# 跨语言用户态文件系统框架读写性能优化

顾 荣[1,2],罗义力[2],仇伶玮[2],王肇康[1,3],戴海鹏[1,2],黄宜华[1,2]
(1. 计算软件新技术国家重点实验室(南京大学),江苏南京 210023;2. 南京大学计算机科学与技术系,江苏南京 210023;
3. 南京航空航天大学计算机科学与技术学院,江苏南京 211106)

**摘 要:** 以深度学习为代表的数据分析应用越来越多依赖分布式文件系统存储管理大规模数据集. 为了增强数据访问的兼容性,现有分布式文件存储系统通常需提供标准POSIX接口,以支持深度学习等应用的无缝对接. 然而,以内核模块形态开发提供POSIX接口的文件系统非常复杂耗时. 近年来,用户态文件系统(Filesystem in Userspace,FUSE)框架大幅简化了文件系统的开发工作,已被Alluxio和Ceph等诸多知名分布式文件系统使用. 目前常用的用户态FUSE库libfuse仅提供C语言编程接口,但现有大数据分布式文件系统基本都是基于Java语言开发的(例如HDFS和Alluxio等),为了使基于Java语言开发的分布式文件系统可以对接C语言开发的FUSE库,需采用跨语言FUSE框架作为中介. 跨语言FUSE框架利用跨编程语言的函数回调机制,使操作系统FUSE库的C语言函数可以跨语言的调用分布式文件系统提供的Java语言编程接口,从而为大数据分布式文件系统提供标准POSIX接口的访问能力. 但在数据密集型应用中,现有跨语言FUSE框架的执行效率低,导致数据密集型作业(深度学习、大数据分析等)中数据I/O耗时占据了显著的性能开销,成为新的潜在性能瓶颈. 针对此问题,本文首先评估分析了重要且广为使用的跨语言FUSE框架JNR-FUSE的性能,发现并定位其在高并发和小文件场景下存在的性能瓶颈;接着从多方面剖析性能瓶颈根因,进而总结出高效跨语言FUSE框架的性能优化方向,并面向Java语言设计实现了跨语言FUSE框架JNI-FUSE. JNI-FUSE利用延迟分离和元信息缓存等优化技术降低跨语言函数回调开销,从而提升跨语言FUSE框架的性能. 实验结果表明,对比当前性能最好的Java FUSE框架JNR-FUSE,本文提出的JNI-FUSE带来了1.15~6.04倍的FUSE框架性能提升和1.90~2.71倍的文件系统端到端性能提升,并为上层深度学习训练任务带来了1.06~1.73倍的训练加速. 本文设计提出的JNI-FUSE(Java Native Interface-Filesystem in User SpacE)因性能优势,已被知名开源分布式文件系统Alluxio官方接受集成.

**关键词:** POSIX;用户态文件系统;跨语言;性能优化;Java原生接口
**基金项目:** 国家自然科学基金面上项目(No.62072230, No.61872178);江苏省科技厅重点项目(No.BE2021729);软件新技术国家重点实验室开放课题(No.KFKT2021B33);软件新技术与产业化协同创新中心项目
**中图分类号:** TP31 **文献标识码:** A **文章编号:** 0372-2112(2023)06-1590-17
**电子学报URL:**http://www.ejournal.org.cn **DOI:**10.12263/DZXB.20220372

# Reading and Writing Performance Optimization of Cross-Language FUSE Framework

GU Rong[1,2], LUO Yi-li[2], QIU Ling-wei[2], WANG Zhao-kang[1,3], DAI Hai-peng[1,2], HUANG Yi-hua[1,2]
(1. *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu* 210023, *China*;
2. *Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu* 210023, *China*;
3. *College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu* 211106, *China*)

**Abstract:** Big data analytical applications like deep learning-based AI applications more and more rely on distributed file system to store and manage large scale data sets. File systems often need to provide standard POSIX interfaces to enhance their access compatibility with upper-layer applications. However, it is complicated to develop POSIX-compatible file systems in kernel space. In recent years, FUSE (File System in User Space) has been used by many well-known file systems, including Alluxio, Ceph, etc., because it significantly simplifies the file system development. The popular FUSE library libfuse is developed in the C language. However, the popular distributed file systems (like HDFS and Alluxio) for big data applications are developed with the Java language. To make the Java-based distributed file systems use the FUSE

收稿日期:2022-04-08;修回日期:2022-08-16;责任编辑:王天慧

mechanism, the cross-language FUSE frameworks are needed to bridge the gap, which becomes a potential performance bottleneck. The cross-language FUSE framework uses a cross-programming language function callback mechanism to enable the C functions of the FUSE library to call the programming interface provided by the distributed file system in Java. In this way, we can provide the access to the standard POSIX interface for the Java-based distributed file systems. However, the existing cross-language FUSE frameworks are inefficient in performance. It makes the data I/O in data-intensive applications (like deep learning and big data analysis) occupy noticeable proportion of their execution costs. To address the problem, we first systematically evaluate the performance of the widely-used cross-language FUSE framework, and find the bottlenecks of throughput performance in high concurrency and small file scenarios. We then analyze the bottlenecks of the cross-language FUSE framework from multiple perspectives, and propose several directions for optimizing the cross-language FUSE framework. According to the optimization directions, we design and implement JNI-FUSE (Java Native Interface-Filesystem in User SpacE), an efficient cross-language FUSE framework. In JNI-FUSE, we propose the defer detach and meta cache techniques to reduce execution costs of cross-programming language function callbacks. Experiment results show that JNI-FUSE improves the average framework performance from 1.15 times to 6.04 times compared to the cutting-edge cross-language FUSE framework JNR-FUSE. JNI-FUSE improves the end-to-end performance by 1.90 time to 2.71 times, and accelerates the deep learning training by 1.06 times to 1.73 times compared to JNR-FUSE. JNI-FUSE has been accepted and integrated by the well-known open-source distributed file system Alluxio due to its good performance.

Key words: portable operating system interface; user-space file system; cross-language; performance optimization; java native interface

Foundation Item(s): National Natural Science Foundation of China (No.62072230, No.61872178); Key Project of Jiangsu Provincial Science and Technology Department (No.BE2021729); Open Project of State Key Laboratory for Novel Software Technology (No.KFKT2021B33); The project of Collaborative Innovation Center of Novel Software Technology and Industrialization

## 1　引言

人工智能和大数据分析应用通常依赖大规模数据集训练和分析. 基于磁盘的传统单机本地文件系统难以满足此类应用对于性能和可用性的要求. 面向数据密集型应用而设计的分布式数据存储系统(例如HDFS, Alluxio[1], M-Cloud[2])通过可扩展的系统架构,可以较好地解决此问题. 但许多现有人工智能和数据分析应用(例如基于PyTorch的深度学习应用)只支持从提供POSIX[3]接口的传统单机本地文件系统读写数据,面临着难以直接读写分布式数据存储系统数据的问题.

为了支持现有人工智能和数据分析应用快速简单地访问数据,众多存储系统厂商引入POSIX对接层,将分布式文件系统模拟为提供POSIX接口的本地文件系统,以帮助现有人工智能和大数据应用在不修改现有文件I/O接口的情况下透明读取数据. 因此,POSIX对接层的数据访问效率直接影响了上层人工智能和数据分析应用的总体执行效率.

