump。集群里每个节点配备16个节点的内存，每个map/reduce任务的heap大小设置为1GB。默认情况下，我哦们配置input split size是512MB，buffer size是500MB，reduce number是10，partition函数是HashPartition。

错误原因诊断：

对于每一个job，我们首先将其按照一个正常job运行，然后记录在内存溢出发生时已经处理的record数目。之后，我们重新运行这个job，并使用我们的内存用量监控策略来产生heap dump。我们修改过的Eclipse MAT可以自动从每个heap dump中抽取用户对象和框架对象，并计算它们的大小。最后，我们开发了一个诊断工具DiagOOM来完整整个诊断流程，定位出错原因。

诊断结果

下表展示了我们的方法的诊断结果，包含job名字（里面链接指向真实的应用），匹配到的规则，以及诊断出的错误原因是不是真正的错误原因。其中15个错误原因已经被专家（Hadoop committer或有经验的开发者诊断出来），剩下的5个错误（标记为＋）通过间接验证的方法出来（比如尝试一些修复方法来看内存溢出错误是否会消失）。对号表示我们的方法诊断出来的错误原因与真实的错误原因一样。半对号表示我们诊断出来的错误原因与真实的错误原因部分相同，除了内存消耗量大的代码端不能诊断出来（因为这5个应用的用户代码由高层语言或高层库产生）。下面的案例研究展示了我们的方法如何利用量化后的“内存用量－数据流－配置参数”和量化的规则来诊断出来错误原因。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **阶段** | **Job名称** | **规则** | **错误原因** | **诊断正确** | **验证（修复方法）** |
| Map | [NLPLemmatizer](http://stackoverflow.com/questions/20247185/java-lang-outofmemoryerror-on-running-hadoop-job) | 4,8 | 中间计算结果过大，单个record过大 | ✓ | 将Rn拆分成多个小的records |
|  | [InMemWordCount](http://puffsun.iteye.com/blog/1902837) | 1 | Map-level的中间累积结果 | ✓ | 降低input split size |
|  | [MapSideAggregation](http://stackoverflow.com/questions/16684712/out-of-memory-due-to-hash-maps-used-in-map-side-aggregation) | 1 | Map-level的中间累积结果 |  | 降低input split size |
|  | [PigDistinctCount](http://mail-archives.apache.org/mod_mbox/pig-user/201105.mbox/%3CBANLkTi=Tnc8icJo48LKQDhEuT=jXBpD+oA@mail.gmail.com%3E) | 3 | Group-level的中间累积结果 |  | 减小<k, list(v)>组 |
|  | [CDHJob](http://stackoverflow.com/questions/13674190/cdh-4-1-error-running-child-java-lang-outofmemoryerror-java-heap-space) | 9-1 | 固定缓冲区过大 | ✓ | 降低map buffer大小 |
|  | [MahoutBayes](http://stackoverflow.com/questions/10080800/outofmemory-error-when-running-the-wikipedia-bayes-example-on-mahout) | 5 | 加载大量外部数据 | ✓ | 减小外部数据（训练模型） |
|  | [MahoutConvertText](http://stackoverflow.com/questions/22921936/mahout-exception-java-heap-space) | 5 | 加载大量外部数据 | ✓(+) | 减小外部数据（训练数据） |
|  | [HashJoin](http://stackoverflow.com/questions/15316539/hadoop-mapper-over-consumption-of-memoryheap) | 5 | 加载大量外部数据 | ✓(+) | 减小外部数据（第一个表） |
| Shuffle | [ShuffleInMemory](https://issues.apache.org/jira/browse/MAPREDUCE-5580) | 6-1 | 不合适的数据划分 | ✓(+) | 增加reducer个数 |
|  | [WordCount-like](http://stackoverflow.com/questions/19298357/out-of-memory-error-in-mapreduce-shuffle-phase) | 9-2 | 虚拟缓冲区过大 | ✓ | 降低缓冲区大小 |
|  | [PigJoin](http://stackoverflow.com/questions/17162679/pig-join-gets-outofmemoryerror-in-reducer-when-mapred-job-shuffle-input-buffer-p) | 9-2 | 虚拟缓冲区过大 | ✓ | 降低缓冲区大小 |
|  | [NestedDISTINCT](http://mail-archives.apache.org/mod_mbox/pig-user/201201.mbox/%3C4F174294.8090509@cern.ch%3E) | 3 | Group-level累积中间结果 | (+) |  |
|  | [PigOrderLimit](https://mail-archives.apache.org/mod_mbox/pig-user/201201.mbox/%3CD570DEB688737C44A53497A16D0A7CAC0789B0@EAGF-ERFPMBX42.ERF.thomson.com%3E) | 3,7 | Group-level累积中间结果，热点key |  |  |
| Reduce | [GraphPartitioner](http://stackoverflow.com/questions/12831076/oom-exception-in-hadoop-reduce-child) | 3 | Group-level累积中间结果 | ✓ |  |
|  | [FindFrequentValues](http://stackoverflow.com/questions/23042829/getting-java-heap-space-error-while-running-a-mapreduce-code-for-large-dataset) | 3 | Group-level累积中间结果 | ✓ |  |
|  | [ReduceJoin](http://stackoverflow.com/questions/15541900/why-does-the-last-reducer-stop-with-java-heap-error-during-merge-step) | 3,7 | Group-level累积中间结果，热点key | ✓ |  |
|  | [PositionalIndexer](http://www.cs.cmu.edu/~lezhao/TA/2010/HW2/) | 3,7 | Group-level累积中间结果，热点key | ✓ |  |
|  | [BuildInvertedIndex](http://stackoverflow.com/questions/17980491/building-inverted-index-exceed-the-java-heap-size) | 3,7 | Group-level累积中间结果，热点key | ✓ |  |
|  | [CooccurMatrix](http://mail-archives.apache.org/mod_mbox/hadoop-common-user/201010.mbox/%3CAANLkTi=aNjiUezv-a9yFZpbXXWFsbjeKKyd2KmqCUAWc@mail.gmail.com%3E) | 3,7 | Group-level累积中间结果，热点key | ✓ |  |
|  | [JoinLargeGroups](http://stackoverflow.com/questions/22281188/fail-to-join-large-groups) | 3,7 | Group-level累积中间结果，热点key | (+) |  |

