计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2017.20151062
54(5):1097-1108,2017

# 一种正交分解大数据处理系统设计方法及实现

向小佳[1] 赵晓芳[1] 刘 洋[1] 龚关俊[1] 张 晗[2]
[1](中国科学院计算技术研究所 北京 100190)
[2](北方工业大学计算机学院 北京 100144)
(xiangxiaojia@ncic.ac.cn)

# An Orthogonal Decomposition Based Design Method and Implementation for Big Data Processing System

Xiang Xiaojia[1], Zhao Xiaofang[1], Liu Yang[1], Gong Guanjun[1], and Zhang Han[2]
[1](*Institute of Computing Technology, Chinese Academy of Science, Beijing* 100190)
[2](*School of Computer Science, North China University of Technology, Beijing* 100144)

**Abstract** Big data stimulates a revolution in data storage and processing field, resulting in the thriving of big data processing systems, such as Hadoop, Spark, etc, which build a brand new platform with platform independence, high throughput, and good scalability. On the other hand, substrate platform underpinning these systems are ignored because their designation and optimization mainly focus on the processing model and related frameworks & algorithms. We here present a new loose coupled, platform dependent big data processing system designation & optimization method which can exploit the power of underpinning platform, including OS and hardware, and get more benefit from these local infrastructures. Furthermore, based on local OS and hardware, two strategies, that is, lock-free based storage and super optimization based data processing execution engine, are proposed. Directed by the aforementioned methods and strategies, we present Arion, a modified version of vanilla Hadoop, which show us a new promising way for Hadoop optimization, meanwhile keeping its high scalability and upper layer platform independence. Our experiments prove that the prototype Arion can accelerate big data processing jobs up to 7.7%.

**Key words** big data processing system; computing framework; localization; lock free; super optimization; excecution engine

**摘 要** MapReduce等计算框架的出现开启了大数据处理新纪元,以Hadoop,Spark为代表的大数据处理系统具有大吞吐率、跨平台、高可扩展的优势,并得到广泛应用.然而,为避免与具体的操作系统、硬件平台绑定,这些系统的设计与优化集中在计算模型、调度算法等方面,无法充分利用底层平台的优势.提出了一种基于正交分解的大数据处理系统设计与优化方法,将系统分解为松耦合的多个功能正交的模块,使存储、处理功能分离出来,交给能够利用底层平台操作系统甚至硬件资源的存储、执行引擎,原大数据系统退化为调度平台;进而,提出基于锁无关机制的存储底层优化策略和基于指令超级优化的执行引擎底层优化策略.以此为指导,以Hadoop作为兼容和改进的对象,实现了原型大数据处理系统

**收稿日期**:2015-12-09;**修回日期**:2016-07-19
**基金项目**:国家自然科学基金项目(61202061,61202413);中国科学院计算技术研究所创新课题项目(20146080)
This work was supported by the National Natural Science Foundation of China (61202061, 61202413) and the Innovation Program of Institute of Computing Technology, Chinese Academy of Sciences (20146080).

Arion. Arion 既能保持 Hadoop 的跨平台、高可扩展的优势,又能消除任务执行的瓶颈,其本地化的设计与优化手段对非 Hadoop 平台同样有效. 通过在原型系统上的实验证明,Arion 能够提升大数据处理任务的执行效率,最高达 7.7%.

**关键词** 大数据处理系统;计算框架;本地化;锁无关;超级优化;执行引擎

**中图法分类号** TP391

网络大数据的复杂性、不确定性、涌现性[1]给当前 IT 系统的架构、计算能力带来了挑战和机遇,催生了大数据处理框架. 围绕着这些计算框架,诞生了各种大数据处理系统. 例如用于批量大数据处理的 Google GFS 与 MapReduce[2],Nokia 的 Disco[3]、面向流式处理的 Google Dremel[4], Microsoft 的 Dryad[5],Twitter 的 Storm[6],Yahoo 的 S4[7]等,学术界和开源社区也围绕着面向批量大数据处理的 Apache Hadoop、基于 Hadoop 的更具实时性的 Impala①、伯克利 AMP Lab 的基于 RDD[8]的、面向工作集迭代应用的 Spark[9]展开了深入研究,国内的互联网巨头百度、阿里、腾讯等也在 Hadoop 等系统上部署了应用.

各类系统面向不同的应用,设计有针对性的计算模型、调度算法、数据结构,从而不断演进. 如 Dremel,Storm 等流式处理模型,相对于 Hadoop,更适合海量流数据的即时查询;而 Spark 则针对 MapReduce 模型不擅长的迭代处理和交互应用,提出了 RDD 内存数据集及相关迭代模型;Hadoop 自身的计算框架由原本单一的 MapReduce 演化出了基于有向无环图(directed acyclic graph, DAG)的更为灵活的 Tez②;Hadoop 自身的调度系统也从单一的全局任务调度发展到了新一代的 Yarn[10],分离了 JobTracker 的资源管理与任务调度功能.

然而,由于大数据处理系统规模大,强调平台无关性,避免与具体的操作系统、硬件平台挂钩,上述系统的演进都忽视了对底层平台技术的利用. Intel 中国研究院的 NativeTask[11]通过设计外挂的计算引擎模块,将部分 Hadoop 计算引擎内部的计算外延到 Java 虚拟机之外,取得了一定的本地化效果,思想值得借鉴,但还未充分发挥存储结点、计算结点本地操作系统、硬件平台的潜力. 国内的百度公司也提出了 Hadoop 的 C++扩展③,通过使用类似 Pipe 的协议将 Map 和 Reduce 两阶段的 Java 执行逻辑替换为 C++编写并预编译好的二进制可执行文件,向本地化迈进了一步,但其失去了中间逻辑表示的灵活性,同时本地化仅限于 Map 和 Reduce 的用户逻辑,没有考虑通用中间处理环节的本地化,也没有深度挖掘代码的优化空间.

我们首先提出一种基于正交分解的大数据处理系统设计与优化方法,主张大数据处理系统的设计应采用松耦合架构,能明确区分出功能正交的调度管理、数据存储、任务执行三大功能模块;与任务逻辑紧相关的数据存储与任务执行应下沉到具体的硬件平台去完成,大数据系统只负责最核心的资源与任务调度. 进而,在此方法的指导下,我们提出了基于锁无关机制的存储底层优化策略,以及基于二进制指令超级优化的执行引擎底层优化策略. 前者利用存储结点操作系统的原语,构造锁无关数据结构(lock free data structure)作为构造共享存储的基石;后者基于大数据系统的应用针对性,采用超级优化(super optimization)方法,在执行逻辑中间表达或二进制代码层面作离线静态分析,利用编译技术来发挥底层平台潜力.