POSIX对接层需要借助Linux操作系统提供的POSIX兼容文件系统框架,其可进一步由内核态或用户态两种方式之一实现. 以内核态形式提供POSIX兼容的文件系统时,通常需要修改操作系统内核源码,开发和维护难度较高,而用户态文件系统(Filesystem in Userspace, FUSE)[4]允许在不更改操作系统内核的情况下对接自定义的文件系统,大幅降低了文件系统的设计开发门槛,因此成为很多文件系统POSIX对接层的实现方式. 各种用户态文件系统框架[5,6]被广泛使用在Alluxio[1], Ceph[7]和Gluster[8]等主流分布式文件系统中,它们皆借助FUSE提供POSIX访问接口. 与内核态文件系统相比,FUSE虽然简化了文件系统开发过程,但也牺牲了一定的数据访问性能.

目前常用的用户态FUSE库libfuse仅提供C语言编程接口,但分布式文件系统可能采用非C语言开发(例如Java). 为了使基于其他语言开发的分布式文件系统可以对接基于C语言开发的FUSE库,需要额外引入跨语言FUSE框架,以实现文件系统编程接口与基于C语言的FUSE库之间的互相调用. 鉴于现有大数据分布式文件系统基本都是基于Java语言开发(例如HDFS, Alluxio等),因此跨语言FUSE当前主要关注的问题就是面向Java的跨语言FUSE实现.

跨语言FUSE框架的引入进一步放大了用户态文件系统的性能问题. 在深度学习训练、大数据分析等数据密集型作业中,数据I/O的速度会显著影响作业执行时间,进而影响用户的使用成本. 然而,跨语言FUSE框架从系统内核层到语言实现层横跨多个系统层次,比单一语言FUSE框架实现更为复杂[9],导致其性能瓶颈尚未得到充分的分析,现有跨语言FUSE框架存在严重的性能问题.

针对现有面向Java的跨语言FUSE框架存在的性

能问题,本文首先以实验实证的方式分析了影响跨语言FUSE框架性能的关键因素,从而总结了跨语言FUSE框架的性能优化方向. 在优化方向的指导下,本文面向Java语言提出并实现了基于高效跨语言函数回调的跨语言FUSE框架JNI-FUSE. 具体地,本文贡献可以归纳为如下三个方面:

(1)通过设计实验分析,深度评估了目前广为使用的跨语言FUSE框架JNR-FUSE在数据密集型作业下的性能表现,发现其在高并发和小文件场景下存在明显的性能瓶颈. 进一步地,本文从原理与实验两方面分析了性能瓶颈的产生原因,并提出高效跨语言FUSE框架的性能优化方向.

(2)根据上述总结的性能优化方向,本文面向Java提出并实现了高效跨语言FUSE框架JNI-FUSE,该框架通过延迟分离和元信息缓存等优化技术,降低了跨语言FUSE过程中函数回调开销. 本文优化方法可进一步应用于其他基于JVM编程语言开发的文件系统的跨语言FUSE场景.

(3)通过大量实验评估发现,对比当前性能最好的Java FUSE框架JNR-FUSE,本文提出的JNI-FUSE读写性能提升1.15~6.04倍、文件系统端到端性能提升1.90~2.71倍,并可为深度学习训练任务带来1.06~1.73倍的训练加速. 本文设计提出的JNI-FUSE因性能优势,已被知名开源分布式文件系统Alluxio官方接受集成,且作为默认配置使用.

本文的组织结构如下. 第2节介绍FUSE框架和跨语言FUSE框架的背景知识及相关工作,第3节分析跨语言FUSE框架存在的性能瓶颈并总结高效跨语言FUSE框架的优化方向,第4节介绍JNI-FUSE框架的设计与实现,在第5节通过实验评估了JNI-FUSE的性能,第6节总结全文工作.

## 2 相关工作与背景知识

### 2.1 相关工作

跨语言FUSE框架相关工作可分为FUSE性能分析、FUSE性能优化与跨语言FUSE框架设计等三类. 在FUSE性能分析方面,用户态文件系统FUSE相比内核态文件系统提供了更高的安全性和可靠性[10],但它也带来了一定的性能损耗[9],其中最为突出的瓶颈是内核态和用户态的多次通信. Vangoor等人[9]以本地文件系统Ext4为基础,详尽分析了FUSE在不同负载下的性能损耗情况. 实验结果表明FUSE最高可带来超过83%的性能损耗,尽管通过优化FUSE配置,大部分工作负载的性能损耗可以降低到5%以内,但仍存在部分负载(尤其是元数据操作频繁的工作负载)性能大幅弱于内核态文件系统.

在FUSE性能优化方面,FUSE因为横跨内核态与用户态,其性能优化更加困难. Ishiguro等人[11]和Narayan等人[12]曾尝试在特定情况下减少FUSE内核态和用户态的上下文切换. 为进一步优化FUSE性能,Bijlani[10]等提出允许FUSE文件系统针对某些请求自定义一些特殊优化的处理逻辑,从而在不改变原有复杂逻辑的基础上提升性能. 以上FUSE框架优化工作均只面向文件系统与FUSE库采用相同语言实现的情况,重点关注减少内核态和用户态间的消息通信开销,但未考虑跨语言FUSE中跨语言函数调用带来的性能问题,本文在这方面将开展研究.

在跨语言FUSE框架设计方面,目前相关研究较少,主要有JNR-FUSE[13]和FUSE-J[14],两者都提供了FUSE编程接口的Java语言绑定. 相比FUSE-J,JNR-FUSE的应用范围更广且性能更好,并仍处于活跃的维护状态.

### 2.2 用户态文件系统FUSE

图1展示了用户态文件系统的设计架构. FUSE通过Linux的虚拟文件系统(Virtual File System,VFS)机制提供了统一的POSIX访问接口. 它包含一个内核模块(FUSE Driver)和一个用户态函数库libfuse[15],两个模块通过读写一个块设备文件/dev/fuse交互. FUSE Driver负责将应用从VFS传过来的文件系统请求转换为FUSE请求,并加入FUSE处理队列;libfuse通过读取/dev/fuse,从FUSE队列拉取FUSE请求,并交给用户态的工作线程处理,工作线程会调用用户定义的回调函数处理文件系统的访问请求,最后将处理结果通过/dev/fuse写回内核.

为了处理应用程序对用户态文件系统的并发访问请求,libfuse会创建多个工作线程(Daemon线程)并发处理FUSE请求. FUSE的工作线程管理机制的优化原则是尽可能利用高并发度减少FUSE请求的总等待和处理时间. 具体而言,文件系统通常会读写硬盘等较慢速的设备文件,涉及大量I/O请求,导致性能瓶颈通常在于I/O. 为了减少I/O等待带来的计算资源浪费、降低所有请求的总等待时间,FUSE的工作线程管理机制通常会创建足够多的工作线程,以保证所有文件系统调用请求都尽可能被及时处理. FUSE在初始化时仅会创建一个工作线程,当应用程序对文件系统发起并发访问时,应用程序发出的FUSE请求将会在libfuse库内排队等待工作线程处理. 当libfuse库发现排队等待的FUSE请求数量过多时,表明当前工作线程数量不足,libfuse库会创建额外的工作线程以加速处理FUSE请求. 在典型配置下,如果存在超过10个工作线程处于空闲状态,FUSE会回收空闲工作线程以释放资源. 在高并发场景下,受跨语言FUSE高执行开销的影响,

FUSE工作线程的实际处理速度低于I/O请求的到达速度,导致I/O请求堆积、大量工作线程被创建,这些工作线程在对应的I/O请求完成后又被销毁,从而带来大量线程管理开销.

