DOI:10.7544/issn1000-1239.2017.20170108
54(6):1381-1390,2017

面向固态硬盘的Spark数据持久化方法设计

陆克中[1] 朱金彬[2,4] 李正民[3] 隋秀峰[4,5]
[1](深圳大学计算机与软件学院 广东深圳 518060)
[2](广东工业大学计算机学院 广州 511400)
[3](国家计算机网络应急技术处理协调中心 北京 100029)
[4](计算机体系结构国家重点实验室(中国科学院计算技术研究所) 北京 100190)
[5](中国工程院战略咨询中心 北京 100088)
(kzlu@szu.edu.cn)

Design of RDD Persistence Method in Spark for SSDs

Lu Kezhong[1], Zhu Jinbin[2,4], Li Zhengmin[3], and Sui Xiufeng[4,5]
[1](*College of Computer Science & Software Engineering, Shenzhen University, Shenzhen, Guangdong* 518060)
[2](*School of Computer Science and Technology, Guangdong University of Technology, Guangzhou* 511400)
[3](*National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing* 100029)
[4](*State Key Laboratory of Computer Architecture (Institute of Computing Technology, Chinese Academy of Sciences), Beijing* 100190)
[5](*Strategic Studies Centre, Chinese Academy of Engineering, Beijing* 100088)

Abstract SSD (solid-state drive) and HDD (hard disk drive) hybrid storage system has been widely used in big data computing datacenters. The workloads should be able to persist data of different characteristics to SSD or HDD on demand to improve the overall performance of the system. Spark is an industry-wide efficient data computing framework, especially for the applications with multiple iterations. The reason is that Spark can persist data in memory or hard disk, and persisting data to the hard disk can break the insufficient memory limits on the size of the data set. However, the current Spark implementation does not specifically provide an explicit SSD-oriented persistence interface, although data can be distributed proportionally to different storage mediums based on configuration information, and the user can not specify RDD's persistence locations according to the data characteristics, and thus the lack of relevance and flexibility. This has not only become a bottleneck to further enhance the performance of Spark, but also seriously affected the played performance of hybrid storage system. This paper presents the data persistence strategy for SSD for the first time as we know. We explore the data persistence principle in Spark, and optimize the architecture based on hybrid storage system. Finally, users can specify RDD's storage mediums explicitly and flexibly leveraging the persistence API we provided. Experimental results based on SparkBench shows that the performance can be improved by an average of 14.02%.

Key words big data; hybrid storage; solid-state drive (SSD); Spark; persistence

收稿日期:2017-02-27;**修回日期**:2017-04-14

基金项目:国家"八六三"高技术研究发展计划基金项目(2015AA015305);广东省自然科学基金项目(2014A030313553);广东省省部产学研项目(2013B090500055);深圳市基础研究学科布局项目(JCYJ20150529164656096)
This work was supported by the National High Research and Development Program of China (863 Program) (2015AA015305), the Guangdong Natural Science Foundation of China (2014A030313553), the Guangdong Science and Technology Foundation (2013B090500055), and the Shenzhen Science and Technology Foundation (JCYJ20150529164656096).

通信作者:李正民(lzm@cert.org.cn)

摘 要 基于固态硬盘(solid-state drive, SSD)和硬盘(hard disk drive, HDD)混合存储的数据中心已经成为大数据计算领域的高性能载体,数据中心负载应该可将不同特性的数据按需持久化到SSD或HDD,以提升系统整体性能.Spark是目前产业界广泛使用的高效大数据计算框架,尤其适用于多次迭代计算的应用领域,其原因在于Spark可以将中间数据持久化在内存或硬盘中,且持久化数据到硬盘打破了内存容量不足对数据集规模的限制.然而,当前的Spark实现并未专门提供显式的面向SSD的持久化接口,尽管可根据配置信息将数据按比例分布到不同的存储介质中,但是用户无法根据数据特征按需指定RDD的持久化存储介质,针对性和灵活性不足.这不仅成为进一步提升Spark性能的瓶颈,而且严重影响了混合存储系统性能的发挥.有鉴于此,首次提出面向SSD的数据持久化策略.探索了Spark数据持久化原理,基于混合存储系统优化了Spark的持久化架构,最终通过提供特定的持久化API实现用户可显式、灵活指定RDD的持久化介质.基于SparkBench的实验结果表明,经本方案优化后的Spark与原生版本相比,其性能平均提升14.02%.

关键词 大数据;混合存储;固态硬盘;Spark;持久化

中图法分类号 TP303

“大数据”描述了信息爆炸时代所产生的海量数据,它不仅聚焦于数据规模本身,更强调了数据量激增背景下的数据分析与应用所面临的巨大挑战[1].与传统数据相比,大数据来源广、种类丰富且格式多样,其中囊括了结构化、半结构化和非结构化数据[2].目前,大数据已经渗透到人类社会的各行各业,已成为起决定性作用的生产要素.挖掘隐藏在大数据内部的有价值的信息,可以有效推进相关工作的展开,提高领导者的决策和管理水平[3-4].大数据技术是指从各种类型的大数据中快速提取高价值信息的能力,其中涉及数据的采集、清洗、存储、管理、信息挖掘和可视化等内容.面对海量数据,如何在有效的时间内管理、分析并提取有价值的信息,成为人们亟需解决的问题.然而,无论是规模、种类还是结构,大数据对人们驾驭数据的能力提出了巨大挑战.

Spark是目前高效且在产业界被广泛使用的大数据计算框架,是通用、快速的大规模数据处理引擎[5].1)Spark提供了统一的解决方案,可以用于交互式查询(Spark SQL)、实时流处理(Spark Streaming)、机器学习(Spark MLlib)等复杂任务;2)Spark通过弹性分布式数据集(resilient distributed dataset, RDD)划分阶段和任务,通过高效的有向无环图(directed acyclic graph, DAG)执行引擎优化子任务执行顺序,并通过基于内存的计算大幅提升数据处理效率;3)Spark数据管理依赖于HDFS,Hive等多种数据源,并且集群模式下的Spark实现了横向扩展,支持大规模数据的处理.RDD是Spark区别于其他大数据计算框架最重要的概念,它是一种具有高度容错机制的、只读的分布式数据集.Spark应用程序中,每一个RDD会被分成多个分区,且Spark以分区为单位对RDD进行各种操作[6].