### 案例研究

案例1: InMemWordCount

这个应用的目的是去统计维基百科中每个单词出现的次数。这个例子与MapReduce原始论文 [MapReduce] 中给出的WordCount的例子的功能一样。唯一的差别是这里的map()分配了一个HashMap来聚合每个word产生的中间结果<word, count>，而非像原始的WordCount那样直接输出<word, count>。这里，每个record的value是一个line，而key是这个line在文本中的位置。但从源码看来，我们只能猜测错误原因是wcMap缓存了太多的<word, count>中间结果。然而，原因也可能是StringTokenizer st在处理一个异常大的line的时候发生内存溢出错误。在不知道代码语义的情况下，我们的方法先去还原map()用户代码的内存用量趋势（见图5a）。然后，我们的诊断工具发现[R1,Rn)上的内存用量具有线性增长趋势，这里n=9,651，而且R9,651处的内存增长只有（Mo=1.2MB）。所以，Rule 1被匹配，错误原因是map-level的累积中间结果过大（也就是产生了大量的<word, count>中间结果）。更确切地，诊断工具发现累积的计算结果被存放在一个411MB的wcMap:HashMap里面，这个wcMap被wcMap.put(word,1)所引用（用=>标示）。我们诊断出的错误原因与有经验的应用开发者诊断出的错误原因一致：wcMap存放了太多的<word, count>中间结果。Rule 1的下一步是去诊断出不合适的配置参数是input split size，这里是512MB。为了修复这个错误，用户可以降低input split size来减少相应的累积结果，或者仍然使用最初版本的WordCount（没有累积操作）。

|  |
| --- |
| **public class Mapper** {  Map wcMap = new HashMap<String, Integer>();  **public void** map(Long key, Text value) {  st = **new** StringTokenizer(value.toString());  **while** (st.hasMoreTokens()) {  String word = st.nextToken();  **if** (wcMap.containsKey(word))  wcMap.**put**(word, wcMap.get(word) + 1);  **else**  wcMap.**put**(word, 1); => wcMap:HashMap (411MB)  }  }  } |

案例2: NLPLemmatizer：

这个应用通过使用一个第三方库StanfordLemmatizer来对维基百科中的words进行屈折变换（lemmatize）。每个record的value是文本中的一行（line），key是该行（line）在文本中的位置。从代码中我们不能直接确定错误原因，原因可能是slem累积了太多的中间结果，当前record产生的中间结果太大，或者两者都有。在还原出map()的内存使用趋势（见图5b）后，我们的诊断工具识别出[R1,Rn)，这里n＝9，上没有一个线性增长趋势，但R9处有一个陡峭增长，而且这个陡峭增长只与R9有关。所以Rule 4被匹配，得到错误原因是record-level中间计算结果过大。更确切地，这个record-level计算结果包含一个232MB的ArrayList和一个50MB的String，这两者都被slem.lematize()引用。然后，Rule 4的下一步是去调用Rule 8去计算R9的大小，计算得到R9的大小是50MB（属于特别大的line），远远大于其他的records（12MB）。所以，Rule 8被匹配到，错误原因是单一record过大。我们诊断出的错误原因与专家（库作者）诊断出的相同：lematize()在对每个line进行打标签（tag）的时候会是分配一个大的临时的数据结构，这个数据结构一半比要处理的line大好多被。为了修复这个错误，用户可以将R9分割成多个小的records（line）。

|  |
| --- |
| **public class Mapper** {  StanfordLemmatizer slem = new StanfordLematizer();  **public void** map(Long key, Text value) {  String line = value.toString();  **for** (String word: slem.lemmatize(line)) => 282MB  emit(word, 1);  }  } |

案例3: CooccurMatrix：

这个应用目的是去计算维基百科中词共现矩阵（word co-occurrence matrix）。 Reduce()分配了一个自定义的数据结构（叫做OHMap）来聚合每个<k,list(v)> group里面的records。每一个key是一个word，value是一个OHMap，用来持有该word的邻居words。只分析源代码的话，我不能确定plus()的语义，更别说错误原因了。我们的诊断工具首先识别出[G1,Gn) (n = 3,897,853)上没有线性增长关系，因为[G1,Gn]上的增长斜率（S1~S3,897,852）是0。然而，在Gn（n = 3,897,853）上的内存用量增长是线性增长。所以，Rule 3被匹配到，错误根源是group-level的中间累积结果（也就是wordMap持有了太多word的临近词）。这些累积结果被存放在一个372MB的OHMap里面，被引用的代码段是wordMap.plus(iter.next())。然后，Rule 3的下一步调用Rule 7来检测产生这么多word的原因是不是热点key。我们的诊断工具发现G3,897,853比其他的<k, list(v)>组要多6倍的records，而这个k是“of”。另外，在当前的R17处，有一个内存陡增，然而这个陡增（用红色虚线标示）与G3,897,853里面的records都相关，因为当reduce()单独处理R17的时候，内存的增量只有30MB。错误栈中的关键词“Arrays.copyOf()”告诉我们这个陡增的原因是数据结构膨胀。一些容器型的数据结构，比如ArrayList和HashMap，可以在快要被填满的时候扩张1.5或者2倍。为了修复这个错误，用户可以尝试去将累积的中间结果周期性地spill到磁盘上。

|  |
| --- |
| **public class Reducer** {  **public void reduce**(Text key, Iterable<OHMap> values) {  Iterator<OHMap> iter = values.iterator();  OHMap wordMap = **new** OHMap();  **while** (iter.hasNext()) { // for(V value: values)  wordMap.**plus**(iter.next()); => wordMap (287MB)  }  emit(key, wordMap);  }  } |