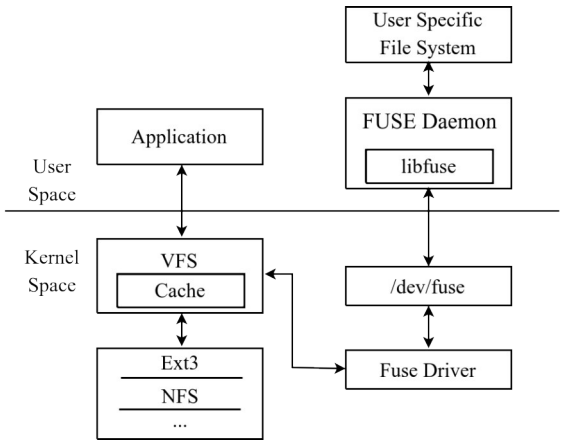



图1　用户态文件系统架构

### 2.3　跨语言FUSE框架JNR-FUSE

目前大数据生态中的常用分布式文件系统(例如HDFS,Alluxio等)多以Java语言实现,而FUSE的用户态函数库libfuse采用C语言实现. 这些文件系统为了能够基于FUSE机制提供POSIX接口,必须依赖跨语言FUSE框架.

跨语言FUSE框架的核心在于实现从libfuse库中的C语言函数,回调文件系统以Java语言提供的应用程序接口. C语言函数通过JVM提供的Java原生接口(Java Native Interface,JNI)[16]机制与JVM虚拟机交互,实现对象创建、方法调用等. 但JNI使用方式较为烦琐,为了降低JNI编程难度,JNA(Java Native Access)[17]和JNR(Java Native Runtime)[18]等技术基于JNI提供了更为简洁的跨语言函数调用方式.

Tselovalnikov等人[13]基于JNR技术提出了Java平台上最为常用的跨语言FUSE框架JNR-FUSE. JNR-FUSE基于JNR实现了FUSE接口的Java绑定,继承了JNR跨平台、易于实现的优点. 图2是JNR-FUSE架构图. JNR为了实现跨平台的通用性、减少用户的编程负担,引入了多层封装结构,这使得JNR-FUSE中跨语言函数调用链更长,性能瓶颈更难以分析. 在第3节,本文将以JNR-FUSE为例,分析FUSE在跨语言使用时存在的性能瓶颈.

## 3　跨语言FUSE框架JNR-FUSE性能分析

### 3.1　性能分析设置

跨语言FUSE过程的端到端总体性能一般由跨语言FUSE框架、FUSE机制本身、文件系统与底层存储介质等因素共同决定. 本文采用Java平台上最为成熟且广为使用的跨语言FUSE框架JNR-FUSE[13]作为评估对象,进行性能评估与现象分析.

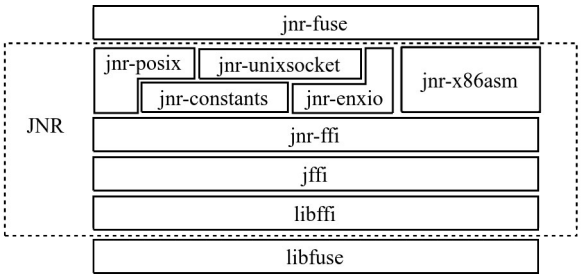



图2　JNR-FUSE框架架构

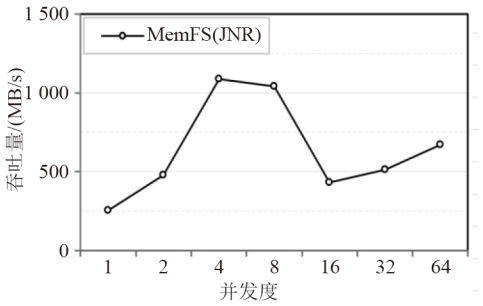
为屏蔽文件系统与底层存储介质的干扰影响,本文设计并实现了具备低管理开销的伪内存文件系统MemFS作为FUSE的目标文件系统. 为了降低MemFS本身的管理开销,MemFS存储的所有文件系统元信息皆保留在内存中,并且不实际存储文件数据,而仅记录文件长度. 为降低MemFS的内存空间开销,MemFS的写入操作不会触发任何额外的磁盘或内存数据拷贝,MemFS的读取操作会返回预先填充的虚拟数据. 因此采用MemFS作为目标文件系统能更准确地反映JNR-FUSE框架自身的性能开销情况.

在深度学习训练、大数据分析等数据密集型作业中,数据I/O的速度会极大地影响作业的执行时间. 高并发和小文件是数据密集型作业的两个重要特征. 为了提高深度学习等模型训练速度,往往会采用多个线程进行高并发读写;当数据处理对象是语音片段、图片等,往往需读写由大量小文件组成的数据集[19]. 本文分别测试并收集了JNR-FUSE框架在高并发和小文件两种场景下访问MemFS的性能数据,实验中采用的服务器软硬件配置见第5节性能评估部分的表3(如无特殊说明,下同).

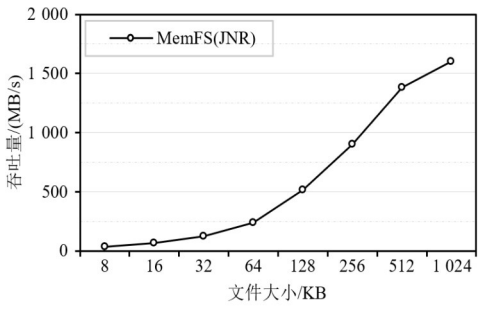
高并发场景使用不同数量的线程并发读取81 920个文件,每个文件规模固定为128 KB. JNR-FUSE的吞吐量实验结果如图3($a$)所示. 在较低线程数(不超过8线程)的情况下,吞吐量随着线程数量增多而增长. 但当线程数量从8扩展到16时,JNR-FUSE出现了严重的性能下降. 16线程的吞吐量只有8线程的40%,大于16线程时,继续增加线程数量,吞吐量有一定上升,但始终达不到8线程时的吞吐量.

小文件场景使用32个线程并发读取不同规模的大量小文件,性能实验结果如图3($b$)所示. JNR-FUSE的吞吐量随着文件规模的增大而增长. 当单个文件大小为1 MB时,吞吐量增长为1 598 MB/s;当单个文件大小减小到4 KB时,吞吐量降低为19 MB/s,仅为1 MB情况的1.2%.

以上实验结果表明,JNR-FUSE在高并发和小文件




(*a*) 高并发性能评估




(*b*) 小文件性能评估

图3 JNR-FUSE框架访问MemFS的性能评估

场景下存在严重的性能问题. JNR-FUSE在高并发场景下性能会异常下降;而在读取大量小文件时,JNR-FUSE的性能远低于大文件读取. 在深度学习训练、大数据分析等数据密集型作业中,经常会有高并发和小文件的工作负载,JNR-FUSE将面临严峻的性能挑战.

### 3.2 性能瓶颈分析

基于上节发现的JNR-FUSE性能实验结果,本节重点分析其性能瓶颈产生原因. 跨语言FUSE框架的时间开销主要可分为三部分:跨语言函数回调开销、跨语言数据拷贝开销、FUSE工作线程管理开销. 本节从以上三个维度对以JNR-FUSE为代表的跨语言FUSE框架的性能瓶颈进行分析.