持久化(persist)RDD分区数据到内存或硬盘实现了对计算任务中间结果的缓存,以供后续迭代任务直接读取中间结果,避免了重复计算,大幅提升了数据处理效率[7].另外,持久化数据到硬盘,打破了内存容量不足对数据集规模的限制,使得Spark处理大数据游刃有余.

固态硬盘(solid-state drive, SSD)的出现为提升存储系统性能带来了新的机遇,SSD具有低功耗、低延迟、体积小等优点[8].与传统企业级硬盘(hard disk drive, HDD)通过移动机械臂来寻址方式不同,SSD完全构建于半导体芯片上,因此具有随机访问性能.然而,由于SSD容量成本过高、寿命有限等不足,完全使用SSD替换HDD会大幅提升产业成本.为了合理利用SSD的高性能和HDD的低廉价格等优势,基于SSD和HDD混合存储的异构数据中心得到人们普遍研究和应用[9].当前的研究主要涉及了SSD与HDD的组织架构以及容量比例分配,并针对不同特征的数据提出了合理、高效的数据存储策略.目前,Google,Amazon,Facebook,Baidu等国内外大型互联网公司都已经将SSD应用到数据中心的存储系统中[10-12],通过结合数据冷热条件,实现了数据的按需持久化,提升了系统整体性能.

然而,数据中心计算框架Spark所提供的数据持久化语义对存储介质不具备感知能力,无法实现RDD分区数据的按需持久化.具体而言,在Spark迭代计算任务中,不同RDD的重复利用率往往不同,显然,将不同重复利用率的RDD有选择地持久化到SSD或HDD,充分利用SSD高速读写和HDD大容量的特性,可有效提升Spark性能,而原生Spark提供的单一持久化语义根据Spark配置信息按照比例地放置RDD分区的存储位置,针对性和

灵活性不足,无法实现上述按需持久化目标.因此,基于混合存储系统,为应用的开发者提供面向不同存储介质的 RDD 持久化编程接口,实现用户根据程序中数据的不同特征有针对性地进行数据的持久化成为进一步提升 Spark 性能的关键技术之一.

本文首次基于混合存储系统提出面向 SSD 的数据持久化策略.探索了 Spark 数据持久化原理,并基于 SSD 和 HDD 的混合存储系统优化了 Spark 的持久化架构.设计 DeviceAdapter 模块完成存储设备的映射,并提供显式持久化 API 实现 RDD 的按需持久化.

1 研究背景

1.1 存储异构数据中心的发展

IDC 的研究报告指出:"自 2005 年至 2020 年,人类社会所产生的数据将增长 300 倍——从 130 艾字节增长到 40 000 艾字节"[13].数据的爆炸式增长从根本上改变了传统数据的规模、种类和结构等,催生了新一代数据中心的数据存储和管理方式的变革.尤其对于大数据分析型数据中心,其任务高频率的读、写磁盘操作对数据中心存储系统的性能提出巨大的挑战.SSD 的引入,显著地提升了数据中心的性能和能效.SSD 可以帮助企业在一个快速发展的大数据时代中最大限度地提升数据的存储和分析效率.目前,诸多新兴技术可以有效地提升 SSD 的 I/O 带宽和降低访问延迟.而 HDD 仍然能为那些对存储性能要求较低的数据提供大量的存储效率.IDC 的研究数据显示,大量的数据被数据中心收集并捕获后,并不经常被访问,称之为冷数据,约占全球数据的 90%;而剩余的 10%的数据被收集并捕获后,会经常性地被访问,称之为热数据.显然,将全部的数据都存储在高性能、低延迟的存储设备是不合理的,成本是极为昂贵的.因此,将 SSD 和 HDD 以合理的方式进行组合,通过构建混合存储系统有可能带来性能的大幅提升,同时保障成本可控.

目前,基于 SSD 和 HDD 混合存储的异构数据中心在学术界得到广泛研究,其从组织结构上分为 2 类:1)SSD 作为 HDD 的缓存[14-16];2)SSD 作为 HDD 的同层持久化存储[17-18].在 SSD 作为 HDD 的缓存组织结构中,SSD 负责存放 HDD 少量数据的拷贝,所有的数据请求优先查找 SSD,如果请求的数据在 SSD 中就直接由 SSD 服务,否则从 HDD 上拷贝相应的数据到 SSD 中.这种分层结构,通过部署高命中率的数据映射机制,可以有效地提升存储系统的 I/O 性能.在另一种结构中,SSD 作为 HDD 的同层持久化存储,用户或系统可以根据数据的冷热条件将不同类别的数据按需持久化到 SSD 或 HDD,利用 SSD 高速读、写的特征来提升热数据的访问效率,同时,利用 HDD 大容量的特性来提升数据的存储效率,从而提升系统的整体性能.本文基于该结构展开面向固态硬盘的 Spark 数据持久化方法的研究.在工业界,基于 SSD 和 HDD 混合存储的异构数据中心已经得到了普遍的应用.如 Google、Facebook、百度以及阿里云等国内外大型互联网公司都已经将 SSD 引入数据中心,并结合自身业务需求,合理地构建了基于 SSD 和 HDD 混合存储的异构数据中心.如图 1 所示,三星公司的研究报告指出,基于 SSD 和 HDD 的混合存储系统可以大幅度地降低数据中心能耗至原来的 1/7,同时将数据中心性能提升 2 倍[19].

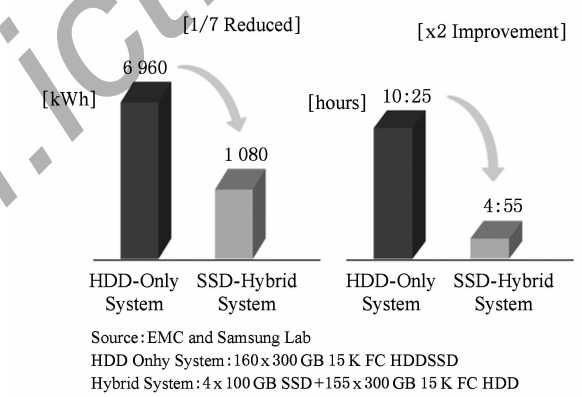

Fig. 1 Performance evaluation between traditional and heterogeneous data centers[19]

图 1 传统与存储异构数据中心性能评估[19]

1.2 混合存储系统中的 Spark 数据持久化问题

Spark 的功能涵盖了大数据计算的各个分支领域,如 SQL 类处理、实时流数据处理、机器学习和图计算等复杂计算任务.Spark 广泛的应用范围、简洁易用的 API,使得 Spark 已经成为越来越受欢迎的大数据计算平台之一.Spark 最重要的一项功能是持久化 RDD 分区数据到内存或硬盘,被持久化的分区数据可被其他迭代任务直接读取,避免了重复计算.