基于上述技术,以 Hadoop 作为兼容和改进的对象,我们实现了原型大数据处理系统 Arion. Arion 采用了松耦合的方式,将 Hadoop 的任务调度与存储、计算引擎拆分开,通过通信协议连接;数据存储下沉到 ArionFS,这是一个采用 C 语言实现的大数据分布式文件系统,其中实现了基于锁无关(lock free, LF)结构的元数据存储,以及基于状态机的文件数据锁无关共享读写协议;任务执行则下沉到基于 LLVM[12]的本地化执行引擎,能够在代码中间表示(IR)层面或二进制代码层面采用类似窥孔(peep-hole)这样的超级优化方法,提升执行引擎效率.

① Cloudera Impala. https://github.com/cloudera/impala

② Apache Tez. http://incubator.apache.org/projects/tez.html

③ Hadoop C++ Extension: A Framework for Making MapReduce More Stable and Faster. http://wenku.baidu.com/view/1859f988d0d233d4b14e69c8.html

## 1 方法与设计

### 1.1 基于正交分解的大数据处理系统设计与优化方法

基于正交分解的设计优化方法原则有4点:

1) 大数据处理系统的设计应能明确区分出调度管理、数据存储、任务执行三大正交功能模块;

2) 各功能模块松耦合,由类RPC通信协议互联;

3) 大数据系统只负责最核心的资源与任务管理,工作于中间层,与平台、语言无关;

4) 与任务逻辑紧相关的两大功能模块,数据存储与任务执行,应外包到具体的硬件平台去完成,且应平台相关,以期充分发挥平台潜力.

平台相关是指利用物理节点的操作系统、硬件机制来设计与优化,与平台底层软硬件紧耦合.优点是能够充分利用平台资源,挖掘其潜力;缺点是某一平台的优化成果,如优化二进制代码,不能无缝迁移到其他平台;但这并不影响整个系统的平台无关性与可扩展性,这是由第3点原则保证的.

进而,我们提出了2条平台相关优化设计策略:

1) 基于锁无关机制的存储底层优化策略.利用存储结点操作系统的原语,构造锁无关数据结构作为元数据共享存储的基石;进而可将锁无关机制扩展到文件数据的访问流程中.

2) 基于指令超级优化的执行引擎底层优化策略.大数据系统有应用针对性,平台执行引擎中的计算也具备一定的模式,有规律可寻,因此可采用超级优化的方法,在执行逻辑中间表示或二进制硬件指令层面作离线静态分析,提取优化指令序列,创建优化数据库,采用一次写入多次读取(write once read multi-time, WORM)模式,利用编译技术来挖掘底层平台的潜力.

### 1.2 Arion整体架构设计

Arion以Hadoop作为兼容和改进的对象,将数据存储、任务执行与调度管理分离;以Hadoop pipe[①]为基础,使用C语言扩充了一套功能模块间的通信协议;任务的调度管理工作于中间层,采用LLVM的中间表示语言IR作为任务执行逻辑的载体,与具体底层平台、语言无关;数据存储下沉到全C语言实现的分布式文件系统ArionFS,支持锁无关元数据与数据访问;任务执行则下沉到基于LLVM的本地化执行引擎,能够在代码中间表示(IR)层面或二进制代码层面采用超级优化方法,提升执行引擎效率.

图1为Arion的整体框架图.一个高级语言(例如C++)书写的MapReduce大数据处理程序的处理过程如下:1)该程序作为输入,通过编译前端(例如Clang[②])编译后转变为中间表达IR文件;接下来,中间表达被提交给大数据处理框架,例如Hadoop,Hadoop会根据框架的定义生成大数据处理任务,交给JobTracker调度,具体工作派发给任务执行代理Task Tracker,而IR文件则通过Hadoop上传到大数据存储文件系统ArionFS中;2)任务执行代理通过类RPC的通讯协议ArionPipe与本地化执行引擎通信,后者会根据指令从ArionFS中获取IR文件,从预编译的函数库中抽取本次处理会使用的连接件(见第2节),并将它们组装起来;3)LLVM本地执行引擎执行上述组装好的任务,起始阶段会通过连接件reader从ArionFS读取需要处理的数据子集Split,结束阶段通过writer向ArionFS回写处理结果.

图1中最上部虚线框中部分为退化Hadoop,主要负责调度管理,数据存储与任务执行的功能被拆分出去;外部存储通过加载ArionFS的函数库接入,通信协议为ArionFS的基于状态机的锁无关共享读写协议(见2.3.2节);外部任务的控制与状态反馈则由Task Tracker任务执行代理接入,通信采用ArionPipe协议.

图1的中部为基于LLVM的任务执行引擎及其适配层.执行引擎中,SuperOptimization指令级加速引擎依赖平台技术和编译超级优化技术,挖掘优化代码片段,提升引擎效率;Pass Manager为LLVM的编译阶段管理结构;即时编译(just in time, JIT)引擎为代码执行的核心.适配层为针对框架(Hadoop)所编写的便于大数据处理工作流无缝衔接的通用代码库,一般会被预编译成JIT引擎可以执行的二进制代码,或易于发布的IR形式,并于任务执行时按需动态加载.这些框架相关的大数据处理通用代码在Arion中按功能封装为类,后文称之为连接件,典型连接件如用于Map和Reduce阶段衔接的通用Partitioner.

存储如图1的底层,即数据管理层.该层为全C

① Hadoop Pipes. http://www-01.ibm.com/support/knowledgecenter/SSGSMK_6.1.1/mapreduce_integration/map_reduce_hadoop_pipes.dita