#### 3.2.1 跨语言函数回调

FUSE依靠文件系统开发者定义的回调函数完成文件系统功能. FUSE库libfuse使用C语言实现,当文件系统与libfuse使用不同语言开发时,则存在跨语言函数回调的过程. 以JNR-FUSE为例,文件系统开发者用Java语言开发文件系统的各种操作,各种操作以回调函数的形式被FUSE的工作线程(Daemon线程)调用,中间依次经过FUSE层、跨语言调用层(例如JNR Native Libraries)、JVM层等,如图4所示.

从libfuse库本地工作线程(Native Thread)回调文件系统提供的JVM方法的流程包含以下步骤.

(1)线程链接(Attach)[20]:将本地线程链接到一个JVM线程,本地线程成为能被JVM统一调度管理的JVM线程,并获取能和JVM交互的JNIEnv句柄.

(2)元信息初始化:使用JNIEnv句柄获取待调用方法的MethodID等调用元信息,这些元信息在之后的调用过程中将被用于定位待调用的Java方法.

(3)函数参数初始化:使用JNIEnv句柄将C语言的本地数据类型转化为JVM数据类型,对于本地的结构体等复合类型,还需额外通过JNIEnv句柄调用JVM中的对应构造函数,以将结构体转换为JVM中对应的临时对象.

(4)跨语言调用:使用JNIEnv句柄、调用方法元信息和转换后的函数参数,通过JNI机制调用JVM中的对应方法.

(5)环境清理:清理跨语言调用使用的临时对象,释放无用的元信息、临时对象等.

(6)线程分离(Detach):将本地线程从JVM中分离,释放JNIEnv句柄,使当前线程再次成为独立调度管理的本地线程.

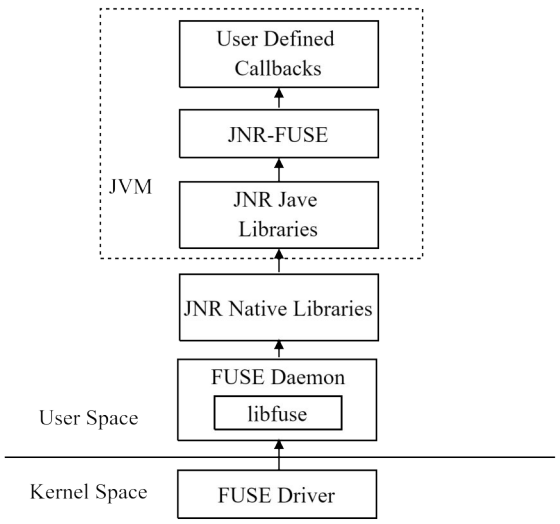



图4 基于JNR-FUSE的回调函数调用链

相比同语言内的函数调用,跨语言函数调用包含了复杂的准备步骤和清理步骤,会大幅影响系统整体性能. 当使用跨语言函数回调的方式实现FUSE时,数据密集型工作负载会频繁触发跨语言函数回调. 在大量小文件读取场景中,跨语言函数调用次数将大幅增加. 当读取总大小为10 GB的大量小文件时,触发的跨语言调用次数与文件大小的关系如表1所示. 大量跨语言函数回调显著增加了文件系统的额外开销,从而造成了JNR-FUSE在小文件场景下出现性能下降的问题.

#### 3.2.2 跨语言数据拷贝

数据拷贝也是文件系统设计中不可忽视的开销因

**表1　跨语言调用次数与文件大小关系**

| 文件大小 | 跨语言调用次数 | 相对比率 |
|---|---|---|
| 128 KB | 573 441 | 1.0 |
| 32 KB | 1 966 081 | 3.43 |
| 8 KB | 7 864 321 | 13.71 |

素．当FUSE和用户自定义文件系统以动态链接的形式使用同一个内存地址空间时，FUSE所产生的数据拷贝是简单的内存间数据移动．比如在read和write等系统调用中，文件系统仅需通过memcpy函数把文件数据移动到FUSE的缓冲区即可．但对于采用Runtime管理内存的Java语言而言，JNR-FUSE等跨语言FUSE框架存在着额外的跨语言数据拷贝问题，如图5所示．例如read系统调用要求将存储在JVM堆上空间的用户态文件系统的文件数据拷贝到由FUSE管理的堆外原生的内存缓冲区中；对于write系统调用而言，则需要相反的过程．

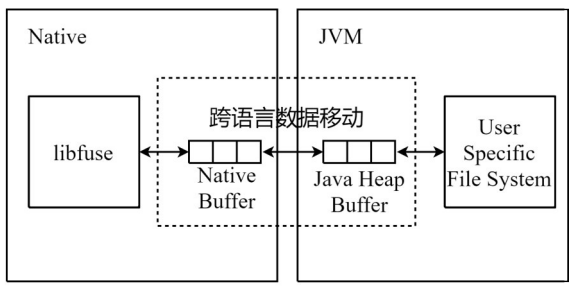
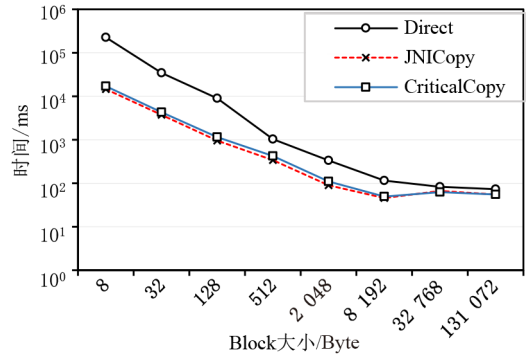



图5　跨语言数据拷贝

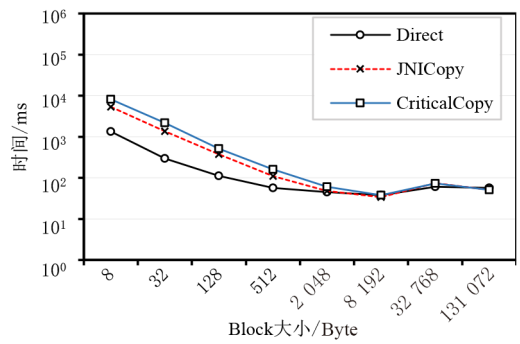
JNR-FUSE采用JNI拷贝(JNICopy)[21]的方式实现跨语言数据拷贝，其通过调用JNI方法实现堆外数据缓冲区和堆上缓冲区间的数据拷贝．每次数据拷贝均涉及一次JNI调用，采用单独的缓冲区完成，带来额外开销．因此当需要拷贝的文件越小、数量越多时，JNI调用越频繁，数据拷贝效率越差．

除了JNICopy，JVM还提供了DirectByteBuffer(Direct)[22]与临界区拷贝(CriticalCopy)[23]两种方式实现堆上空间和堆外空间的数据拷贝．Direct方式通过Java内置的直接缓冲区编程接口，在Java代码中直接读写堆外内存数据．Direct方式是Java原生支持的堆外数据访问方式，由Java管理缓冲区空间，因此其安全性最高．虽然该方式可避免JNI调用，但创建DirectByteBuffer对象会带来额外的执行开销．CriticalCopy方式通过调用JNI获取堆上缓存区的内存地址，然后使用memcpy函数直接将堆上的数据复制到堆外缓冲区中．然而此过程会阻塞JVM垃圾回收过程，并且直接暴露堆上对象的内存地址，安全性最低．