案例4: PigDistinctCount：

内存溢出错误栈现实这个错误发生在combine()中。然而，很难去确定错误原因，因为这个应用是上层语言Pig产生的，没有显式的用户代码。该应用的整个处理过程如下：该脚本首先对tableA中的pagerank那一列进行groupBy操作，然后在每一个pagerank的group里面进行count(distinct pageurl)操作，也就是统计每个pagerank值对应的不同的pageurl的个数。

|  |
| --- |
| a = LOAD “tableA” as (pagerank:int, pageurl, aveduration:int);  b = GROUP a BY pagerank;  c = FOREACH b {  urls = DISTINCT a.pageurl;  GENERATE group, COUNT(urls), SUM(a.aveduration);  };  STORE c into “/output/newTable”; |

我们的方法检测出在G1里面内存用量有一个线性增长趋势，G1同时包含747,931个records（见图5d）。所以，Rule 3被匹配，错误原因是combine()在处理G1中的records时，产生了过多的group-level的中间累积结果（也就是包含相同pagerank的不同pageurl）。与案例3相同，R747931上的陡峭增长原因也是数据结构扩张。更确切地，累积的中间结果包含一个77MB的pig.data.BinSedesTuple和281MB的pig.data.InternalDistinctBag，这两个大对象都被PigCombine.processOnePackageOutput()引用。然后查看Pig的API会发现，第一个对象InternalDistinctBag会将产生的中间数据存放到一个ArrayList里面进行排序。然后，如果想要定位产生这个group-level累积结果的具体操作，我们可以进一步去查看Pig守则 [PigManual]，然后会发现DISTINCT操作是错误的来源。这个操作会生成一个数据结构（也就是InternalDistinctBag）来持有所有的输入records，然后将这些records进行排序和去重。为了修复这个错误，用户可以选择不同的key，或者尝试将DISTINCT操作分割成多个流式的操作（利用框架的排序机制）。

* 1. 方法缺陷及讨论

当前方法的最大问题是需要一些用户手工操作：(1) 当一些用户代码用量出现一些特殊的趋势（比如含有大幅度跳跃的情况），我们的规则不能被匹配到，用户需要判定哪个区域的输入records是错误相关的（我们的工具可以判定top N内存增幅相关的输入数据）。(2) 用户需要将诊断出来的错误类型（比如group-level的累积结果与代码语义结合起来），特别是当用户代码是由高层语言产生的时候。自动理解用户代码的语义是困难的。

我们当前的方法没有增加应用运行时的overhead，仅仅依靠重新运行时的数据流信息和heap dumps。唯一的overhead是需要用户重新运行一下job产生所需的heap dumps。

尽管我们的方法是针对MapReduce框架设计的，也可以用于其他的分布式处理框架如Dryad和Spark。虽然这些MapReduce-like的框架有灵活的基于DAG的数据流，但他们的基本数据依赖关系与MapReduce数据流相同。这些框架另一个特点是支持pipeline。然而，这些框架的用户代码仍然是一个个处理record（也就是streaming-style），所以用户代码对象仍然有固定的生命周期。另外，job的配置参数如buffer size，partition number和partiton函数同样在这些框架有效。

Macintosh HD:Users:xulijie:Documents:Texlipse:BigData2015:figures:evaluation.pdf

* 1. 相关工作比较

通用编程语言里的内存相关问题已经被深入研究过，但是内存溢出错误诊断的工作很少，尤其是大数据应用中的内存溢出问题。

**内存泄漏检测：**

内存泄漏是常见错误而且可以导致内存溢出，所以研究人员提出了很多内存泄漏的检测方法。比如，flow-sensitive分析[FlowAnalaysis] 将内存泄漏诊断问题变换为可达性问题。Cork [Cork] 利用对象的类型增长信息来检测包含无用对象的数据结构。Container profiling [ContainerProfiler] 会跟踪container上的操作来定位没有别使用到的数据元素。然而，这些检测方法不能被直接用于数据并行应用的内存溢出错误诊断，因为这些方法不能量化内存用量－数据流－静态配置参数之间的关系。在我们的实验中，我们还没有发现

* 1. 本章小结

内存溢出错误是MapReduce应用中常见且严重的运行时错误。我们研究发现不合适的配置参数，数据倾斜，和内存使用密集的用户代码可以导致内存溢出错误。我们提出了一种以数据流为中心的方法来检测内存溢出错误。我们在真实的内存溢出错误上评测了我们的方法，结果显示我们的数据流量化模型和规则可以有效地诊断出MapReduce应用中的内存溢出错误。在未来的工作中，我们会在更多的应用（比如Spark应用）上评测我们的诊断方法。

# 结束语

本章系统总结了本论文的主要贡献，包括：

* 1. 论文工作总结
  2. 进一步的工作

# 参考文献

[

[ACE] Aliyun Cloud Engine (ACE). http://www.aliyun.com/product/ace/

[Amazon 2011] Amazon ElastiCache. 2011. http://aws.amazon.com/elasticache/

[Ari 2002] Ari I, Amer A, Miller EL, Brandt SA, Long DE. Who is more adaptive? ACME: adaptive caching using multiple experts. In：Proc. of Workshop on Distributed Data and Structures (WDAS '02). 2002.

[Amza 2005] Amza C, Cox AL, Zwaenepoel W. A Comparative Evaluation of Transparent Scaling Techniques for Dynamic Content Servers. In：Proc. of the 21st Int’l Conference on Data Engineering (ICDE '05). 2005. 230-241.

[Bhide 2002] Bhide M, Deolasee P, Katkar A, Panchbudhe A, Ramamritham K, Shenoy P. Adaptive Push-Pull: Disseminating Dynamic Web Data. IEEE Trans. on Computers, 51 (6):652-668, 2002.

[Blum 1998] Blum A, Mitchell T. Combining labeled and unlabeled data with co-training. In：Proc. of the 11th annual conf. on Computational learning theory. 92-100. 1998.

[Brewer 2000] Brewer EA. Towards robust distributed systems. In: Proc. of the 19th Annual ACM Symp. on Principles of Distributed Computing (PODC 2000). 2000.

[Burnham 2002] Burnham KP, Anderson DR. Model selection and multi-model inference: a practical information-theoretic approach. 2nd ed., New York: Springer-Verlag, 2002.

[Chen 2009] 陈康, 郑纬民. 云计算:系统实例与研究现状. 软件学报. 2009, 20 (5): 1337-1348.