② Clang: A C Language family frontend for LLVM. http://clang.llvm.org

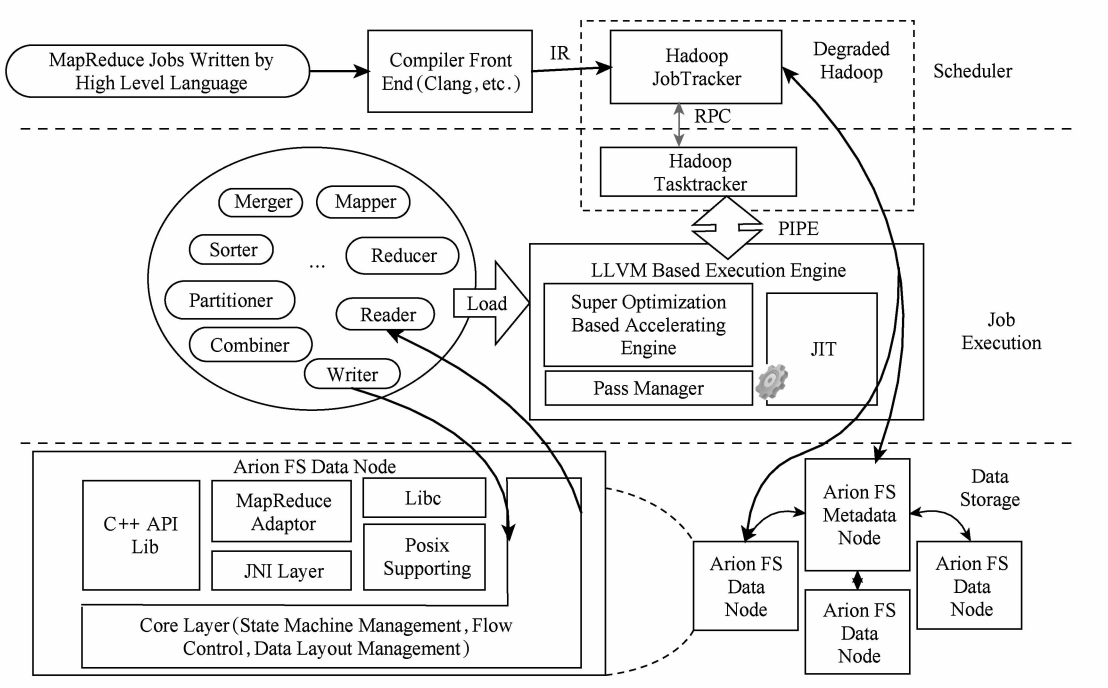


Fig. 1 Arion architecture

图 1 Arion 框架图

语言实现的锁无关大数据文件系统 Arion FS. 采用锁无关机制可以有效解决数据访问过程中的冲突,提高并发性. 锁无关机制包括基于 OS 原语的锁无关元数据和基于状态机的数据锁无关读写协议两部分.

数据管理层左部为 ArionFS 软件模块堆栈,底部为核心层,上部为接口层. 核心层封装了文件系统的名字空间布局管理(layout)、流管理、状态机管理等功能,能够直接调用 OS 本地系统调用,Libc 函数库;接口层高度模块化,支持标准的 Posix 文件访问协议及其上的 Libc 函数库,同时提供二次开发用 C/C++ API 和 Java API,其中 Posix 相关模块在内核实现.

综上,Arion 采用面向平台的本地化设计方法,对 Hadoop 做了正交化分解,形成了专职调度管理的退化版大数据框架、基于 LLVM 的任务执行引擎以及大数据存储 ArionFS 的三大松耦合模块架构.

### 1.3 数据存储底层优化

分布式文件系统作为共享大数据的载体,其上的数据访问具有高并发的特点. 为了利用底层优化技术来应对海量高并发的负载,我们采用全 C 语言实现了一个原型大数据分布式文件系统 ArionFS,其特点是:1)C 语言实现,能够直接利用操作系统的底层函数库和资源. 2)充分利用 Lock Free 结构. 文件系统的元数据存储采用 Lock Free 的 Hash Table,能够直接利用操作系统原语和数据结构设计来解决热点元数据的并发访问冲突,避免死锁等问题;文件系统的海量数据访问采用基于状态机的锁无关读写协议,解决并发访问冲突,提高并发性.

ArionFS 不支持类似 HDFS 的多副本流水线技术,其高可靠机制目前在卷、块设备这一层级实现.

文件系统的元数据存储采用了锁无关 Hash 表(lock free Hash table, LFHT),支持对元数据条目查询、插入、删除,这里的元数据主要是 DENTRY 条目(后文简称条目),用来存储文件名与具体数据分布等属性的映射关系. 相对于基于锁的并发元数据访问,实现锁无关 Hash 表有 3 个特点:

1) 碰撞 Hash 值的条目置于同一 Hash 桶中,Hash 桶组织为链表结构,基于 CAS(比较与交换)原语来解决读写冲突;

2) 基于 Pin 来标识条目的所有权,支持批量删除(Pin 是一种所有权声明工具,用来声明对资源的占有,控制对资源的回收,以免出现一致性问题);

3) 不同 Hash 桶采用动态分配队列(dynamic allocated array)来组织,访问文件条目速度为 LogN,且空间动态分配删除,节省存储资源.

文件系统的数据访问采用了基于状态机的锁无关读写协议,取消了共享文件并发访问时的锁操作(file lock). 数据的锁无关读写是基于 CAS 的锁无关算法在状态机层面的扩展. ArionFS 中,客户端与服务器端都有各自的运行时状态机. 每个状态代表

一个执行阶段,伴随着一段文件访问的执行逻辑,逻辑执行的结果是事件,事件会触发状态机跳向下一状态或者下一个子状态机,循环往复.这里的状态机可以类比于 CPU,每个状态所封装的执行逻辑类比于指令,因此,对 ArionFS 的客户端和服务器端而言,状态所封装的执行逻辑是原子的,这是在状态机层面实现类 CAS 原语的前提.

锁无关读写流程如图 2 所示,图 2 中通信的三方包括分布式文件系统的元数据服务器、客户端、数据服务器.读写由客户端发起,执行过程如左部的客户端 IO 子状态机所示:①客户端首先进入初始化状态;②根据要访问的文件名或 ID 向元数据服务器发消息以获得相应文件的分布等信息;③检查文件属性、访问权限;④初始化代表访问请求的消息;⑤通过专用信道向数据服务器发送代表访问请求的消息,并等待回复;若回复为数据访问成功,则接收返回数据或结果,并提交给下一状态;否则根据一定策略,选择重试或者放弃;⑥检查返回结果;⑦如状态机执行过程中发生故障,检查 IO 错误,处理并退出子状态机.