因为系统实现机制不同，上述三种数据拷贝方式的效率不同．本文通过实验测试了这三种方式通过不同容量的缓冲区拷贝10 GB数据的总耗时．由于缓冲区的创建和销毁会产生额外开销，本文分别测试了不复用缓冲区和复用缓冲区时的性能，实验结果如图6所示．实验结果表明在复用缓冲区的情况下DirectByteBuffer的数据拷贝速度最快，但DirectByteBuffer对象的创建和销毁会产生极大的额外开销，导致其在不复用缓冲区时显著慢于其他两种方式．同时，对所有的数据拷贝方式而言，增大缓冲区容量均能显著优化性能．当缓冲区容量增大到128 KB时，三者几乎不存在性能差距．相反，减小缓冲区容量会导致数据拷贝次数增多，使数据拷贝总用时增长．




(*a*) 不复用Buffer



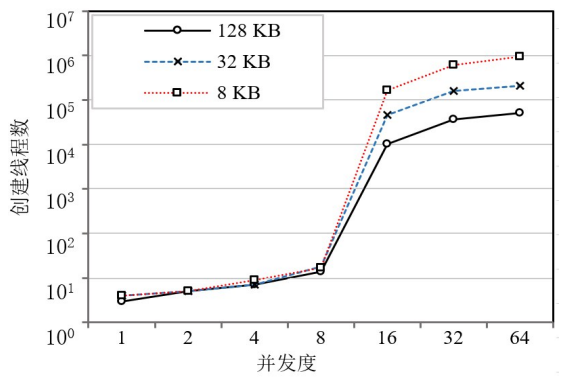

(*b*) 复用Buffer

图6　跨语言数据移动性能评估(纵轴采用对数坐标)

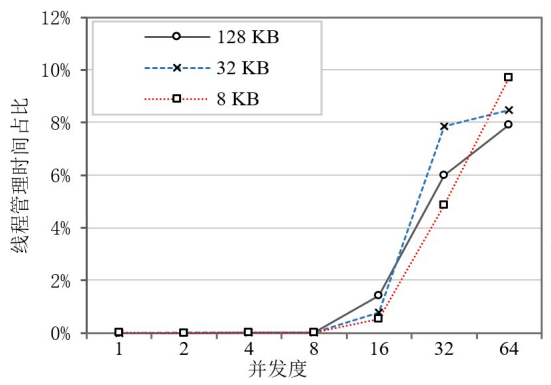
实验结果表明当缓冲区容量较小(小于128 Byte)时，三种跨语言数据移动方式存在较为明显的性能差异．但在实际应用中，这部分开销难以构成主要性能瓶颈．首先，实际应用中极少出现小于512 Byte的文件，并且数据拷贝基本都优先以大块方式进行，因此三种方式性能差距较小．其次，三种跨语言数据拷贝执行时间较短，其开销在整个跨语言FUSE过程中占比不高．如图6所示，当缓冲区容量大于128 Byte时，三种数据拷贝方式均能于10s内完成10 GB数据拷贝，数据拷贝带宽超过1 GB/s，远大于文件系统自身数据拷贝所能达到的实际吞吐量．

### 3.2.3 FUSE工作线程管理

高并发场景下的实验结果表明FUSE现有的工作线程管理机制难以适应大量线程并发读写的场景. 如图7(*a*)所示,随着读取线程数量增加,在读取并发度不超过8时,FUSE创建的线程数量很少,而当读取并发度达到16及其以上时,FUSE创建的线程数量呈指数增长. 小文件场景会进一步加重此问题. 在读取32 KB和8 KB的文件时,创建线程数量分别达到了读取128 KB文件时的4.1倍和18.6倍. 图7(*b*)进一步展示了FUSE工作线程管理开销对系统性能的影响. 当读取并发度为1~8时,因为只创建了很少的线程,线程管理时间开销基本占比可忽略不计;而当读取并发度从16增加到64时,线程管理开销占比不断增加,在8 KB文件规模和64读取并发度设置下,约有10%的执行时间被浪费在线程管理上.




(*a*) 创建线程数量(纵轴采用对数坐标)




(*b*) 线程管理开销占总时间比例

图7 FUSE工作线程管理开销分析

造成上述问题的原因在于FUSE的工作线程管理机制. 文件系统通常会读写硬盘等较慢速的设备文件,涉及大量I/O请求,导致FUSE的工作线程通常被阻塞在I/O操作上. FUSE的工作线程管理机制通过创建足够多的工作线程(可超过CPU核数的限制),以提升文件系统的总I/O吞吐量,使尽可能多的文件系统调用可以被及时处理. FUSE在初始化时仅会创建一个工作线程,当FUSE发现排队等待的FUSE请求数量过多、当前工作线程数量不足时,FUSE会创建额外的工作线程以并发处理FUSE请求. 然而在高并发场景下,当FUSE工作线程的实际处理速度低于I/O请求的到达速度时,会造成I/O请求堆积,FUSE会创建大量工作线程,这些工作线程在对应的I/O请求完成后又被销毁,从而带来不可忽视的线程管理开销.

实际上,上述问题不只限于跨语言FUSE场景,在C语言原生的FUSE调用场景也可能发生. 该问题产生的根源在于文件系统的回调函数处理速度慢于I/O请求的达到速度,造成大量I/O请求堆积. 该问题与文件系统采用的程序设计语言无关. 但因为跨语言FUSE相比原生C语言函数回调额外增加了跨语言调用开销,该问题在实际应用中变得更加显著.

从上述分析可知,降低FUSE工作线程管理开销的方法包括直接方式和间接方式两类. 在直接方式方面,可优化FUSE的工作线程管理机制,例如限制最大工作线程数量、推迟工作线程销毁时机,从而降低FUSE的工作线程管理开销. 但这类方法需要修改用户态FUSE库libfuse的源代码,需重新编译和部署操作系统环境,维护开销较高,还存在不兼容现有上层应用的风险,难以快速应用于企业的生产环境. 与上述直接方式不同,间接方式则是通过降低跨语言FUSE的回调函数开销,提升跨语言FUSE请求的处理速度,降低I/O请求堆积的风险,从而间接减少FUSE工作线程管理机制对整体性能的负面影响.

### 3.3 性能分析总结

上述的性能分析剖析了现有跨语言FUSE框架JNR-FUSE在高并发和小文件场景下存在的性能瓶颈及其产生的原因:第一,JNR-FUSE会频繁触发跨语言函数回调,在小文件场景下会大幅增加函数回调次数;第二,堆内内存和堆外内存间频繁的数据拷贝开销在极端小文件场景(文件小于128 Byte)下也不容忽略;第三,FUSE本身的工作线程管理机制会导致在高并发场景下易造成工作线程的频繁创建和销毁.

基于上述分析,本文提出优化跨语言FUSE框架的性能可遵循的三个优化方向:(1)设计低开销的跨语言函数回调机制,从而实现对大量小文件的高效访问;(2)选择低开销且高安全性的跨语言数据拷贝方式,从而实现对大量文件的高吞吐、高安全访问;(3)设计轻量级的工作线程管理机制,从而降低高并发场景下工作线程创建与销毁所带来的额外开销. 在企业生产环境下,因为FUSE库libfuse通常以操作系统内置库的形式提供,在不修改libfuse库源代码的情况下,FUSE库的工作线程管理机制难以修改,因此在设计系统兼容性

强的跨语言FUSE框架时应更关注前两个优化方向.