[Chiu 2010] Chiu D, Shetty A, Agrawal G. Elastic Cloud Caches for Accelerating Service-Oriented Computations. In：Proc. of the ACM/IEEE Int’l Conf. for High Performance Computing, Networking, Storage and Analysis (SC '10). 2010. 1-11.

[Chockler 2010] Chockler G, Laden G, Vigfusson Y. Data caching as a cloud service. In：Proc. of the 4th Int’l Workshop on Large Scale Distributed Systems and Middleware (LADIS '10). 2010.

[Choi 2002] Choi J, Noh SH, Min SL, Ha EY, Cho YK. Design, Implementation, and Performance Evaluation of a Detection-Based Adaptive Block Replacement Scheme. IEEE Trans. on Computers, 51(7): 793-800, 2002.

[Chong 2008] Chong EKP, Zak SH. An Introduction to Optimization, 3rd ed., New Jersey: John Wiley & Sons Inc., 2008.

[Cloud\_computing 2007] Cloud computing. Wikipedia. 2007. http://en.wikipedia.org/ wiki/Cloud\_computing

[Cohen 1995] Cohen WW. Fast effective rule induction. In：Proc. of the 12th Int’l Conf. on Machine Learning (ICML '95). 115–123. 1995.

[Cooper 2010] Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking Cloud Serving Systems with YCSB. In：Proc. of ACM Symp. on Cloud Computing (SoCC '10). 2010. 143-154.

[Couceiro 2011] Couceiro M, Romano P, Rodrigues L. PolyCert: Polymorphic Self-Optimizing Replication for In-Memory Transactional Grids. In：Proc. of ACM/IFIP/USENIX 12th Int’l Middleware Conference (Middleware '11). 2011. 309-328.

[Couchbase] Multitenancy with couchbase. Couchbase. http://www.couchbase.com /docs/couchbase-manual-1.8/couchbase-introduction-architecture-buckets. html

[Das 2011] Das S, Nishimura S, Agrawal D, Abbadi AE. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration. In：Proc. of the 37th Int’l Conf. on Very Large Data Bases (VLDB '11). 2011. 494-505.

[Dell 2005] Dell white paper. An Overview of Xen Virtualization. http://www. dell.com/downloads/global/power/ps3q05-20050191-Abels.pdf. 2005

[Deolasee 2001] Deolasee P, Katkar A, Panchbudhe A, Ramamritham K, Shenoy P. Adaptive Push-Pull: Disseminating Dynamic Web Data. In：Proc. of Int’l World Wide Web Conference (WWW '01). 2001. 265-274.

[Draper 1998] Draper NR, Smith H. Applied Regression Analysis. 3rd ed., New York: John Wiley & Sons Inc., 1998.

[Elmore 2011] Elmore AJ, Das S, Agrawal D, Abbadi AE. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In：Proc. of the annual ACM SIGMOD Conference (SIGMOD '11). 2011. 301-312.

[Extreme\_Transaction\_Processing 2012] Extreme Transaction Processing. Wikipedia. 2012. http://en.wikipedia.org/wiki/Extreme\_Transaction \_Processing

[Earls 2010] Earls A. Distributed data grids: Foundation for future cloud computing? 2010. http://searchsoa.techtarget.com/news/1518647/Data-Grids- Foundation-for-future-cloud-computing

[Feng 2008] Feng D, Lu C, Wang F. eMuse: QoS Guarantees for Shared Storage Servers. In：Proc. of the 22nd Int’l Conf. on Advanced Information Networking and Applications Workshops (AINAW '08). 2008. 264-269.

[Fetai 2012] Fetai I, Schuldt H. Cost-Based adaptive concurrency control in the cloud. Technical Report, CS-2012-001, Basel: The University of Basel, 2012.

[Gartner 2010] Fenn J. Hype cycle for emerging technologies. Gartner Report, 2010. http://www.planetlarg.net/my-scripts/docs-to-read/gartner/hype\_cycle\_for\_emerging\_tech\_2010.pdf

[GigaSpaces 2009] Security concepts. Gigaspaces XAP. 2009. http://www.gigaspaces.com/ wiki/display/XAP7/Security+Concepts

[GigaSpaces 2010] Making dynamic scaling simple. GigaSpaces XAP. 2010. http://www.gigaspaces.com/xap7-1

[Gilbert 2002] Gilbert S, Lynch N. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant Web services. ACM SIGACT News, 2002, 33(2).

[Gold 2011] Gold E. Extreme transaction processing. 2011. http://natishalom.typepad.com/nati\_shaloms\_blog/files/gigaspaces-dream -machine.pdf

[Gossip 2007] Gossip. Wikipedia. 2007. http://en.wikipedia.org/wiki/Gossip\_protocol

[Gramacy 2003] Gramacy R, Warmuth M, Brandt SA, Ari I. Adaptive caching by refetching. Advances in Neural Information Processing Systems 15. 2003. 1465–1472.

[Gualtieri 2010] Gualtieri M, Rymer JR. The forrester wave: Elastic caching platforms. Q2, 2010. ftp://ftp.software.ibm.com/software/solutions/soa/pdfs/wave\_ elastic\_caching\_ platforms \_q2\_2010.pdf

[Gualtieri 2010\*] Gualtieri M. Elastic caching platforms balance performance, scalability and fault tolerance. 2010. http://blogs.forrester.com/mike\_ gualtieri/10-03-18-elastic\_caching\_platforms\_balance\_performance\_scalability\_and\_fault\_toleranc

[Gulati 2009] Gulati A, Ahmad I, Waldspurger C. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In Proc. of USENIX Conference on File and Storage Technologies (FAST '09). 2009. 85-98.

[Gupta 2005] Gupta D, Gardner R, Cherkasova L. XenMon: QoS Monitoring and Performance Profiling Tool. HP Technical Report (HPL-2005-187). 2005.

[Gupta 2006] Gupta D, Cherkasova L, Gardner R, Vahdat A. Enforcing Performance Isolation Across Virtual Machines in Xen. In：Proc. of the 7th ACM/IFIP/USENIX Middleware Conference (Middleware '06). 2006. 342-362.