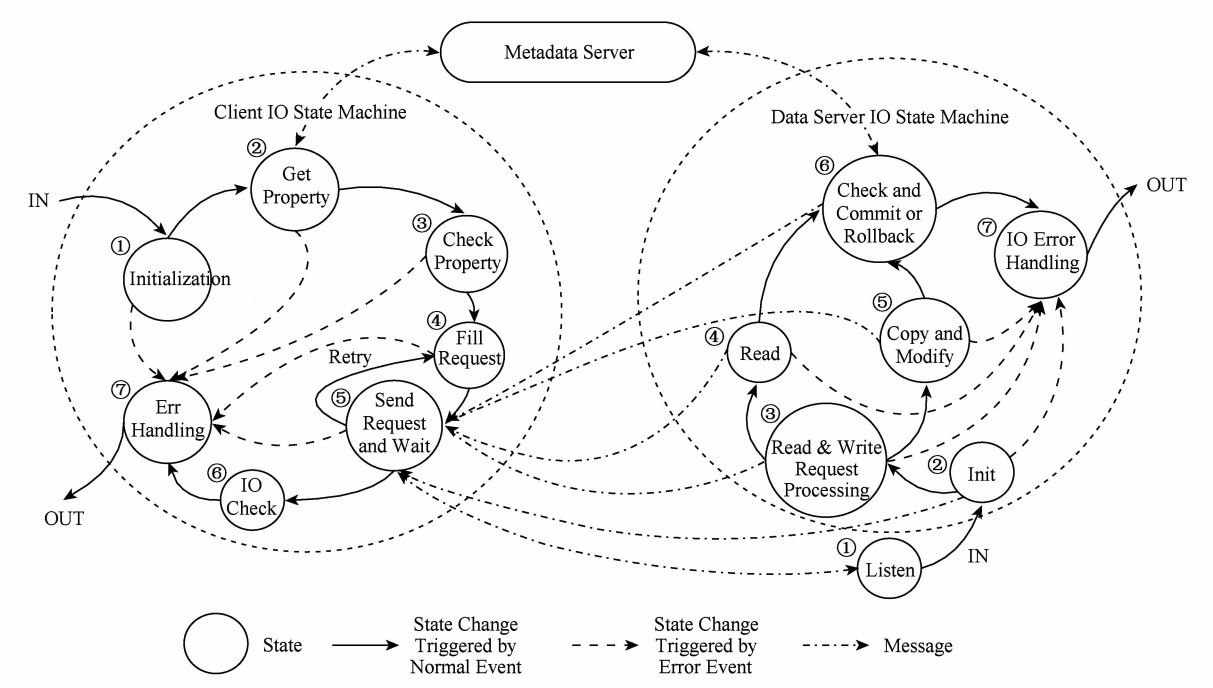


Fig. 2 Lock free IO state machine for data reading and writing

图 2 文件数据锁无关读写状态机

数据服务器端的 IO 子状态机如图 2 右半部所示,其执行过程是:

1) 主状态机执行监听任务,等待各个客户端的访问请求,一旦接收到请求消息,转入 IO 子状态机;

2) 初始化状态机,初始化相关数据结构,启动流处理机制;

3) 分析请求类型,如果是读请求,派送给"读 IO"状态;如果是写请求,派送给"拷贝并修改"状态;

4) 读 IO 状态中,执行读取操作,并将结果以消息的形式发送给客户端;

5) 拷贝并修改状态,数据会被拷贝出来 1 份,形成新版本,并按照写请求来进行更新;

6) "结果检查并提交或退出"状态是原子的,对于读请求结果检查总是成功,对于写请求,该状态执行逻辑会检查数据的版本,确认数据没有被服务器的其它并发线程修改过,确认结果为真则提交修改并通知元数据服务器,确认结果为假则修改作废并通知客户端;

7) 如状态机执行过程中发生故障,检查 IO 错误,处理并退出子状态机.

### 1.4 任务执行底层优化

大数据平台具有应用针对性,平台执行引擎中的计算也具备一定的模式,计算的规律性使得我们可以利用线下耗时的 SuperOptimizaion 编译优化技术来构建具有应用针对性的优化执行引擎,从指令级别去加速计算.图 3 所示为指令级加速引擎的框图.

图 3 中部为执行引擎使用的辅助存储:优化代码数据库和指纹 Map.首先,引擎的优化以基本块(basic block)为单位,即程序中可连续执行的指令序列的集合.代码优化数据库用来存放优化过后的

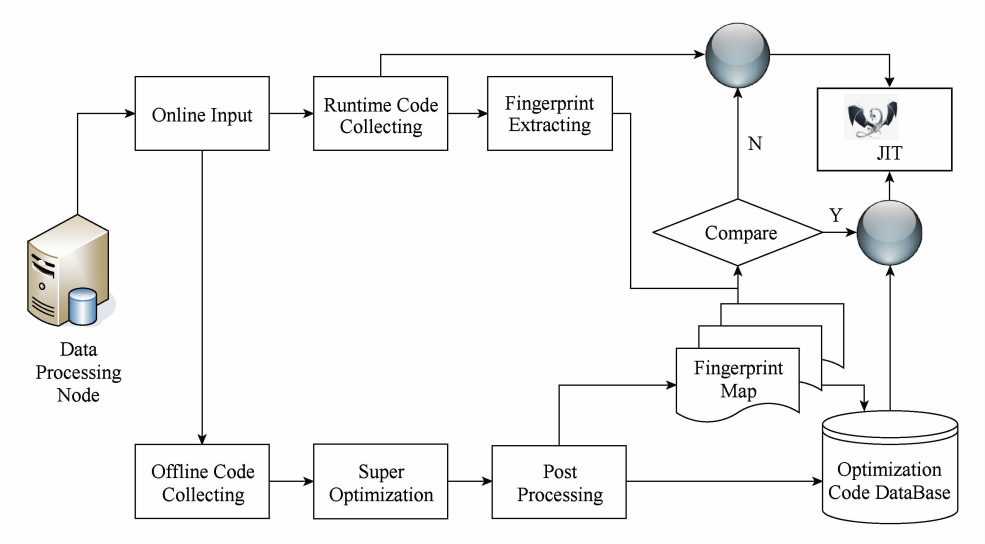


Fig. 3 Super optimization based execution engine framework

图 3 Super Optimization 指令级加速引擎框图

基本块;指纹 Map 则记录了基本块的 Hash 值与数据块存放地址间的映射关系.

图 3 下部为学习流程,功能模块依序为离线采集模块、超级优化模块、后处理模块. 采集模块以 Basic Block,即程序中可连续执行的指令序列集合,为单位,结合部分启发式的规则,来预先选取值得进一步优化的程序段. 被选中的基本块进一步被各个优化模块处理. 目前的原型实现中包括 3 种优化:常量合并(constant fold)、无用变量消除(unused variable suppression)、子表达式提取(sub expression extraction). 这些优化在其他工程中,如 GCC,都是比较成熟的技术,我们这里优化的特点是:1)2 阶段,不仅在 LLVM 的中间表示(IR)阶段作优化,而且在平台选择后,执行 Target-Dependent 的优化;2)穷举模式的优化,对于无用变量消除、子表达式提取,对表达式采用黑盒法作输入值的穷举测试,通过观测确定可以消除的变量或提炼表达式,虽然耗时,但由于在架构中属于离线操作,所以不影响平台的执行效率. 这种优化方式不仅对平台上用户提交的大数据应用,甚至对于成熟的算法也具备优化的潜力.

学习流程后处理模块以优化成功的基本块为输入,计算指纹,将指纹及优化成功后的基本块中间表示或机器码段分别存入指纹 Map 和优化代码数据库.

图 3 上部为工作流程,功能模块依次为在线输入模块、运行时采集模块、指纹提取模块、比较模块、优化 JIT 引擎. 采集模块处理经在线输入模块导入的程序,提取其中的基本块;指纹提取模块以采集模块提取的基本块为输入,计算其指纹;比较模块将计算得到的指纹与指纹 Map 中的指纹作匹配,根据结果(命中与否)来控制开关,保证输入 JIT 引擎的是最优的机器码段. 优化 JIT 引擎负责执行,对于中间表示代码,JIT 首先编译基本块为相应的机器码段,进而将其交给 CPU 执行;对于机器码段,则由 CPU 直接译码执行.

## 2 实 现

我们以 Hadoop 为基础,采用我们的设计思路和优化方法进行改造,得到原型大数据处理系统 Arion,其中,任务执行层位于大数据框架与硬件平台的衔接处,是系统正交分解的关键. 首先,执行层中的连接件是框架技术的外延,是封装为类的大数据处理通用代码库,在大数据处理中会被执行引擎频繁调用;其次,执行引擎向上承接用户的大数据处理请求,向下驱动底层平台,其选型与实现直接影响系统的整体性能和跨平台特性.