## 4　JNI-FUSE框架设计实现

根据跨语言FUSE框架设计的优化方向,本文面向以Java语言开发的分布式文件系统,设计了跨语言函数回调开销小、跨语言数据拷贝安全性高的跨语言FUSE框架JNI-FUSE,以加速数据密集型作业在分布式文件系统上的运行速度.

本文设计实现的JNI-FUSE框架原型总共包含约3 700行代码,其中包括C语言实现的本地代码约1 400行和Java语言实现的JVM接口代码约2 300行. JNI-FUSE的源代码①已被知名分布式文件系统Alluxio接受并集成作为Alluxio默认的FUSE实现方式.

### 4.1　框架架构

作为跨语言FUSE框架,JNI-FUSE连通了操作系统FUSE库libfuse的本地C语言环境与目标分布式文件系统的Java语言环境,使FUSE库libfuse可以通过JNI-FUSE框架的跨语言函数回调机制调用分布式文件系统的相关功能,而分布式文件系统则无须感知libfuse库与内核FUSE驱动的技术细节即可通过JNI-FUSE提供POSIX文件访问接口.

#### 4.1.1　架构设计

JNI-FUSE由本地代码层和JVM层两个模块组成,其架构如图8所示. 本地代码层向下和操作系统提供的libfuse库对接,向上则调用JVM层文件系统提供的Java语言编程接口,以实现相关的文件操作. JVM层则向下对接本地代码层,向上为用户提供统一便捷的文件系统接口AbstractFuseFileSystem. 该接口将文件系统的典型操作抽象为read,write,release等方法,与POSIX文件访问接口保持一致. 分布式文件系统的开发者需要继承该接口,并在该接口的各方法中调用分布式文件系统的对应编程接口,从而对接各种用户态分布式文件系统. JNI-FUSE实现了FUSE库与分布式文件系统的编程接口之间的解耦,分布式文件系统的开发者无须了解FUSE编程的具体细节即可通过JNI-FUSE框架对接操作系统的FUSE库,从而提供POSIX文件接口的访问能力.

根据第3.3节的性能瓶颈分析,跨语言FUSE的可能性能优化方向包括降低跨语言函数回调开销、降低跨语言数据拷贝开销与改进工作线程管理机制. 其中前两个优化方向只与跨语言FUSE框架本身相关,通过改进跨语言FUSE框架自身的架构与实现方法即可达成. 而第三个优化方向与操作系统提供的libfuse库实现相关,需要修改与重新编译相关系统库,使其难以部署于注重稳定的企业生产环境.

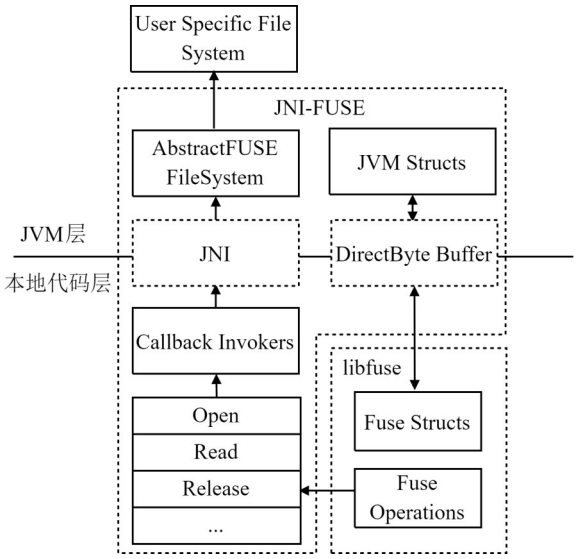



图8　JNI-FUSE框架

因此为了避免修改操作系统提供的libfuse库、提升JNI-FUSE框架对不同libfuse库版本的适应性,本文将JNI-FUSE框架设计为独立于操作系统libfuse库的第三方软件模块,通过延迟分离、元信息缓存等技术降低跨语言函数回调的执行开销,通过降低I/O请求堆积的风险,从而间接减少FUSE工作线程管理机制的负面影响. 对跨语言FUSE的性能优化与对FUSE工作线程管理机制的优化之间是互相独立的,可以互相叠加. 本文的研究主题更关注于跨语言FUSE特有的性能优化问题.

JNI-FUSE框架的独立性使得本文将跨语言FUSE性能优化的重点聚焦于降低跨语言函数回调的执行开销,通过减少I/O请求堆积的发生来间接降低FUSE工作线程管理机制的负面影响.

#### 4.1.2　工作流程

JNI-FUSE处理一次来自应用程序的POSIX文件系统调用的工作流程如图9所示. 运行在用户空间的应用程序通过POSIX文件系统调用(例如read,write等)的方式启动一次基于FUSE的POSIX I/O请求. 应用程序的系统调用由运行在内核中的FUSE驱动处理,FUSE驱动将I/O请求发送至运行在用户态的libfuse库的FUSE工作线程处理. libfuse库的FUSE工作线程调用JNI-FUSE框架本地代码层中的对应函数,将文件系统的I/O请求转发至JNI-FUSE框架处理.

具体地,libfuse库的工作线程首先通过线程链接步骤(步骤1)将FUSE工作线程链接(Attach)为JVM线程[20]. 之后,元信息初始化步骤(步骤2)会获取跨语言函数调用所需的调用方法元信息(包括JVM对应方法

① https://github.com/Alluxio/alluxio/tree/release-2.6.0/integration/jnifuse

的MethodID等). 跨语言数据拷贝步骤(步骤3)将C语言缓冲区内的数据拷贝至Java语言的内存缓冲区中. 基于跨语言函数调用元信息与Java内存缓冲区,JNI-FUSE框架通过JNI机制调用JVM层由分布式文件系统开发者实现的AbstractFuseFileSystem接口的相应方法(步骤4),执行流转入JVM层. JVM层在AbstractFuseFileSystem接口方法中具体地调用目标分布式文件系统的相关函数,从而完成对分布式文件系统的操作. AbstractFuseFileSystem接口调用完成之后,JNI-FUSE框架清理跨语言函数调用产生的临时数据与元信息(步骤5),最后将本地工作线程从JVM分离(步骤6).

一次完整的跨语言FUSE函数回调链路很长,这个过程存在大量冗余的准备和清理工作. 本文通过引入延迟分离(Defer Detach)和元信息缓存(Meta Cache)技术,跳过跨语言FUSE函数回调程中回调准备和清理的相关步骤(包含链接线程、元信息初始化、环境清理与线程分离等四个步骤),从而降低跨语言FUSE的函数回调开销.

### 4.2 延迟分离优化

延迟分离优化主要针对跨语言FUSE过程中执行开销较高的线程链接与线程分离步骤,如图9所示. 这两个步骤的执行开销主要来自JVM的线程管理开销. 当一个本地线程需要从C语言函数中调用Java的方法时,首先需要通过JNI提供的线程链接机制将本地线程链接(Attach)为一个Java线程,从而获得JNIEnv句柄并通过该句柄调用Java方法. 在线程链接过程中,JVM需要执行一系列JVM线程初始化操作(例如关闭浮点数异常、调整线程优先级等),以将本地线程包装为一个JVM线程,由JVM运行时统一管理;线程分离过程则需要销毁对应的JVM线程,从而实现本地工作线程与JVM线程的分离(Detach). JVM线程的初始化与销毁给两个步骤带来了显著的额外开销,当跨语言文件系统函数调用过程(步骤4)执行开销较小时,JVM线程链接与分离带来的额外开销将变得更加显著.