[Hahn 2008] Hahn S, Ostendorf M. A Comparison of Discriminative EM-Based Semi-Supervised Learning algorithms on Agreement/Disagreement Classification. In：Proc. of NIPS Workshop on Speech and Language: Learning-based Methods and Systems. 2008.

[Hall 2003] Hall MA, Holmes G. Benchmarking Attribute Selection Techniques for Discrete Class Data Mining. IEEE Trans. on Knowledge and Data Engineering, 15 (6):1437-1447, 2003.

[Hall 1999] Hall MA. Correlation-based Feature Selection for Machine Learning. [PhD thesis]. University of Waikato, 1999.

[Hastorun 2007] Hastorun D, Jampani M, Kakulapati G, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: Amazon’s highly available key-value store. In：Proc. of ACM Symp. on Operating Systems Principles (SOSP '07). 2007. 205-220.

[Hwang 2013] Hwang J, Wood T. Adaptive Performance-Aware Distributed Memory Caching. In：Proc. of Int’l Conf. on Autonomic Computing (ICAC '13). 2013. 33-43.

[IBM 2004] IBM Websphere extreme scale. 2004. http://www.redbooks.ibm.com/ redbooks/SG247683/wwhelp/wwhimpl/js/html/wwhelp.htm

[IBM 2006] IBM white paper. An architectural blueprint for autonomic computing. 2006.

[Ingenthron 2012] Ingenthron M. Growing data sets beyond memory. 2012. http://www.couchbase.org/wiki/display/membase/Growing+Data+Sets+ Beyond+Memory

[Johnson 2007] Johnson RA, Wichern DW. Applied multivariate statistical analysis. 6th ed., New Jersey: Prentice Hall Inc., 2007.

[Jroups 2002] Jroups. 2002. http://www.jgroups.org/overview.html

[JSR-107 2001] JSR-107. 2001. http://jcp.org/en/jsr/detail?id=107

[JSR-342 2011] JSR-342. 2011. http://jcp.org/en/jsr/detail?id=342

[JSR-347 2011] JSR-347. 2011. http://jcp.org/en/jsr/detail?id=347

[Karger 1997] Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In：Proc. of the 29th Annual ACM Symp. on theory of Computing (STOC '97). 1997. 654-663.

[Kari 2011] Kari C, Kim Y, Russell A. Data Migration in Heterogeneous Storage Systems. In：Proc. of the 31st Int’l Conf. on Distributed Computing Systems (ICDCS '11). 2011. 153-160.

[Khan] Khan I. Using read-through & write-through in distributed cache. http://www.alachisoft.com/resources/articles/readthru-writethru-writebehind.html

[Kira 1992] K. Kira and L. Rendell. A Practical Approach to Feature Selection. In：Proc. of the 9th Int’l workshop on Machine learning. 249-256. 1992.

[Kononenko 1994] Kononenko I. Estimating Attributes: Analysis and Extensions of Relief. In：Proc. of 7th European Conf. on Machine Learning (ECML ' 94). 171-182. 1994.

[Kundu 2010] Kundu S, Rangaswami R, Dutta K, Zhao M. Application Performance Modeling in a Virtualized Environment. In：Proc. of IEEE 16th Int’l Symp. on High Performance Computer Architecture (HPCA '10). 2010. 1-10.

[Kunkle 2008] Kunkle D, Schindler J. A load balancing framework for clustered storage systems. In：Proc. of 15th Int’l Conf. on High Performance Computing (HiPC '08). 2008. 57-72.

[Lim 2010] Lim HC, Babu S, Chase JS. Automated control for elastic storage. In：Proc. of the 7th Int’l Conf. on Autonomic computing (ICAC '10). 2010. 1-10.

[Lin 2010] 林海略,韩燕波.多租户应用的性能管理关键问题研究.计算机学报,2010,33(10):1881-1895.

[Liu 1996] Liu H, Setiono R. A probabilistic approach to feature selection - A filter solution. In：Proc. of the 13th Int’l Conf. on Machine Learning (ICML '96). 319-327, 1996.

[Lu 2001] Lu Y, Saxena A, Abdelzaher TF. Differentiated caching services: a control-theoretical approach. In：Proc. of the 21st Int’l Conf. on Distributed Computing Systems (ICDCS '01). 2001. 615-622.

[Lu 2002] Lu C, Alvarez GA, Wilkes J. Aqueduct: online data migration with performance guarantees. In：Proc. of USENIX Conf. on File and Storage Technologies (FAST '02). 2002. 219-230.

[Lu 2004] Lu Y, Abdelzaher TF, Saxena A. Design, Implementation, and Evaluation of Differentiated Caching Services. IEEE Trans. on Parallel and Distributed Systems, 15(5):440-452, 2004.

[Machina 2009] Machina J, Sodan A. Predicting cache needs and cache sensitivity for applications in cloud computing on CMP servers with configurable caches. In：Proc. of the IEEE Int’l Symposium on Parallel & Distributed Processing (IPDPS '09). 2009. 1-8.

[Megiddo 2003] Megiddo N, Modha DS. Arc: A self-tuning, low overhead replacement cache. In：Proc. of the 2nd USENIX Conf. on File and Storage Technologies (FAST '03). 2003. 115-130.

[Mei 2013] Mei Y, Liu L, Pu X, Sivathanu S, Dong X. Performance analysis of network I/O workloads in virtualized data centers. IEEE Trans. on Service Computing, 6(1):48-63, 2013.

[Memcached 2003] Memcached. Wikipedia. 2003.

http://en.wikipedia.org/wiki/ Memcached

[Memcached\_FAQ 2011] Memcached FAQ. 2011.

http://code.google.com/p/ memcached/wiki/NewStart

[Menon 2006] Menon A, Cox AL, Zwaenepoel W. Optimizing Network Virtualization in Xen. In：Proc. of USENIX Annual Technical Conference (USENIX ATC '06). 2006. 15-28.

[Menascé 1999] Menascé DA, Almeida VAF, Fonseca R, Mendes MA. A Methodology for Workload Characterization of E-commerce Sites. In：Proc. of the 1999 ACM Conf. on Electronic Commerce (E-COMMERCE ' 99). 1999.