图 1 中部任务执行层的左边为面向 Hadoop MapReduce 框架的连接件,通过选择合适功能的连接件,才能无缝衔接 Map 和 Reduce 阶段,形成大数据处理工作流.

连接件以类的形式被预编译成 JIT 引擎可以执行的二进制代码,或易于发布的 IR 形式,并于任务执行时按需动态加载. 对于较常用的连接件,如 Hadoop 中的文件读取类 LineRecordReader(见 Hadoop 官方 API),需要预先编译成所在平台的二进制代码,以动态链接库的形式加载,这种方式更具性能优势;对于不常使用的连接件,如 TeraSort 中

的 TotalOrderPartitioner(见 Hadoop 官方 API),会收集输入数据的样本来构建 Trie 树,进而为 Map 的结果做分区;这是一种特殊逻辑,仅在 TeraSort 处理流程中使用,因此实现中该类以 IR 形式发布,便于增加灵活性,同时减小动态加载的大数据处理通用代码库的内存占用量.

连接件向上与调度管理层通过代理协议来通信和同步.实现中代理协议 ArionPipe 基于 HadoopPipe 来实现,扩充了 HadoopPipe 的命令,使其能够支持更多平台相关的连接件,如用于排序的连接件 Sorter,能够支持分布式缓存等等. ArionPipe 目前是基于 Socket 的点对点协议,在拓扑和协议数据压缩方面还有较大优化空间.

连接件通过调用库函数的形式来集成 ArionFS 客户端,进而能够向下直接访问 ArionFS. 如 LineRecordReader,LineRecordWriter 都分别集成了 ArionFS 客户端,在大数据读取和写入时通过运行锁无关 IO 状态机来与 ArionFS 服务器通信.又如 TotalOrderPartitioner 通过集成 ArionFS 客户端,能够读取缓冲的采样文件,进而构建 Trie 树完成分区任务;Kmeans 大数据计算的 Mapper 用来完成浮点数的聚类,也能够通过 IO 状态机来读取缓冲在 ArionFS 中的分类文件.

Arion 的任务执行引擎基于 LLVM 实现,不但能够利用平台相关技术来优化大数据处理,还能够保证系统的跨平台特性.

## 3 实　　验

为了验证 Arion 正交分解框架改造的正确性以及相对的性能提升,我们做了文件系统测试,性能扩展性测试,以及 MapReduce 任务的阶段分析.

测试集包括类 HiBench 的 4 个大数据应用:

1) DFSIO.用于对分布式文件系统做读写性能测试,每个 Map 任务根据键值做对应文件的读或写操作,根据 Value 值确定文件大小,Reduce 任务收集吞吐率等统计信息.

2) WordCount.用于提取大文件中出现的单词并统计分析单词出现的频次.

3) TeraSort.该实验采用 TeraGen 产生的 1GB 大文件(文件中每行记录在百字节左右,总行数为千万量级),通过 TotalOrderPartitioner 做数据的采样分区(采样参数为 100 KB),进而排序.

4) Kmeans.该实验实现了机器自动聚类,采用了循环迭代的算法,实验中采用了 10 KB 的双精度数据集,并设定了初始聚类参数.

实验共采用了 5 台同配置 HP 服务器、Intel Xeon 32 核处理器、32 GB 的内存、1 TB SATA 硬盘;操作系统为 CentOS 6.文件系统实验、性能扩展性实验中,1 台服务器固定用作 Hadoop/Arion 的 JobTracker 和任务提交、监测客户端;另外 1～4 台服务器用作任务执行 TaskTracker. Case Study 实验中,共采用了 3 台服务器,1 台用作 JobTracker 和客户端,另外 2 台用作 TaskTracker.所有实验中,分布式文件系统(HDFS/ArionFS)部署在参与当前实验的服务器上.

### 3.1 文件系统测试

本实验中,1 台服务器用作:1)MapReduce 任务管理 JobTracker;2)文件系统元数据节点 Namenode;3)任务提交客户端.其他 1～4 台服务器用作:1)任务执行 TaskTracker;2)文件系统数据节点 DataNode.所有 Map 任务会发起对分布式文件系统的读写操作,形成对各个数据节点的 IO 压力,Reduce 任务则收集、分析各个 Map 任务的 IO 相关统计信息.

本实验中,HDFS 和 ArionFS 文件系统的基本分块大小 ChunkSize 皆为 64 MB;由于 ArionFS 暂不支持副本机制,将 HDFS 的副本数设置为 1;实验测试文件集设置为:10 个文件,每个文件 1 GB.每组实验分别作了 10 次,根据平均数绘制实验图表.

由图 4(a),对于写测试,随着分布式文件系统节点数的增加,无论是 HDFS 还是 ArionFS,其执行时间有增加的趋势,而吞吐率有减小的趋势,这是由于 DFSIO 测试中,数据是由客户端节点分发到各个数据节点的,随着数据的分布范围的增加,10GB 实验数据的分发带来了网络开销,致使执行时间增加;另一方面,随着节点数增加,并发 IO 也会带来带宽的聚合、性能的提升,由图 4(a)可知,4 节点实验的执行时间与 3 节点实验相比,反而有所减少.

从完成时间上分析,1～4 个节点的文件系统写测试中,ArionFS 的完成时间较 HDFS 分别减少了 2.0%,2.7%,3.7%,3.1%;从吞吐率上分析,ArionFS 的吞吐率较 HDFS 分别提高了 2.2%,3.1%,1.0%,0.8%.综上,ArionFS 的平均完成时间较 HDFS 减少 2.9%,ArionFS 的平均吞吐率较 HDFS 提高 1.8%.