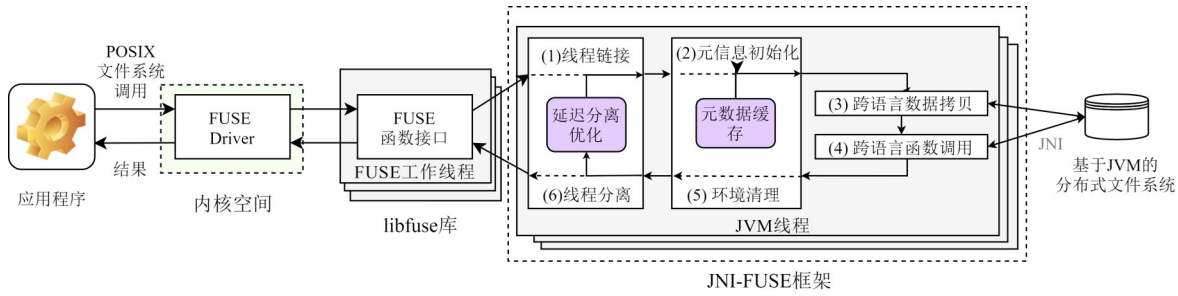



图9 JNI-FUSE框架工作流程

延迟分离技术的总体思路是在跨语言调用时推迟本地线程从JVM分离的时间,以便在后续的跨语言函数回调中复用已链接的线程,跳过线程链接(步骤1)与线程分离(步骤6)步骤,如图9所示. 具体地,在从libfuse库本地线程首次进行跨语言函数回调的过程中,在线程分离步骤暂不释放当前的JVM线程,而是保持当前线程与JVM的链接关系,将线程链接产生的JNIEnv句柄[20]缓存在线程局部变量中. 在之后的跨语言函数回调过程中,线程链接步骤直接读取缓存的JNIEnv句柄,跳过JVM线程链接过程. 跨语言函数调用之后的线程分离步骤则同样被跳过. 采用延迟分离技术后,JNI-FUSE只会在第一次跨语言函数回调的时将本地线程链接到JVM,直到本地线程被销毁时才从JVM线程分离出来,从而大幅降低JVM线程链接与分离带来的额外开销.

延迟分离技术要求各线程缓存的JNIEnv句柄应在FUSE工作线程销毁之前执行释放句柄的操作,以保证JVM线程管理机制正常工作. 但FUSE工作线程的销毁由libfuse库触发执行,JNI-FUSE框架难以控制与掌握. 因此如果采用全局变量缓存各线程的JNIEnv句柄,无法保证JNIEnv句柄随工作线程的结束而正确释放. 为了解决延迟分离过程中JNIEnv句柄的释放问题,JNI-FUSE框架采用了pthread多线程编程技术提供的线程特有数据(thread-specific data)机制来管理每个线程缓存的JNIEnv句柄. JNI-FUSE利用该机制将每个工作线程的JNIEnv句柄声明为该线程特有数据对象,并为该数据对象注册特定析构函数. 在工作线程被操作系统销毁之前,pthread会自动调用线程特有数据对象的注册析构函数,从而正确释放JNIEnv句柄.

延迟分离技术因为缓存JNIEnv句柄、保持JVM线程链接,而额外增加了与工作线程数成正比的空间开销,但该部分额外空间开销的总量较小. 该空间开销主要由两部分组成:(1)工作线程缓存的JNIEnv句柄所占空间,(2)工作线程对应的JVM内部线程对象所占空间. 具体而言,每个JNIEnv句柄包含一系列JVM函数指针与数据指针信息. 此外,工作线程链接到JVM时,JVM会为工作线程创建对应的JVM线程对象,并在JVM栈区与堆区申请与初始化相应的内存空间. 延迟

分离优化会一直保持工作线程与JVM的链接状态. 上述JVM内部线程的存储开销在工作线程销毁后即被释放. 在实际应用中,JNI-FUSE在高并发场景下通常产生20个左右的工作线程(详见第5.2.1节的实验),单个线程延迟分离优化所增加的平均物理内存开销为636.52 KB,其产生的总物理内存开销约12.43 MB,因此额外存储空间开销不大.

### 4.3　元信息缓存

元信息缓存技术主要针对跨语言函数回调过程中的元信息初始化(步骤2)与环境清理(步骤5)步骤. 元信息初始化步骤需要为跨语言函数调用准备调用元信息,包括获得JVM中对应方法的MethodID等. 因为同一个Java方法的JNI调用元信息是静态的、不随线程而改变,所以当同一个文件系统操作被应用程序重复调用多次时,重复的元信息初始化与环境清理将带来冗余的执行开销.

元信息缓存技术的总体思路是在框架初始化时缓存各种文件系统操作(例如open,read,release等)所需的元信息,在后续的跨语言函数回调过程中直接复用相关信息,从而跳过元信息初始化与环境清理的步骤. 因为各方法的调用元信息是用户不可更改的只读信息,且元信息只与回调函数相关,与工作线程或JVM线程无关,因此跨语言函数调用元信息只须初始化一次,即可在多个工作线程中复用,而无须考虑多线程访问一致性的问题.

具体地,JNI-FUSE框架在框架初始化时,会为用户实现的AbstractFuseFileSystem接口内每种支持的POSIX文件系统调用预取对应的跨语言函数回调相关元信息,并缓存在内存中(图10). 缓存的调用元信息在所有线程之间共享,从而避免多个线程之间冗余的元信息获取开销. 在应用程序的一次跨语言FUSE函数调用的过程中,JNI-FUSE框架在元信息初始化步骤中从内存中提取对应的跨语言函数调用元信息. 基于提取出的元信息,JNI-FUSE即可利用JNI机制发起跨语言函数调用.

值得注意的是,本文提出的元信息缓存技术缓存的并不是文件系统的元信息,而是跨语言函数调用的元信息(例如MethodID、调用句柄等). 文件系统自身的元信息缓存通常由文件系统客户端管理,其与JNI-FUSE框架的元信息缓存技术是独立的,两者可以联合使用以提升文件系统的跨语言FUSE性能.

元信息缓存技术同样引入了额外的内存空间开销. 元信息缓存需存储各类文件系统调用对应JVM方法的MethodID信息,其存储开销与调用的JVM方法数量成正比. 对于单个JVM方法而言,本文实验所采用的64位HotSpot JVM中,每个MethodID包含了一个指向JVM内部方法对象的指针,占用8个Byte. JNI-FUSE调用的JVM方法数量的上限由AbstractFileSystem接口包含的POSIX文件系统调用数量决定,而实验所采用的v2.9.5版本的libfuse库最多支持44类文件系统调用,因此元信息缓存所需的额外空间至多为352 Byte,开销很小.