在混合存储系统中,根据数据的冷热条件将不同类别的数据按需持久化到 SSD 或 HDD,可有效提升数据的访问和存储效率.具体到 Spark 应用中,不同 RDD 的重复利用率往往不同,换言之,不同 RDD 的冷热度存在明显差异.因此,在仅考虑存储

层的持久化问题时,如将高热度的 RDD 分区数据完全持久化到 SSD,将热度相对较高或基于其他原因的关键 RDD 分区数据持久化到 HDD,充分发挥混合存储的特性,可有效加速大数据的计算速度. 然而,Spark 所提供的持久化语义对存储介质不具备感知能力,无法实现 RDD 分区数据的按需持久化.

针对该问题,做 3 组实验进行验证:将原生 Spark2.0 部署到仅配置一块 HDD、仅配置一块 SSD 以及同时配置一块 HDD 和一块 SSD 三种服务器平台上,同时在 Spark 配置文件中,每一块存储设备配置一个临时文件目录. 图 2 所示为我们设计的 MicroBench 的 RDD 转换关系,其中 *RDD_a*,*RDD_b* 和 *RDD_e* 的依赖度比较高,所以我们对这 3 个 RDD 做 persist(DISK_ONLY)持久化. 以 RDD 的分区为单位,分别统计 3 种情况下被持久化的数据分布情况.

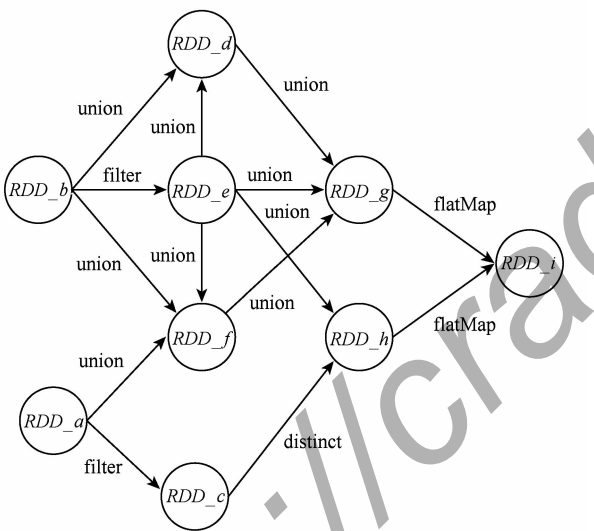

Fig. 2 Transformation of RDD in MicroBench

图 2 MicroBench 的 RDD 转换关系

统计实验结果发现,*RDD_a*,*RDD_b* 和 *RDD_e* 全部被划分为 28 个分区. 第 1 组实验中,3 个 RDD 的分区数据全部被持久化到 HDD;第 2 组实验中,3 个 RDD 的分区数据全部被持久化到 SSD;第 3 组实验中,每个 RDD 的 14 个分区被持久化到 SSD,另一半被持久化到 HDD.

统计结果表明,被持久化的 RDD 分区数据的分布和各实验中临时文件目录的配置存在一定关系,即原生 Spark 仅按照临时文件目录个数比例来确认持久化数据的存储位置. 而在 SSD 和 HDD 混合存储系统中,由于 *RDD_e* 的依赖度较高,希望将 *RDD_e* 完全持久化到 SSD 而 *RDD_a* 和 *RDD_b* 持久化到 HDD,以此提升 Spark 数据处理效率是无法实现的. 因此,原生 Spark 所提供的持久化语义灵活性较差,程序员无法根据 Spark 应用数据的特征,显式地依据 RDD 的冷热特性实现按需持久化. 有鉴于此,本文将进一步探索面向 SSD 的数据持久化方法.

原生 Spark 持久化框架如图 3 所示. 其中,临时文件目录由用户通过配置文件配置,且可以同时配置多个. 临时文件目录的选择决定了数据持久化地址,具体一个 RDD 的某一分区数据持久化地址的选择是由 Utils 模块的 nonNegativeHash 方法完成,该函数的设计原理是保持每一个目录以相同的概率被选取. 如上文所做的第 3 组实验中,当配置了 2 个临时文件目录时,每个目录都有 50%被使用的概率.

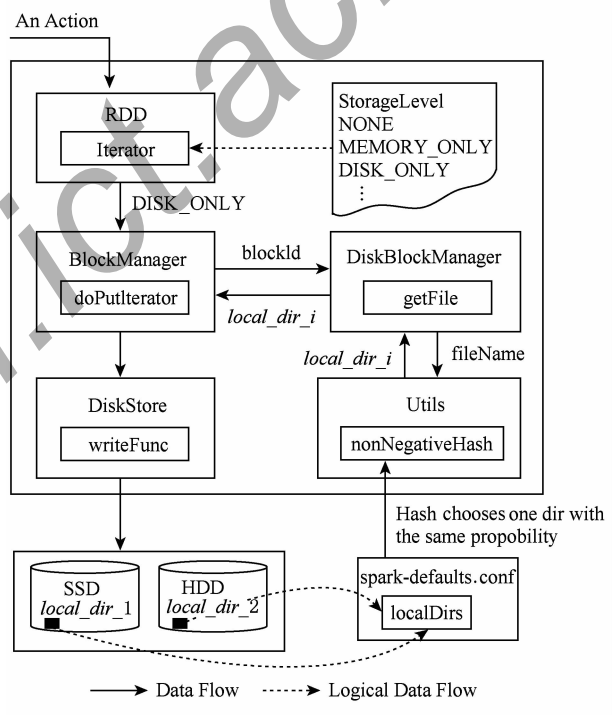

Fig. 3 Persistence framework of native Spark

图 3 原生 Spark 持久化框架

结合图 3 所示 Spark 数据持久化框架可知,Spark 对 SSD 的存在无感知能力的根本原因可归纳为 3 点:

1) Spark 配置文件采用单一参数保存多个临时文件目录,将指向 SSD 和 HDD 的目录进行混合管理;

2) nonNegativeHash 方法未有效地区分不同临时文件目录所在存储介质数据访问性能的差异,等概率的选择目录;

3) 对不同的存储介质,统一使用 DISK_ONLY 为上层应用提供持久化接口,而此接口通过 StorageLevel 反馈给用户.