由图 4(b),对于读测试,随着分布式文件系统节点数的增加,无论是 HDFS 还是 ArionFS,其执行时间变化不大,略有减少,这里执行时间记录的是

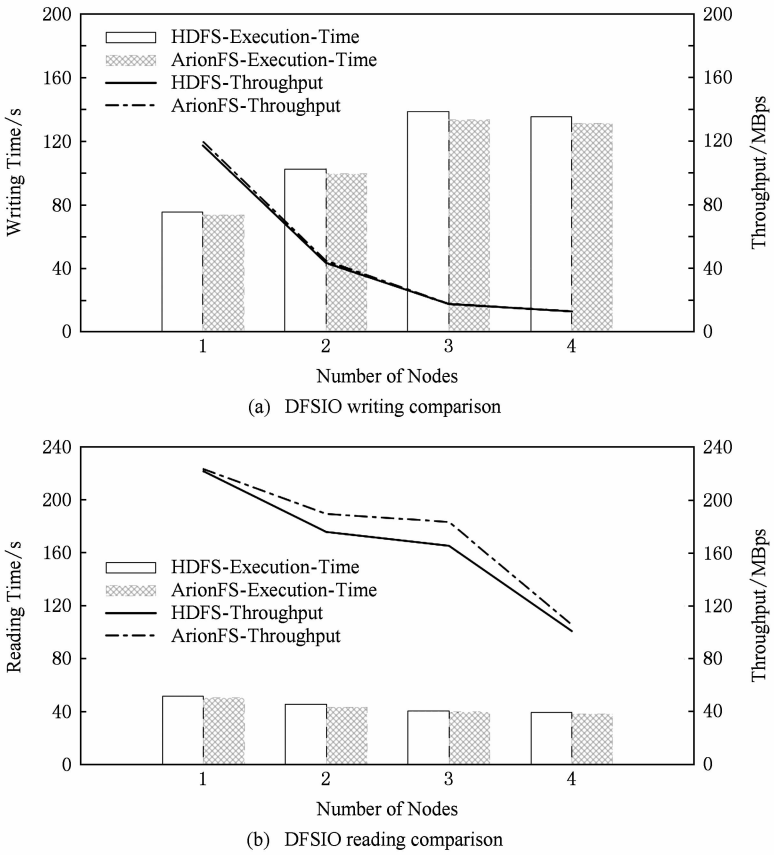


Fig. 4 Distributed file system test based on DFSIO benchmark

图 4 基于 DFSIO 测试集的分布式文件系统测试

整个实验从开始到结束的总时间,即包括文件读阶段,也包括 Shuffle,Merge 等衔接阶段,以及最后的 Reduce 阶段和统计信息分析阶段;吞吐率方面,两个文件系统均随着节点数的增加而减小,这是由于数据分布带来了网络开销,ArionFS 在吞吐率上的表现要略优于 HDFS. 需要说明的是,本实验中吞吐率与完成时间不成反比的原因是 DFSIO 实验中吞吐率并非由完成时间计算得出,而是根据每个 Map 任务中的文件读取时间来计算的.

从完成时间上分析,1~4 个节点的文件系统读测试中,ArionFS 的完成时间较 HDFS 分别减少了 2.0%, 4.2%, 1.8%, 2.9%; 从吞吐率上分析, ArionFS 的吞吐率较 HDFS 分别提高了 0.7%, 7.7%,10.8%,4.7%. 综上,ArionFS 的平均完成时间较 HDFS 减少 2.7%,ArionFS 的平均吞吐率较 HDFS 提高 6.0%.

### 3.2 性能、扩展性测试

本实验首先从性能上针对 3 种负载展开 Hadoop 与 Arion 的对比. 实验中,文件系统的基本分块大小 ChunkSize 和大数据处理引擎的处理单元大小 SplitSize 皆为 64 MB.

由图 5(a),WordCount 实验中,单节点 Arion 在 Vanilla Hadoop 基础上优化后,执行时间缩短 3.0%;2,3,4 个节点也分别缩短 5.0%,3.0%, 3.0%,平均缩短 3.5%. 根据图 5(b)(c),TeraSort 实验中,相较 Vanilla Hadoop,Arion 的执行时间平均缩短 7.7%;Kmeans 实验中,平均缩短 3.5%.

其次,由实验结果可知,各个实验中随着节点数的增加,无论是 Vanilla Hadoop 还是 Arion,其执行时间都是在缩短的,可见 Arion 在做正交分解和优化后,保留了原有大数据处理系统的良好扩展性. 其中,Kmeans 实验后期,随着节点的增加,性能改善较小,多节点时扩展性表现一般的原因是 Kmeans 随着节点的增多,瓶颈主要集中在 Reduce 阶段,包括从各个分节点调度、提取、排序数据,并重新计算分类中心点.

### 3.3 Case Study

本实验中我们作典型 MapReduce 任务执行各阶段的精细化分析. 表 1、表 2 中,依序(由下至上)记录了如下 9 种大数据任务处理阶段的执行时间:

1) 适配层的连接件 Reader,如 LineRecordReader,用于与 ArionFS 对接,通过运行锁无关 IO

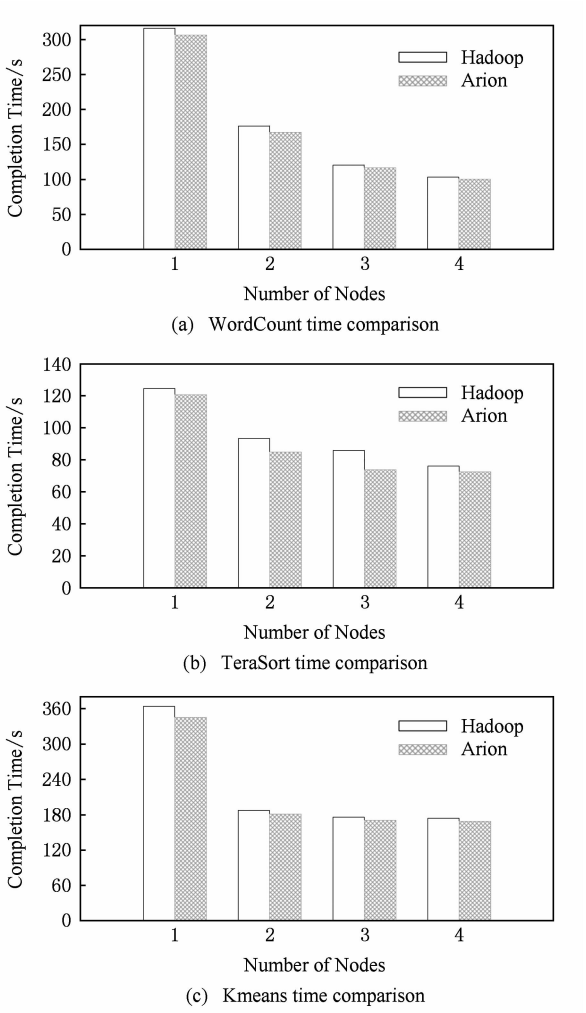


Fig. 5 Performance and scalability test

图 5 性能及扩展性测试

状态机读取分布式文件系统的数据;

2) Mapper,MapReduce 框架的 Map 任务处理阶段;

3) 连接件 Partitioner,如 3.1 节介绍的 TotalOrderPartitioner,用于对 Mapper 的输出结果根据一定的算法做分区;

4) 连接件 Combiner,用于对 Mapper 的输出结果作部分合并;

5) 连接件 Merger,对多个分布式的 Mapper 的输出结果做归并提取;

6) 连接件 Sorter,在 Reduce 预处理阶段,做多路 Mapper 提取数据的排序;

7) Reducer,MapReduce 框架的 Reduce 任务处理阶段;

8) 连接件 Writter,如 3.1 节介绍的 LineRecordWriter,用于与 ArionFS 对接,通过运行锁无关 IO 状态机向分布式文件系统写入数据;

9) Others,除去上述阶段外的其他开销,如 Arion 中 LLVM 引擎与上层 Hadoop JVM 虚拟机间的通信等.

本实验皆采用 1 台 JobTracker 服务器、2 台 TaskTracker 服务器的硬件配置,与 Map 相关的任务处理阶段的执行时间为平均值.