[Microsoft 2011] Programming model. Windows Server AppFabric Caching. 2011. http://msdn.microsoft.com/en-us/library/hh334298.aspx

[Microsoft 2011\*] Security model. Windows Server AppFabric Caching. 2011. http://msdn.microsoft.com/en-us/library/ff718179.aspx

[Moore 2003] Moore AW. Information Gain. 2003.

http://www.autonlab.org/tutorials/ infogain11.pdf

[Multicollinearity 2005] Multicollinearity. Wikipedia. 2005.

http://en.wikipedia.org/ wiki/Multicollinearity

[Multitenancy 2008] Multitenancy. Wikipedia. 2008.

http://en.wikipedia.org/wiki/ Multitenancy

[NCache 2008] NCache: Caching topologies. 2008.

http://www.alachisoft.com/ ncache/caching-topology.html

[Neter 1985] Neter J, Waserman W, Kutner M. Applied Linear Statistical Models: Regression Analysis of Variance and Experimental Designs, 3rd ed., New York: McGraw Hill, 1985.

[Nishtala 2013] Nishtala R, Fugal H, Grimm S, Kwiatkowski M, Lee H, Li HC, McElroy R, Paleczny M, Peek D, Saab P, Stafford D, Tung T, Venkataramani V. Scaling Memcache at Facebook. In：Proc. of 10th USENIX Symp. on Networked Systems Design and Implementation (NSDI '13). 2013. 385-398.

[Nori 2010] Nori AK. Distributed caching platforms. In: Proc. of the 36th Int’l Conf. on Very Large Data Bases (VLDB 2010). 2010. 1645-1646.

[NoSQL 2009] NoSQL . Wikipedia. 2009. http://en.wikipedia.org/wiki/NoSQL.

[Oracle 2009] Platform-as-a-Service private cloud with oracle fusion middleware. Oracle White Paper, 2009. http://www.oracle.com/us/technologies/ cloud/036500.pdf

[Oracle 2011] Caching data sources. Oracle. 2011. http://download.oracle.com/docs/ cd/E24290\_01/coh.371/e22837/cache\_rtwtwbra.htm#CFHEDIGA

[Patrick 2009] Patrick CM, Garg R, Son SW, Kandemir M. Improving I/O Performance using Soft-QoS Based Dynamic Storage Cache Partitioning. In：Proc. of 11th IEEE Int’l Conf. on Cluster (Cluster '09). 2009. 1-10.

[Peralta 2007] Peralta P. Successfully scaling Java applications in spring. Oracle Corp. 2007. http://www.nejug.org/events/download?f=41

[Pfaffhauser 2010] Pfaffhauser F. Scaling a Cloud Storage System Autonomously [Master Thesis]. ETH. 2010.

[Pierre 1998] Pierre G, Makpangou M. Saperlipopette!:a distributed Web caching systems evaluation tool. In：Proc. of ACM/IFIP/USENIX 1st Int’l Middleware Conference (Middleware '98). 389-405. 1998.

[Pierre 2001] Pierre G, Kuz L, Steen M, Tanenbaum AS. Differentiated Strategies for replicating Web Documents. Computer Communications. 232-240. 2001.

[Pierre 2002] Pierre G, van Steen M, Tanenbaum AS. Dynamically selecting optimal distribution strategies for Web documents. IEEE Trans. on Computers, 51(6):637-651, 2002.

[Prabhakar 2010] Prabhakar R, Srikantaiah S, Kandemir M, Patrick CM, Kandemir M. Adaptive Multi-Level Cache Allocation in Distributed Storage Architectures. In：Proc. of the 24th ACM Int’l Conf. of Supercomputing (ICS '10). 2010. 211-221.

[Prabhakar 2009] Prabhakar R, Srikantaiah S, Patrick CM, Kandemir M. Dynamic Storage Cache Allocation in Multi-Server Architectures. In：Proc. of the ACM/IEEE Int’l Conf. for High Performance Computing, Networking, Storage, and Analysis (SC '09). 2009. 1-12.

[Pu 2010] Pu X, Liu L, Mei Y, Sivathanu S, Koh Y, Pu C. Understanding Performance Interference of I/O Workload in Virtualized Cloud Environment. In：Proc. of the 3rd IEEE Int’l Conf. on Cloud Computing (Cloud '10). 2010. 51-58.

[Pohl 2005] Pohl C. Adaptive caching of distributed components. [PhD Thesis]. TU Dresden, 2005.

[Qin 2011] Qin X, Zhang W, Wang W, Wei J, Zhong H, Huang T. Online Cache Strategy Reconfiguration for Elastic Caching Platform: A Machine Learning Approach. In：Proc. of 35th Annual IEEE Int’l Computer Software and Applications Conference (COMPSAC '11). 2011. 523-534.

[Qin 2013] 秦秀磊, 张文博, 魏峻, 王伟, 钟华, 黄涛. 云计算环境下分布式缓存技术的现状与挑战.软件学报,2013,24(1):50-66.

[Rosa 2011] Rosa L, Rodrigues L, Lopes A. Goal-Oriented Self-management of In-memory Distributed Data Grid Platforms. In Proceedings of the 3rd IEEE Int’l Conf. on Cloud Computing Technology and Science (CloudCom ’11). 2011. 587-591.

[SAE] Sina App Engine (SAE). http://sae.sina.com.cn/

[Salehie 2009] Salehie M, Tahvildari L. Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems (TAAS), 4(2): 42 pages, 2009.

[Schubert 2010] Schubert L, Jeffery K, N-Lutz B. The future of cloud computing: Opportunities for European cloud computing beyond 2010. 2010. http://cordis.europa.eu/fp7/ict/ssai/docs/cloud-report-final.pdf

[Selvakuberan 2008] Selvakuberan K, Indradevi M, Rajaram DR. Combined Feature Selection and classification-A novel approach for the categorization of web pages. Journal of Information and Computing Science, 3(2):83-89, 2008.