第 2 节将结合上述原因,提出在混合存储系统中,面向 SSD 的显式数据持久化方案,并以编程接口的方式提供用户根据数据特征按需选择持久化介质.

2 面向 SSD 的持久化

基于 1.2 节分析,研究 Spark 如何正确地感知混合存储系统底层存储设备,提供更为灵活的持久化 API. 首先,将底层存储设备的差异暴露给用户,打破 DISK_ONLY 的屏蔽作用,并向用户提供更为精确的持久化 API,实现 Spark 应用程序的按需持久化,具体实现如下.

如图 4 所示,Spark 持久化架构的具体优化方案如下:

1) 增加"SSD"和"HDD"临时文件目录管理变量,将临时文件目录的混合管理方式改为由"SSD"和"HDD"分别管理指向 SSD 和 HDD 的临时文件目录;

2) 增加设备适配器 DeviceAdapter 模块,接收用户设置的数据持久化级别,同时读取用户配置的临时文件目录,实现持久化级别参数到 SSD 或 HDD 的精确映射;

3) 增加 SSD_ONLY 和 HDD_ONLY 两个持久化级别,将混合存储系统特征暴露给用户. 同时,扩展 StorageLevel 的作用域,如图 3 所示,StorageLevel 仅作用于 BlockManager,为用户和 BlockManager 提供数据持久化级别. 我们将 StorageLevel 作用域进一步延伸至 DeviceAdapter 模块.

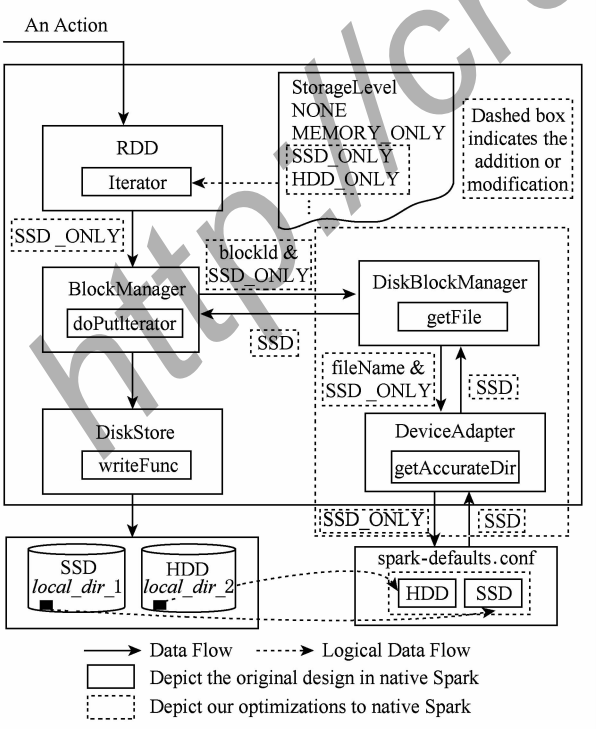

Fig. 4 Persistence framework of optimized Spark

图 4 优化版 Spark 持久化框架

其中,DeviceAdapter 的算法设计如下所述.

算法 1. 固态硬盘/磁盘持久化地址适配算法.

输入:数据持久化级别 *StorageLevel*、用户标记的级别 *storageLevel*、临时文件目录 *SSDLocation* 和 *HDDLocation*;

输出:数据持久化物理地址 *address*.

```
procedure DeviceAdapter
  call BlockManager.getFile;
  /* set SSDLocation and HDDLocation */
  Begin
  switch(storageLevel)
    case SSD_ONLY: /* 匹配固态硬盘 */
      address=SSDLocation;
    case HDD_ONLY: /* 匹配普通磁盘 */
      address=HDDLocation;
  End
End procedure.
```

3 性能评估

在 Spark2.0 版本基础上,依据第 2 节对 Spark 框架做的优化,对 Spark 数据持久化功能模块做了对应的修改,向用户提供了新的持久化 API(SSD_ONLY 和 HDD_ONLY),并进行了重新编译和系统地部署. 本节分别基于如图 2 所示的 MicroBench 和开源的 SparkBench[20](其中的机器学习和图计算 2 类负载),针对原生 Spark2.0 和经本方案优化后的 Spark 进行性能评估实验,并对实验结果做充分地分析.

这里需要对 SparkBench 进行如下修改:将源代码中的 cache() 方法全部替换成 persist() 方法,并且对于机器学习类负载将初始 RDD 数据集按照随机比例进行分割,并且依据此比例将数据持久化到不同的存储介质中;而对于图计算负载,则是针对顶点和边访问特征的不同选择持久化介质. 介质的选择是通过调用 persist(SSD_ONLY)或 persist(DISK_ONLY)接口来实现的.

实验环境如表 1~3 所示,其中表 1 说明了 Spark 集群的搭建和服务器配置信息,表 2 说明了 SSD-HDD 混合存储系统的配置信息,表 3 说明了服务器上的相关软件版本及安装信息.

Table 1 Spark Cluster Configuration

表 1 Spark 集群及服务器配置信息

Parameter	Value
Cluster Mode	Spark on Yarn
Number of Nodes	1
Spark. local. dir	/ssd, /hdd
OS	CentOS 7
CPU	Intel® Xeon® CPU E5645 @2.40 GHz
Cores	48
Memory/GB	32

Table 2 SSD-HDD Heterogeneous Storage Configuration

表 2 SSD-HDD 混合存储系统配置

SSD/HDD	Parameter	Value
SSD	Material	Kingston HyperX Predator PCIe
	Capacity/GB	480
	Piece	1
	I/O/MBps	1 400/1 000
HDD	Material	Seagate ST1000NM001
	Capacity/TB	1
	Piece	1
	I/O/MBps	119/113

Table 3 Related Software Information

表 3 相关软件信息

Software	Edition	Installation
Spark	2.0	~/cluster/spark
Hadoop	2.7.3	~/cluster/Hadoop
JDK	1.7.0	/usr/apache/java
Scala	2.11.8	/usr/apache/scala

我们分别进行 MicroBench 和 SparkBench 实验,验证优化后的 Spark 可以实现 RDD 分区数据的 SSD 或 HDD 精确持久化,并在此基础上通过不同的持久化方案可以有效地提升 Spark 性能. 这里做如下说明,面向固态硬盘的 Spark 数据持久化方法设计,旨在充分利用混合存储系统的大容量、高性能的特性,以提升 Spark 大数据计算效率. 具体地,我们的目标是在最小化 SSD 的使用量的前提下,最大限度地提升 Spark 大数据计算效率,以此减少 SSD 的擦除次数,兼顾了实现绿色数据中心的目的.