表 1 为 WordCount 实验,采用 100 MB 的文件,而计算引擎的 Split Size 为 64 MB,共切分为 2 个 Map 任务.该实验 Arion 的面向底层平台的优化集中在 Reader,Mapper,Reducer,Writer 四个阶段.如表 1,Arion 的 Mapper 阶段经优化其执行时间较 Hadoop 减少 7.9%,但是,在 Others 阶段,Arion 的通信、调度开销较大,执行时间反而增加了 5.0%.因此,总时间 Arion 仅比 Hadoop 减少了 3.0%.

**Table 1 Wordcount Stage Analysis**

**表 1 Wordcount 执行阶段分析**

| Test Set | Hadoop | Arion |
|---|---|---|
| Combiner | 2.78 | 2.78 |
| Mapper | 31.38 | 28.9 |
| Partitioner | 0 | 0 |
| Combiner | 1.833 | 1.8 |
| Merger | 1.8 | 1.8 |
| Sorter | 0.397 | 0.3 |
| Reducer | 0.596 | 0.5 |
| Writer | 1.407 | 1.4 |
| Others | 18.183 | 19.77 |

表 2 为 TeraSort 实验,采用 1 GB 的文件,而计算引擎的 Split Size 为 512 MB,共切分为 2 个 Map 任务.该实验 Arion 的面向底层平台的优化集中在 Reader, Mapper, Partitioner, Merger, Reducer, Writer 六个阶段.如表 2 所示,相对于 Hadoop, Arion

**Table 2 Terasort Stage Analysis**

**表 2 Terasort 执行阶段分析**

| Test Set | Hadoop | Arion |
|---|---|---|
| Combiner | 10.98 | 9.76 |
| Mapper | 13.45 | 12.13 |
| Partitioner | 7.17 | 6.37 |
| Combiner | 0 | 0 |
| Merger | 9.14 | 8.13 |
| Sorter | 0.02 | 0.02 |
| Reducer | 30.91 | 26.46 |
| Writer | 16.82 | 14.13 |
| Others | 6.64 | 8.6 |

的 Mapper 阶段执行时间减少 9.8%,Partitioner 部分,即 TotalOrderPartitioner,执行时间减少 11.2%,Reducer 部分减少 14.4%,这几部分提升较大与计算密集,引擎优化效率较高有关;但是,在 Others 阶段,Arion 的通信、调度开销较大,执行时间反增 29.5%.因此,总时间 Arion 仅比 Hadoop 减少了 10.0%.

综上,本实验表明:Arion 对大数据处理部分执行阶段、读写阶段的优化具有明显效果,但也会引入额外的通信、调度开销.

## 4 相关工作

大数据处理系统的发展是二维的.从横向看,各个系统的框架主要围绕着数据处理负载的特性来进化.Google 的 MapReduce[2] 是典型的 2 阶段处理框架,具备一定的通用性,但最初主要是针对搜索引擎网页的批量抓取而设计.Nokia 的 Disco[3] 系统也采用了针对批量大数据处理而设计的框架.Microsoft 的 Dryad[5] 则是主要面向工作流处理负载的框架,这类框架还包括 Google 的 Dremel[4],Twitter 的 Storm[6],Yahoo 的 S4[7] 等.面向图计算,Pregel[13] 和开源的 Giraph① 是典型代表.针对 MapReduce 模型不擅长的迭代处理和交互应用,伯克利 AMP Lab 提出了基于 RDD[8] 内存数据集的迭代模型处理框架,并设计了内存大数据处理系统 Spark[9].

从纵向阶段上看,各大数据处理系统自身也是围绕着框架、调度算法的升级而升级的.Hadoop 自身的计算框架由原本单一的 MapReduce 演化出了基于 DAG 的更为灵活的 Tez;自身的调度也从单一的全局任务调度发展到了新一代的 Yarn[10],分离了 JobTracker 的资源管理与任务调度功能.Mesos[14] 是参照操作系统内核而设计的大数据资源管理系统,其自身也是围绕着资源调度种类的增加、资源调度算法的升级而发展的,在 Hadoop 生态体系中,其作用类似于 Yarn.

综上,现有大数据处理系统的发展无论从那个维度看,计算模型的改进、框架和算法的升级是关注点,底层平台相关技术的利用存在空白.Intel 中国研究院的 NativeTask[11] 通过设计外挂的计算引擎模块,将部分 Hadoop 计算引擎内部的计算外延到 Java 虚拟机之外;百度公司的 Hadoop 的 C++扩展②,通过使用类似 Pipe 的协议将 Map 和 Reduce 两阶段的 Java 执行逻辑替换为 C++编写并预编译好的二进制可执行文件,都取得了一定的本地化效果,但都未充分发挥存储结点、计算结点本地平台的潜力.

大数据文件系统是大数据处理的基石,如 Google 公司的 GFS[15],Amazon 的 S3,Dynamo[16],Apache 的 Cassandra③,Hadoop 中的 HDFS[17] 以及基于快速网络的 FDS[18] 等,它们都提供类 Key/Value 的存储抽象和基于副本的可靠性机制,为上层的大数据应用服务.新一代内存文件系统 Tachyon[19] 以内存数据集为主要抽象为上层提供大数据存储服务,其中引入了 Lineage[20] 的抽象来实现内存数据的可靠性,大大减少了存储空间占用,提高了写入带宽,但也引入了重计算的调度等问题.同样采用 Lineage 抽象的还有 BADFS[21],该文件系统提供了显示的描述式语言接口,使得内核能够从用户输入获得 Lineage 信息.由上述可知,大数据文件系统的主流聚焦于在本地系统上建立一层分布式抽象层,提供适合处理的数据模型和可靠性保障等服务,缺乏本地化相关领域的研究.

任务执行引擎需要运行时(managed runtime environment, MRE)支撑,这主要是由于运行时能够为程序运行提供灵活性和安全性.成熟运行时,如 SmallTalk[22],Self[23],JVM[24] 都有严格的高级语言绑定,如 type-safe 的检查,需要程序符合其 MRE 的设计.微软的 CLI[25] 能够支持多种语言,但对语言互操作性有严格限制,不支持非控制(unmanaged)语言的优化,且不开源.LLVM[12] 是一个开源的编译器设计框架,提供通用、基于静态单一分配(static single assignment, SSA)模式的代码中间表示,同时具备系列开源前端,支持多种高级语言,后端能做多遍优化,因此是本文选型方案.

## 5 结论和局限性及未来工作

我们提出一种基于正交分解的大数据处理系统设计与优化方法,主张将整个系统划分为功能上互相正交的、松耦合的模块,将数据存储与任务执行下

① http://giraph.apache.org

② Hadoop C++ Extension: A Framework for Making MapReduce More Stable and Faster. http://wenku.baidu.com/view/1859f988d0d233d4b14e69c8.html

③ http://cassandra.apache.org

沉到具体的硬件平台去完成,大数据系统只负责最核心的资源与任务调度.进而,在此方法的指导下,我们提出了基于锁无关机制的存储底层优化策略,以及基于指令超级优化的执行引擎底层优化策略.这种优化方式与主流的面向计算模型、框架、算法的发展趋势并不矛盾,是正交、互补的关系,有效填补了技术空白.