[Shen 2007] Shen H, Xu C. Locality-Aware and Churn-Resilient Load Balancing Algorithms in Structured Peer-to-Peer Networks. IEEE Trans. on Parallel and Distributed Systems, 18(6): 849-862, 2007.

[Shue 2012] Shue D, Freedman MJ, Shaikh A. Performance isolation and fairness for multi-tenant cloud storage. In：Proc. of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI '12). 2012. 349-362.

[Sivasubramanian 2003] Sivasubramanian S, Pierre G, Steen MV. A case for dynamic selection of replication and caching strategies. In：Proc. of the 8th workshop on Web Caching and Content Distribution. 2003. 275-282.

[Subramanian 2006] Subramanian R, Smaragdakis Y, Loh GH. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads. In：Proc. of the 39th Annual IEEE/ACM Int’l Symp. on Micro architecture (MICRO '06). pp. 385-396, 2006.

[Terracotta 2009] Terracotta server arrays architecture. 2009. http://64.95.112.233/ documentation/ terracotta-server-array/server-arrays

[Terracotta 2011] Terracotta DSO documentation. 2011.

http://www.terracotta.org/ confluence/display/ docs/Home

[Tesauro 2007] Tesauro G. Reinforcement Learning in Autonomic Computing: A Manifesto and Case Studies. IEEE Internet Computing, 11(1):22-30, 2007.

[Thaler 1998] Thaler D, Ravishankar C. Using name-based mappings to increase hit rates. IEEE/ACM Trans. on Networking, 6(1):1–14, 1998.

[Totok 2006] Totok A, Karamcheti V. Improving Performance of Internet Services through Reward-Driven Request Prioritization. In：Proc. of 14th IEEE Int’l Workshop on Quality of Service (IWQoS '06). 2006. 60-71.

[TPC-W] TPC-W. http://www.tpc.org/tpcw/default.asp.

[Trushkowsky 2011] Trushkowsky B, Bodik P, Fox A. The SCADS Director: Scaling a Distributed Storage System under Stringent Performance Requirements. In：Proc. of USENIX Conf. on File and Storage Technologies (FAST '11). 2011. 163-176.

[Wang 1999] Wang HW. Partial least-squares regression method and applications. Beijing: National Defense Industry Press, 1999.

[Wei 2012] Wei Z, Pierre G, Chi C. Scalable join queries in cloud data stores. In: Proc. of the 12th IEEE/ACM Int’l Symp. on Cluster, Cloud and Grid Computing (CCGrid 2012). 2012. 547-555.

[Weinberger 2007] Weinberger KQ, Tesauro G. Metric Learning for Kernel Regression. Journal of Machine Learning Research, 2007, 2: 612-619.

[WEKA] WEKA. http://www.cs.waikato.ac.nz/ml/weka.

[Whittingham 2006] Whittingham MJ, Stephens PA, Bradbury RB, Freckleton RP. Why do we still use stepwise modeling in ecology and behaviour? Journal of Animal Ecology, 2006, 75(5): 1182–1189.

[Wiggins 2012] Wiggins A, Langston J. Enhancing the Scalability of Memcached. Intel Technical Report. http://download-software.intel.com/sites/default /files/m/0/b/6/1/d/45675-memcached\_05172012.pdf

[Zhang 2011] Zhang W, Wang S, Wang W, Zhong H. Bench4Q: A QoS-Oriented E-Commerce Benchmark. In：Proc. of 35th Annual IEEE Int’l Computer Software and Applications Conference (COMPSAC '11). 2011. 38-47.

[Zhu 2008] Zhu X. Semi-Supervised Learning Literature Survey. University of Wisconsin-Madison, technical report. http://pages.cs.wisc.edu/~jerryzhu/ pub/ssl\_survey.pdf. 2008.

[Zhu 2012] 朱鑫. 面向多租户的分布式缓存系统设计与实现. [硕士论文]. 中国科学院研究生院, 2012.

发表文章目录

1. **Xiulei Qin**, Wei Wang, Wenbo Zhang, Jun Wei, Xin Zhao, Hua Zhong, Tao Huang. *PRESC2*: Efficient Self-Reconfiguration of Cache Strategies for Elastic Caching Platforms. Computing Journal, Springer (SCI indexed, IF: 0.807).
2. **秦秀磊**, 张文博, 魏峻, 王伟, 钟华, 黄涛. 云计算环境下分布式缓存技术的现状与挑战. 软件学报. 2013, 24(1):50-66.
3. **秦秀磊**, 张文博, 王伟, 魏峻, 赵鑫, 钟华, 黄涛. 面向云端key/value存储系统的开销敏感的数据迁移方法. 软件学报. 2013, 24(6):1403-1417.
4. **Xiulei Qin**, Wenbo Zhang, Wei Wang, Jun Wei, Xin Zhao, Tao Huang. Towards a Cost-Aware Data Migration Approach for Key-Value Stores. In Proceeding of the 14th IEEE International Conference on Cluster Computing (*Cluster 2012*), September 24-28, 2012, Beijing. pp.551~556.
5. **Xiulei Qin**, Wenbo Zhang, Wei Wang, Jun Wei, Xin Zhao, Tao Huang. Optimizing Data Migration for Cloud-based Key-Value Stores. In Proceeding of the 21st ACM International Conference on Information and Knowledge Management (*CIKM 2012*), October 29-November 2, 2012, Maui, USA. pp.2204~2208.
6. **Xiulei Qin**, Wei Wang, Wenbo Zhang, Jun Wei, Xin Zhao, Tao Huang. *ElastiCat*: A load rebalancing framework for cloud-based key-value stores. In Proceedings of 19th IEEE International Conference on High Performance Computing (*HiPC 2012*), December 18-21, 2012, Pune, India. pp.1~10.
7. **Xiulei Qin**, Wenbo Zhang, Wei Wang, Jun Wei, Hua Zhong, Tao Huang. A Comparative Evaluation of Cache Strategies for Elastic Caching Platforms. In Proceedings of 11th International Conference on Quality Software (*QSIC 2011*), July 13-14, 2011, Madrid, Spain. pp.166~175.
8. **Xiulei Qin**, Wenbo Zhang, Wei Wang, Jun Wei, Hua Zhong, Tao Huang. Online Cache Strategy Reconfiguration for Elastic Caching Platform: A Machine Learning Approach. In Proceeding of the 35th Annual IEEE International Computer Software and Applications Conference (*COMPSAC 2011*), July 18-22, 2011, Munich, Germany. pp.523~534.
9. **Xiulei Qin**, Jun Wei, Wenbo Zhang, Hua Zhong, Tao Huang. A Two-Phase Approach to Subscription Subsumption Checking for Content-Based Publish/Subscribe Systems. In Proceeding of the 24th IEEE International Conference on Advanced Information Networking and Applications (*AINA 2010*), April 20-23, 2010, Perth, Australia. pp.1278~1285.