3.1 MicroBench 性能评估

本节通过 MicroBench 性能评估实验,验证面向固态硬盘的 Spark 数据持久化方法设计的正确性和灵活性,即实现了面向固态硬盘的 Spark 数据持久化方法. 按图 2 所示设计方案,首先验证优化后 Spark 可以做到 RDD 分区数据按需持久化. 对 *RDD_e* 调用 persist(SSD_ONLY)做持久化,而对 *RDD_a* 和 *RDD_b* 则调用 persist(HDD_ONLY)做持久化,统计结果如表 4 所示. 统计结果显示,优化后 Spark 的持久化结果完全按照程序员所调用的持久化 API 对分区数据进行存储,实现了 Spark 应用程序中间数据面向固态硬盘的持久化,进而提升了 Spark 持久化 API 的灵活性.

Table 4 Partitions Distribution of Different Schemes Using Optimized Spark

表 4 优化版 Spark 不同持久化方案的分区分布

RDD	Partition Distribution
RDD_e	/ssd:28 partitions /hdd:0 partitions
RDD_a	/ssd:0 partitions /hdd:28 partitions
RDD_b	/ssd:0 partitions /hdd:28 partitions

下面分别对优化后的 Spark 和原生 Spark,变化不同 RDD 的持久化介质进行更加深入的实验分析.

图 5 显示的结果为:依次分别针对每一个 RDD 调用 2 种不同的持久化方法,图 5 所示的 MicroBench 的执行时间,其中 persist(DISK_ONLY)和+persist(SSD_ONLY)分别表示原生 Spark 和提供面向 SSD 的持久化接口的优化 Spark. 根据结果可获知,在支持混合存储系统的平台上,相比于默认的按比例持久化方案,显式调用面向 SSD 的接口进行持久化可以获得更好的性能提升,这是因为该接口可将 RDD 的全部分区都存储于 SSD 中,而不是原来的按比例在介质间分配分区. 同时,性能提升的幅度也和 RDD 的访问特征有着密切的关系. 例如,根据图 2, *RDD_e* 的依赖本来较多,但是由于它是利用 filter

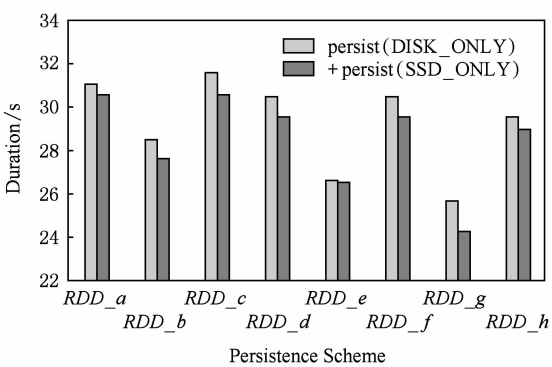

Fig. 5 Performance evaluation of each RDD

图 5 针对每一个 RDD 的持久化方案比较

操作生成的,使得其规模不大,因此即使不对其进行持久化性能影响也并不大,这就是图 5 中持久化 *RDD_e* 的性能收益并不显著的原因.

图 6 是从 RDD 的串并行关系和 RDD 的计算复杂度、RDD 的热度等多个角度考虑不同持久化方案的比较,探索将哪些 RDD 按需持久化到 SSD 或 HDD 可以最大限度地提升系统性能. 综合图 5 和图 6,对 *RDD_*(*c*,*h*)调用本文提供的接口做持久化性能提升得更加明显,其原因在于这些 RDD 的计算更加复杂并且后续使用频繁,将其全部分区都持久化到 SSD 可获得更大的性能收益. 实验结果表明,相对于原生 Spark,本文所提持久化方案可以将性能平均提升 2.7%,最大提升 4.7%.

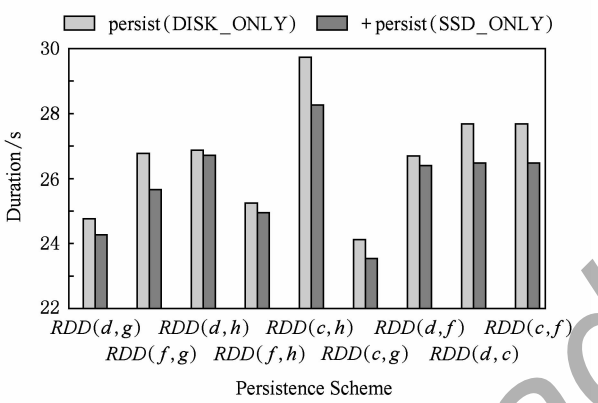

Fig. 6 Performance evaluation of schemes based on parallel and dependency

图 6 基于并行和依赖关系的持久化方案比较

对已经持久化到存储系统层的 RDD,再选择持久化到 SSD 或 HDD,能够进一步获得的性能提升将与生成该 RDD 的计算复杂度以及该 RDD 后续被访问的频繁程度(热度)有很大的关系,不妨将其称之为依赖系数(RDD 的依赖系数为该 RDD 的子 RDD 个数). 由此,将高依赖系数的 RDD 分区数据持久化至 SSD,相比于系统默认的按分区比例的持久化方案将获得更加明显的性能提升. 尽管 MicroBench 实验结果显示,优化后 Spark 性能提升幅度有限,但这是当前使用的数据规模和应用复杂度简单所导致的,显然可以构造出效果更佳明显的 RDD 访问行为.