基于上述技术,以 Hadoop 作为兼容和改进的对象,我们实现了原型大数据处理系统 Arion,采用正交分解,将 Hadoop 的任务调度与存储、计算引擎拆分开,通过通信协议连接;数据存储下沉到 C 语言实现的锁无关大数据分布式文件系统 ArionFS 中;任务执行下沉到基于 LLVM 的本地化执行引擎.经实验证明,同等硬件配置下,Arion 的性能优于 Hadoop.

本技术局限在于:分解后下沉到平台的数据存储与任务执行引擎带来的加速比依赖于负载模式、代码特点及 JIT 引擎实现,对于数据分布均衡、处理逻辑简单、优化代码段执行频率低的任务,如何提高收益,深挖平台潜力是我们下一步工作的重点.另外,正交分解方法在 Spark 平台上的应用也是未来工作之一,Spark 平台将调度交给 Mesos、存储交给 HDFS 等,已然符合正交分解思想,并为后续面向平台优化做了铺垫;其计算引擎层面的优化是难点,需要针对 RDD 的 Transformation 和 Action 执行逻辑中间表达作超级优化,充分考虑代码特点及 JIT 引擎实现;存储可以直接交给 ArionFS.

## 参 考 文 献


[1] Wang Yuanzhuo, Jin Xiaolong, Cheng Xueqi. Network big data: Present and future [J]. Chinese Journal of Computers, 2013, 36(6): 1125-1138 (in Chinese)
(王元卓, 靳小龙, 程学旗. 网络大数据: 现状与展望[J]. 计算机学报, 2013, 36(6): 1125-1138)

[2] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters [J]. Communications of the ACM, 2008, 51(1): 107-113

[3] Mundkur P, Tuulos V, Flatow J. Disco: A computing platform for large-scale data analytics [C] //Proc of the 10th ACM SIGPLAN Workshop on Erlang. New York: ACM, 2011: 84-89

[4] Melnik S, Gubarev A, Long J J, et al. Dremel: Interactive analysis of Web-scale datasets [J]. Proc of the VLDB Endowment, 2010, 3(1/2): 330-339

[5] Isard M, Budiu M, Yu Y, et al. Dryad: Distributed data-parallel programs from sequential building blocks [C] //ACM SIGOPS Operating Systems Review. New York: ACM, 2007: 59-72

[6] Leibiusky J, Eisbruch G, Simonassi D. Getting Started with Storm [M]. Sebastopol: O'Reilly Media, Inc, 2012

[7] Neumeyer L, Robbins B, Nair A, et al. S4: Distributed stream computing platform [C] //Proc of the 2010 IEEE Int Conf on Data Mining. Piscataway, NJ: IEEE, 2010: 170-177

[8] 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: 2

[9] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster computing with working sets [C] //Proc of the 2nd USENIX Conf on Hot Topics in Cloud Computing. Berkeley, CA: USENIX Association, 2010: 10

[10] Vavilapalli V K, Murthy A C, Douglas C, et al. Apache Hadoop yarn: Yet another resource negotiator [C] //Proc of the 4th Annual Symp on Cloud Computing. New York: ACM, 2013

[11] Yang D, Zhong X, Yan D, et al. NativeTask: A Hadoop compatible framework for high performance [C] //Proc of IEEE Int Conf on Big Data. Piscataway, NJ: IEEE, 2013: 94-101

[12] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation [C] //Proc of Int Symp on Code Generation and Optimization. Piscataway, NJ: IEEE, 2004: 75-86

[13] Malewicz G, Austern M H, Bik A J C, et al. Pregel: A system for large-scale graph processing [C] //Proc of the 2010 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2010: 135-146

[14] Hindman B, Konwinski A, Zaharia M, et al. Mesos: A platform for fine-grained resource sharing in the data center [C] //Proc of Symp on Network System Design and Implementation. Berkeley, CA: USENIX Association: 2011: 22

[15] Ghemawat S, Gobioff H, Leung S T. The Google file system [C] //Proc of the 19th ACM Symp on Operating Systems Principles. New York: ACM, 2003: 29-43

[16] DeCandia G, Hastorun D, Jampani M, et al. Dynamo: Amazon's highly available key-value store [C] //Proc of ACM SIGOPS Operating Systems Review. New York: ACM, 2007: 205-220

[17] Shvachko K, Kuang H, Radia S, et al. The Hadoop distributed file system [C] //Proc of the 26th IEEE Symp on Mass Storage Systems and Technologies. Piscataway, NJ: IEEE, 2010: 1-10

[18] Nightingale E B, Elson J, Fan J, et al. Flat datacenter storage [C] //Proc of the 10th USENIX Symp on Operating Systems Design and Implementation (OSDI 12). Berkeley, CA: USENIX Association. 2012: 1-15

[19] Li H, Ghodsi A, Zaharia M, et al. Tachyon: Reliable, memory speed storage for cluster computing frameworks [C] //Proc of the ACM Symp on Cloud Computing. New York: ACM, 2014: 1-15

[20] Bose R, Frew J. Lineage retrieval for scientific data processing: A survey [J]. ACM Computing Surveys, 2005, 37(1): 1-28

[21] Bent J, Thain D, Arpaci-Dusseau A C, et al. Explicit control in the batch-aware distributed file system [C] //Proc of Symp on Network System Design and Implementation. Berkeley, CA: USENIX Association, 2004: 365-378

[22] Deutsch L P, Schiffman A M. Efficient implementation of the smalltalk-80 system [C] //Proc of the 11th ACM SIGACT-SIGPLAN Symp on Principles of Programming Languages. New York: ACM, 1984: 297-302

[23] Ungar D, Smith R B. Self: The Power of Simplicity [M]. New York: ACM, 1987

[24] Lindholm T, Yellin F, Bracha G, et al. The Java Virtual Machine Specification [M]. Reading, MA: Addison-Wesley, 2014

[25] Miller J S, Ragsdale S. The Common Language Infrastructure Annotated Standard [M]. Reading, MA: Addison-Wesley, 2004

**Xiang Xiaojia**, born in 1977. PhD, associate professor. Senior member of CCF. His main research interests include big data processing system, distributed storage, operating system, etc.

**Zhao Xiaofang**, born in 1965. PhD, professor, PhD supervisor. Her main research interests include computer architecture, data management, information security, etc.

**Liu Yang**, born in 1991. Master candidate. His main research interests include operating system, distributed system, information security, etc.

**Gong Guanjun**, born in 1993. Master candidate. His main research interests include operating system, network security, etc.

**Zhang Han**, born in 1993. Bachelor. Her main research interests include digital media, distributed storage, etc.