参与科研项目目录

1. 国家自然科学基金, "面向多承租的弹性缓存服务关键技术研究" (批准号：61173003)

致 谢

博士五年多的学习生涯是一段既艰苦又令人激动的时光，这段经历令我受益终生。首先要衷心感谢我的导师黄涛研究员，黄老师给我创造了优越的学习与科研环境，使我能够参与到包括自然科学基金、863计划和核高基在内的多项重要的国家级课题中，实践能力和科研能力都得到了很好的锻炼。黄老师对我的论文从选题、研究到写作等各个方面都提出了严格的要求，并给予了悉心的指导和耐心的帮助。黄老师卓越的学术素养，渊博的学识，高尚的道德品格，敏锐深邃的洞察力和思考力，高屋建瓴的战略眼光都让我受益匪浅。“师生一场是缘分”，黄老师2011年对我讲的这句话今天仍言犹在耳，导师对我的影响是深刻而全方位的，体现在做人做事做学问等各个方面。有幸成为黄老师的学生，是我人生中最重要的收获之一。

其次要特别感谢魏峻研究员，在攻读博士学位期间，魏老师对我的研究工作给予了耐心细致的指导，提出了许多富有建设性的宝贵意见和建议，尤其在每个关键节点，都给予了我最无私的支持。魏老师对博士生讨论班的热情投入和高效组织，使我获益良多，同时也是我个人学术成长的一个重要基点；我对如何做科研、如何做好科研有了更加深刻的认识，同时也体味到了沟通对科研的重要意义。魏老师学识渊博，治学态度严谨，具有高度的工作责任感，敏锐的学术洞察力、鉴赏力以及深刻的思考力，幽默健谈，对学生真诚奉献，从魏老师这里，我体会到了永葆激情与梦想的价值所在。

感谢数据网格组组长王伟副研究员，对我而言，他不仅仅是一位项目组长，更是一位待人和善、乐于助人的师兄，一位开朗热情、谦虚正直的朋友，一位有着阅历丰富、随时准备和你侃侃而谈的饭友，一位保持微笑的永恒的合作者。王伟博士心胸宽阔、平易近人、追求品质与完美、具有良好的沟通表达能力、宽阔的知识面、极强的工作责任心以及优秀的组织和管理能力，一直是我和很多同学学习的榜样。

感谢平台组组长张文博副研究员，在平台组的这段日子里，无论是科研工作、项目组织实施还是生活的方方面面，张老师都给予了我本人许多关心和帮助，内心的感动无以言表。张老师敏锐的技术洞察力、认真负责和勤奋忘我的工作态度值得我去学习。

诚挚的感谢软件工程技术中心的其他各位老师。感谢钟华研究员对我工作，学习给予的许多建议和指导，钟老师学识渊博、工作严谨、待学生如朋友、亲切友善令我印象深刻。还要感谢冯玉琳研究员、金蓓弘研究员、叶丹研究员、徐罡副研究员、吴国全副研究员、刘杰、明路、刘玲玲等各位老师，我博士阶段的学习与工作离不开他们的支持与帮助。

特别感谢硕士期间的导师孙波教授，每当科研遇到瓶颈，踌躇不前时，总能得到孙老师最诚挚的鼓励和最坚定有力的支持，这成为我人生中的一笔宝贵的精神财富，支撑我秉持信念，正视困难，勇于迎接新的人生挑战。

感谢数据网格组的全体成员，他们是赵鑫、李萱、纪树平、王彦士、刘朝晖、杨鑫晟、唐震、陈铁南、王晓冉、支孟轩、任凯等。大家平日里的热情大方、勤勉努力、朴实真诚、尽职尽责以及良好的团队合作精神深深的感染着我。也要感谢平台组宋云奎、段智全、周欢云、赵占平、胡振碧、罗涛等诸位同学，感谢他们在科研工作方面给予我的支持和帮助。博士一年级所在项目组马志柔、吴怀林、魏克刚、张更欣和詹孟粮等老师和同学，同样给予了我许多启示，在此一并表示感谢。特别感谢为本文原型系统做出重要贡献的罗嵘、朱鑫两位师弟，大家彼此配合默契，一起见证了从*Idea*到*Impacts*的每一步转变，讨论争执中频频擦出灵感的火花。

感谢陪我一起走过博士生涯的各位同学，他们是陈伟、黄翔、张若定、王焘、高楚舒、白琳、吴恒、伍晓泉、张建华、孙耀、张利锋、徐继伟、窦文生、王卅、许利杰、邵小哲、陈三川、周晓炜、吴东尧、严慧、王帅、伍海江、朱锋、田斐、汪静甜、何海、高强、张晓杰等，感谢他们在科研工作、博士学习及生活中给我的建议和帮助。同时也要感谢并行实验室的室友袁良和颜深根同学，和他们一起度过了5年难忘的寝室岁月。

最后要感谢父母对我多年的抚育之恩以及无私的奉献与关爱。在我攻读学位期间，他们始终是我坚强的后盾和精神上的支柱，不断鞭策鼓励我乐观向上、勇于进取。几年下来，父母每次电话都会关心我的饮食、睡眠和身体情况。谁言寸草心，报得三春晖。在今后的日子里，我唯有用不懈的努力和优异的成绩才能回报父母。

再次感谢所有曾给予我关心和帮助的师长、亲人和朋友，碍于篇幅所限，无法一一表达谢意。

1. [↑](#endnote-ref-1)