3.2 SparkBench 性能评估

Spark 作为目前流行的大数据计算框架,其十分擅长多次迭代计算任务(如机器学习、图计算等). Spark 应用程序的迭代次数与 RDD 持久化性能提升具有密切联系,显然,对于迭代次数多的 RDD,将其持久化到 SSD 对 Spark 性能提升的帮助更大. 另外,数据规模是 Spark 读、写硬盘时间消耗的重要影响因素. 本节我们基于 SparkBench 实现了若干机器学习和图计算负载,从数据量和迭代计算角度比较原生 Spark 和优化后 Spark 的性能. 同时强调面向固态硬盘的 Spark 数据持久化方法的设计,旨在通过向 Spark 用户提供显示的面向固态硬盘的持久化接口,提升 Spark 数据持久化接口的灵活性和针对性,进而使得程序员可以根据 Spark 应用程序的数据特征,实现中间数据的按需持久化. 结合 SSD,HDD 的各自特征与 Spark 应用程序数据的冷热特性,将依赖系数较高的 RDD 持久化到 SSD,以提升中间数据重复利用的读取效率,将依赖系数相对较低的 RDD 持久化到 HDD,以减轻 SSD 因频繁读写而引起擦除操作的压力,同时扩大了 Spark 应用程序可利用的存储空间. 而具体 RDD 持久化接口的选择则依赖于程序员对应用程序的个人经验. 如下为 SparkBench 性能评估实验结果的详细分析.

图 7 和图 8 分别测试 KMeans 和 Linear-Regression 负载性能,通过改变数据规模评估原生 Spark 和优化后 Spark,实验结果显示,系统性能平均提升 19.72%,最大提升 20.5%. 同样,针对机器学习算法,我们测试程序迭代次数对 Spark 性能的影响,图 9 和图 10 显示结果为不同训练规模下,原生 Spark 和

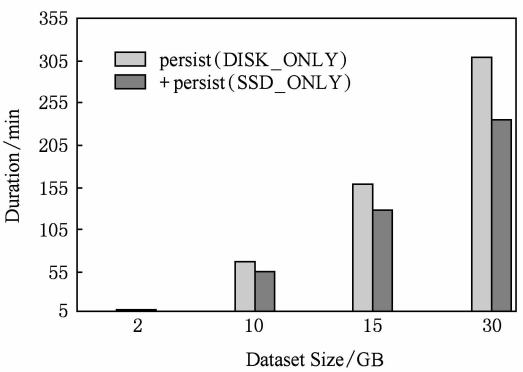

Fig. 7 Performance evaluation of KMeans

图 7 Kmeans 性能评估

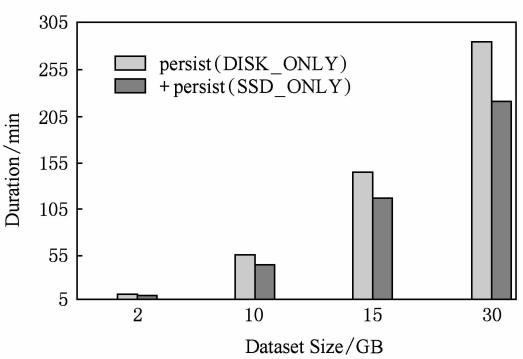

Fig. 8 Performance evaluation of LinearRegression

图 8 LinearRegression 性能评估

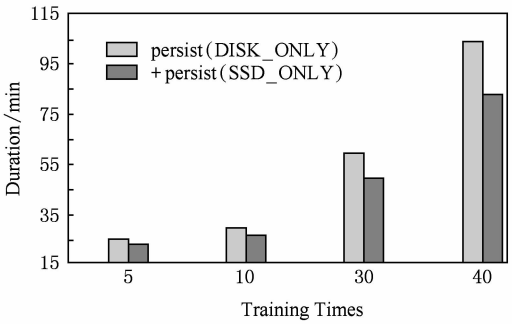

Fig. 9 Performance evaluation of DecisionTree

图 9 DecisionTree 性能评估

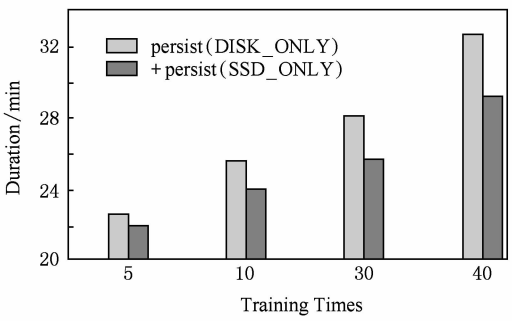

Fig. 10 Performance evaluation of LogisticRegression

图 10 LogisticRegression 性能评估

优化后 Spark 的性能比较,优化后 Spark 性能平均提升 10.04%,最大提升 13.95%.

Spark GraphX 是一个分布式图处理框架,它是基于 Spark 平台提供对图计算和图挖掘简洁易用的而丰富的接口,极大地方便了对分布式图处理的需求. 图 11 和图 12 分别通过测试 SparkBench 的典型图计算负载 PageRank 和 ShortestPath 进行评估优化后 Spark 性能的优越性. 随着图规模的不断扩大,图的结构不断复杂化,即对 Spark 性能要求更高,实验结果表明,优化后 Spark 性能平均提高 11.79%,最大提高 12.62%,且随着图的结构越复杂,优化后 Spark 的性能越显著.

MicroBench 和 SparkBench 性能评估结果说明,我们所提出并实现的按需持久化方案的性能明

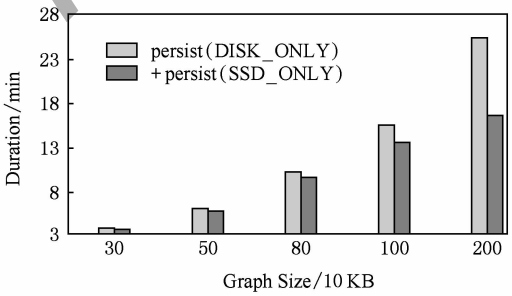

Fig. 11 Performance evaluation of PageRank

图 11 PageRank 性能评估

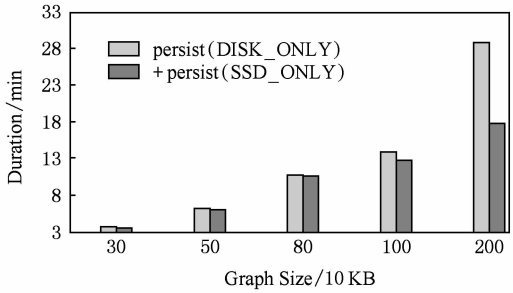

Fig. 12 Performance evaluation of ShortestPath

图 12 ShortestPath 性能评估

显优越于原生 Spark,该方案实现了 Spark 用户依据应用程序特征,对 RDD 进行按需持久化,进一步提升了 Spark 大数据计算性能. 经过分析 RDD 的 DAG 图可以给出一些使用新的编程接口的启发:对于自身计算复杂度较高并且后续使用比较频繁的 RDD,有针对性地将其持久化至 SSD 中,相比于 Spark 原有的处理方式将获得更大的性能提升,当然这需要用户对所开发的应用理解得更加深入,特别是不同 RDD 的访问行为.

4 相关工作

进入“大数据”时代,大数据种类和数据量的激增以及其结构的复杂化给数据分析带来了巨大的挑战. 如何快速、准确地挖掘隐含在大数据内部的高价值信息是目前研究的热门问题,因此,人们研发出 MapReduce,Spark 等大数据计算框架. 目前,国内外关于 Spark 的研究工作层出不穷,较多国内文献利用 Spark 的高性能优势设计了出色的大数据分析算法[21]. 文献[4]介绍了大数据挖掘的若干挑战性问题,文献[22]回顾了近年来有关大数据计算框架、存储及数据分析等问题的发展. 目前较为流行的大数据计算框架有 Hadoop 和 Spark,其中 Spark 基于内存计算且能够持久化计算的中间结果,大幅提升了迭代计算任务的效率. MapReduce 是目前已经非常成熟的大数据并行计算框架,且得到了较为普遍的应用. 文献[23-25]分别研究 MapReduce 应用于快速大规模数据检索、可扩展的云数据匿名化方案和基于 MapReduce 模型实现了关键字感知的服务推荐系统,有效地提升了互联网大数据分析工作的效率. 另外,文献[26]提出并设计了 Mammoth 模型,该模型通过内存调度算法实现内存的全局管理,并设计启发式算法来优化执行单元之间的资源分配,提高了 MapReduce 性能.

文献[27-29]中,作者分别从不同的角度出发研究如何提升 Spark 性能.其中,文献[28]提出自适应调整策略,Spark 在 JVM 的基础上对内存做了进一步的管理,作者提出动态地调整 Spark 内存分配方案,进一步提升 Spark 数据处理效率.检查点是 Spark 实现 RDD 容错的一个重要概念,通过将某 RDD 设置为检查点为快速恢复 RDD 提供了保障.文献[29]提出了自动检查点设置算法,以提升 RDD 的容错性能、改进 Spark 迭代计算任务的效率,从而实现提升 Spark 性能的目的.作者在文献[27]中分析了 Spark 大数据处理过程中 Shuffle 的产生及其对 Spark 性能造成的影响,并介绍了基于排序的 shuffle 可以提升 Spark 性能并应用到 Spark1.1.0 版本中.伴随着人工智能产业的发展,机器学习成为当今学术界和产业界共同的热门话题,且基于机器学习的大数据分析技术得到快速发展.由于 Spark 是基于内存的计算框架,且实现了计算中间结果的持久化功能,使得其在诸如机器学习、图计算等多次迭代计算领域表现出色.作者在文献[30]中介绍了 Spark 并行分布式机器学习库,该库基于数据和模型并行策略,实现对数据和模型存储和相关操作.文献[31]使用 Spark 更高效地利用大数据实现优秀的推荐系统引擎.

目前,关于 Spark 性能提升研究集中于 Spark 内存管理,通过调节 JVM 内存利用率提升 Spark 性能.然而,内存空间的不足限制了 Spark 通过内存加速超大规模数据的计算,而持久化数据到硬盘打破了内存容量不足对数据集规模的限制,使得 Spark 处理大数据游刃有余.本文首次基于混合存储系统提出 Spark 性能优化方案.通过优化 Spark 的持久化框架,实现 Spark 数据的按需持久化.进而,Spark 可以根据程序特征,将高热度 RDD 的分区数据持久化到 SSD,利用存储异构数据中心的特征有效地提升 Spark 性能.

5 总 结

目前 Spark 无法精确持久化 RDD 分区数据到 SSD 或 HDD,无法根据应用程序特征进行按需持久化,导致 Spark 无法充分利用混合存储系统的弹性、高性能的优势以提升自身性能.针对该问题,本文探索了 Spark 数据持久化原理,进而针对混合存储系统对 Spark 的持久化框架进行优化,并向用户提供了 SSD_ONLY 和 HDD_ONLY 持久化 API,实现了 Spark 的显示 SSD 和 HDD 持久化功能.本文通过分析 HDD 和 SSD 读取性能并结合 RDD 读取特征,理论论证了将 RDD 分区数据持久化到 SSD 或 HDD 以及混合存储系统的性能差异.然后进一步通过基准性能评估实验验证了优化后 Spark 对应用程序性能提升明显优于原生 Spark.未来将进一步探索面向异构存储系统的隐式持久化方案,即数据持久化介质的确定由 Spark 框架自动完成,对用户是透明的.

参 考 文 献

[1] Labrinidis A, Jagadish H V. Challenges and opportunities with big data [J]. Proceedings of the VLDB Endowment, 2012, 5(12): 2032-2033

[2] Howe D, Costanzo M, Fey P, et al. Big data: The future of biocuration [J]. Nature, 2008, 455(7209): 47-50

[3] Kriegel H P, Borgwardt K M, Kröger P, et al. Future trends in data mining [J]. Data Mining and Knowledge Discovery, 2007, 15(1): 87-97

[4] Wu Xindong, Zhu Xingquan, Wu Gongqing, et al. Data mining with big data [J]. IEEE Trans on Knowledge and Data Engineering, 2014, 26(1): 97-107

[5] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster computing with working sets [C/OL] //Proc of the 2nd USENIX Conf on Hot Topics in Cloud Computing (HotCloud). Berkeley, CA: USENIX Association, 2012 [2017-01-18]. http://static.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf

[6] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing [C] //Proc of the 9th USENIX Conf on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2012: 15-28

[7] Jiang Zhipeng, Chen Haopeng, Zhou Huan, et al. An elastic data persisting solution with high performance for Spark [C] //Proc of the 2015 Int Conf on Smart City/SocialCom/SustainCom. Piscataway, NJ: IEEE, 2015: 656-661

[8] Rizvi S S, Chung T S. Flash SSD vs HDD: High performance oriented modern embedded and multimedia storage systems [C] //Proc of the 2nd Int Conf on Computer Engineering and Technology (ICCET). Piscataway, NJ: IEEE, 2010: 297-299

[9] Narayanan I, Wang Di, Jeon M, et al. SSD failures in datacenters: What, when and why? [C] //Proc of the 9th ACM SIGMETRICS Int Conf. New York: ACM, 2016: 407-408

[10] Meza J, Wu Qiang, Kumar S, et al. A large-scale study of flash memory failures in the field [C] //Proc of ACM Sigmetrics Performance Evaluation Review. New York: ACM, 2015: 177-190

[11] Ouyang Jian, Lin Shiding, Hou Zhenyu, et al. Active SSD design for energy-efficiency improvement of Web-scale data analysis [C] //Proc of the 2013 Int Symp on Low Power Electronics and Design. Piscataway, NJ: IEEE, 2013: 286-291

[12] Schroeder B, Lagisetty R, Merchant A. Flash reliability in production: The expected and the unexpected [C] //Proc of the 14th USENIX Conf on File and Storage Technologies (FAST'16). Berkeley, CA: USENIX Association, 2016: 67-80

[13] Gantz J, Reinsel D. Extracting value from chaos, IDC IVIEW [R/OL]. Hopkinton, MA: EMC Corporation, 2011 [2017-01-18]. https://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf

[14] Canim M, Mihaila G A, Bhattacharjee B, et al. SSD bufferpool extensions for database systems [J]. Proceedings of the VLDB Endowment, 2010, 3(1/2): 1435-1446

[15] Kang W H, Lee S W, Moon B. Flash-based extended cache for higher throughput and faster recovery [J]. Proceedings of the VLDB Endowment, 2012, 5(11): 1615-1626

[16] Ni Yuanjiang, Jiang Ji, Jiang Dejun, et al. S-RAC: SSD friendly caching for data center workloads [C] //Proc of the 9th ACM Int on Systems and Storage Conf. New York: ACM, 2016: 8:1-8:12

[17] Awasthi A, Nandini A, Bhattacharya A, et al. Hybrid HBase: Leveraging flash SSDs to improve cost per throughput of HBase [C] //Proc of the 18th Int Conf on Management of Data. Mumbai, India: Computer Society of India, 2012: 68-79

[18] Luo Tian, Lee R, Mesnier M, et al. hStorage-DB: Heterogeneity-aware data management to exploit the full capability of hybrid storage systems [J]. Proceedings of the VLDB Endowment, 2012, 5(10): 1076-1087

[19] SAMSUNG. SAMSUNG GREEN SSD [R/OL]. Austin, Texas: SAMSUNG SEMICONDUCTOR, INC., 2010 [2017-01-18]. http://www.samsung.com/us/business/oem-solutions/pdfs/SSI-green_ssd_v4.pdf

[20] Li Min, Tan Jian, Wang Yandong, et al. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform Spark [C] //Proc of the 12th ACM Int Conf on Computing Frontiers. New York: ACM, 2015: No. 53

[21] Zhu Jizhao, Jia Yantao, Xu Jun, et al. SparkCRF: A parallel implementation of CRFs algorithm with Spark [J]. Journal of Computer Research and Development, 2016, 53(8): 1819-1828 (in Chinese)
(朱继召, 贾岩涛, 徐君, 等. SparkCRF: 一种基于Spark的并行CRFs算法实现[J]. 计算机研究与发展, 2016, 53(8): 1819-1828)

[22] Bilal M, Oyedele L O, Qadir J, et al. Big data in the construction industry: A review of present status, opportunities, and future trends [J]. Advanced Engineering Informatics, 2016, 30(3): 500-521

[23] Doulkeridis C, Nørvåg K. A survey of large-scale analytical query processing in MapReduce [J]. The VLDB Journal, 2014, 23(3): 355-380

[24] Meng Shunmei, Dou Wanchun, Zhang Xuyun, et al. KASR: A keyword-aware service recommendation method on MapReduce for big data applications [J]. IEEE Trans on Parallel and Distributed Systems, 2014, 25(12): 3221-3231

[25] Zhang Xuyun, Yang L T, Liu Chang, et al. A scalable two-phase top-down specialization approach for data anonymization using MapReduce on cloud [J]. IEEE Trans on Parallel and Distributed Systems, 2014, 25(2): 363-373

[26] Shi Xuanhua, Chen Ming, He Ligang, et al. Mammoth: Gearing Hadoop towards memory-intensive MapReduce applications [J]. IEEE Trans on Parallel and Distributed Systems, 2015, 26(8): 2300-2315

[27] Rana N, Deshmukh S. Performance improvement in Apache Spark through shuffling [J]. International Journal of Science, Engineering and Technology Research, 2015, 4(3): 1636-1638

[28] Zhao Yao, Hu Fei, Chen Haopeng. An adaptive tuning strategy on Spark based on in-memory computation characteristics [C] //Proc of the 18th IEEE Int Conf on Advanced Communication Technology (ICACT). Piscataway, NJ: IEEE, 2016: 484-488

[29] Zhu Wei, Chen Haopeng, Hu Fei. ASC: Improving Spark driver performance with automatic Spark checkpoint [C] //Proc of the 18th IEEE Int Conf on Advanced Communication Technology (ICACT). Piscataway, NJ: IEEE, 2016: 607-611

[30] Meng Xiangrui, Bradley J, Yavuz B, et al. Mllib: Machine learning in Apache Spark [J]. Journal of Machine Learning Research, 2016, 17(1): 1235-1241

[31] Kulkarni S. A recommendation engine using Apache Spark [D]. San Jose, CA: The Faculty of the Department of Computer Science San Jose State University, 2015

Lu Kezhong, born in 1982. PhD, professor at Shenzhen University. His main research interests include big data theory, parallel and distributed computing, wireless sensor networks, design and analysis of algorithms and computational geometry.

Zhu Jinbin, born in 1989. Master candidate. His main research interests include computer system architecture and design and analysis of algorithms.

Li Zhengmin, born in 1984. PhD candidate. His main research interests include cloud computing, network and network virtualization.

Sui Xiufeng, born in 1982. PhD, associate professor of the Institute of Computing Technology, Chinese Academy of Sciences. His main research interests include high performance computer archi-tectures, system performance modeling and evaluation, and cloud computing (suixiufeng@ict.ac.cn).