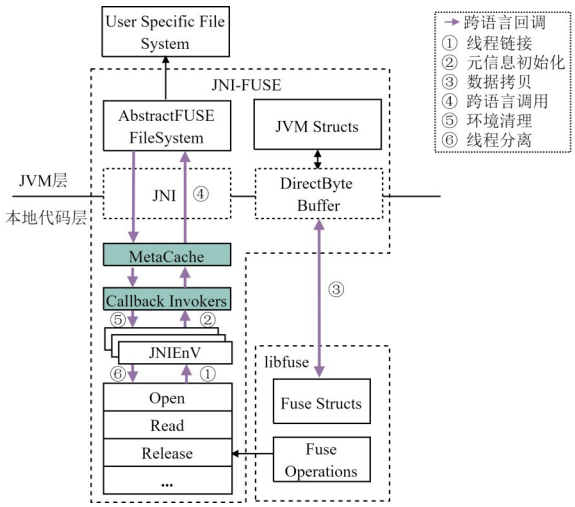



图10　元信息缓存优化

### 4.4　跨语言数据拷贝机制选择

目前在Java语言下可实现跨语言数据拷贝的机制有JNICopy,CriticalCopy和DirectByteBuffer三种. 表2从安全性、通用性与性能三个方面对比了三种跨语言数据拷贝方式. 在三种方式之中,DirectByteBuffer的安全性最高,DirectByteBuffer是Java内置的数据类型,无须用户直接管理缓冲区空间,可在规避内存泄漏等难以察觉和调试的问题的同时,避免直接访问指针可能带来的访问错误. 此外,DirectByteBuffer的通用性最好,其屏蔽了底层硬件和软件环境的细节,尽可能地避免了因硬件和软件环境变化所带来的适配问题,保持了JVM的"Write once,run anywhere"跨平台优势.

作为对比,JNR-FUSE等框架采用了JNICopy等通用性较弱的数据拷贝机制. 为保持跨语言FUSE框架的跨平台通用性,JNR-FUSE等框架额外封装了多层胶水代码,最终带来较大性能损失,得不偿失. 从性能角度比较,第3.2.2节的实验结果表明:DirectByteBuffer的数据拷贝性能在数据块较小时稍弱于JNICopy和CriticalCopy两种方式;但在FUSE的应用场景中数据多采取较大的数据缓冲区(128 KB),DirectByteBuffer与其他跨语言数据移动方式相比并没有明显性能差距. 综合考虑性能、安全性和通用性,JNI-FUSE选择采用DirectByteBuffer实现跨语言数据拷贝.

**表2　不同跨语言数据拷贝机制比较**

| 跨语言数据拷贝机制 | 安全性 | 通用性 | 性能 |
|---|---|---|---|
| JNICopy | 中 | 低 | 高 |
| CriticalCopy | 低 | 中 | 高 |
| DirectByteBuffer | 高 | 高 | 中 |

### 4.5 跨语言适用性分析

本文所提出的延迟分离、元信息缓存优化方法以及选择的跨数据拷贝机制适用于所有基于JVM的编程语言(例如Java,Scala,Kotlin等)开发的分布式文件系统,包括大数据软件栈中广为使用的HDFS,Alluxio等. 其原因是基于JVM的文件系统均需要借助JNI机制,实现libfuse库回调JVM中分布式文件系统提供的相应方法的功能. 本文提出的优化方法针对JNI调用中相关初始化与清理步骤设计,适用于所有基于JVM的编程语言,具备较好的跨语言适用性.

## 5 性能评估

### 5.1 实验设置

本节从JNI-FUSE框架(本文方法)自身性能、文件系统端到端读取性能、应用层深度学习训练性能三个方面评估JNI-FUSE的性能. 本节采用目前最为成熟稳定、性能最佳、使用最广的Java端FUSE框架JNR-FUSE[13]作为对比框架. 为保证测试的准确性和公平性,所有测试均在FUSE的direct_io模式(不使用kernel cache)下进行. 如无特别说明,保留其余参数为默认配置.

考虑到很多大数据深度学习作业都基于容器云平台进行,本节实验在Kubernetes[24]集群中进行,测试程序运行在Docker容器中,通过CSI[25]对接到FUSE. 第5.2节和第5.3节的实验在无GPU的普通计算服务器上进行测试,第5.4节的深度学习训练实验在一台具有两张Tesla V100 GPU的深度学习服务器上进行测试,两台计算服务器的配置如表3和表4所示.

**表3 普通计算服务器软硬件配置信息**

| 参数 | 配置信息 |
|---|---|
| CPU | Intel(R) Xeon(R) Silver 4114 CPU @ 2.20 GHz (40 cores) |
| Memory | 96 GB DDR4 |
| Disk | 240 GB SSD |
| Network Bandwidth | 10 Gbps |
| OS | CentOS Linux release 7.6.1810 |
| JVM Version | Java 1.8.0 |
| FUSE Version | 2.9.5 |
| Alluxio Version | 2.5.0 |

### 5.2 框架性能评估

为评估JNI-FUSE框架(本文方法)自身带来的额外开销,本文以第3.1节所述的伪内存文件系统MemFS作为FUSE的目标文件系统,分别基于JNR-FUSE[13]和JNI-FUSE框架实现了该文件系统的POSIX接口,分别记为MemFS(JNR-FUSE)和MemFS(JNI-FUSE).

**表4 GPU深度学习服务器软硬件配置信息**

| 参数 | 配置信息 |
|---|---|
| CPU | Intel(R) Xeon(R) Silver 4214 CPU @ 2.20 GHz (48 cores) |
| GPU | Tesla V100×2 |
| Memory | 256 GB DDR4 |
| Disk | 4.7 TB HDD |
| Network Bandwidth | 10 Gbps |
| OS | CentOS Linux release 7.6.1810 |
| JVM Version | Java 1.8.0 |
| FUSE Version | 2.9.5 |
| Alluxio Version | 2.5.0 |
| CUDA Version | 11.0 |
| PyTorch Version | 1.7.1 |

#### 5.2.1 读取性能评估

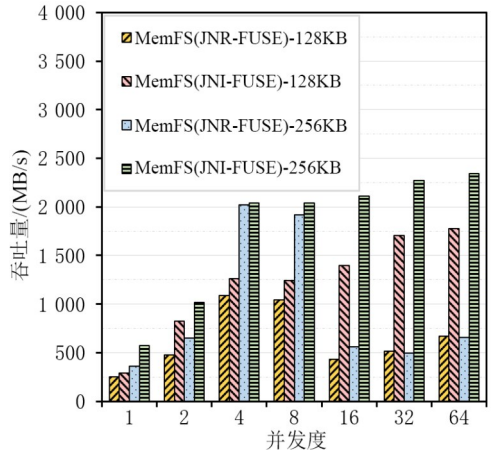
本文在普通计算服务器上分别对比了MemFS(JNR-FUSE)和MemFS(JNI-FUSE)在高并发和小文件场景下的文件读取性能.

高并发场景中,使用不同数量的线程并发从MemFS读取81 920个文件,每个文件大小先后设置为128 KB和256 KB,总大小为10 GB和20 GB. 高并发场景的读取性能测试结果如图11($a$)所示. 随着读取线程数的增长,MemFS(JNR-FUSE)的吞吐量先快速增长,但当并发度超过8时,MemFS(JNR-FUSE)的吞吐量出现骤降,尤其是在并发度从8增加到16时,吞吐量降低了50%以上,其主要受大量工作线程管理开销影响. 而MemFS(JNI-FUSE)的吞吐量则随并发线程数的增长保持增长趋势. 当每个文件为128 KB时,MemFS(JNI-FUSE)相比MemFS(JNR-FUSE)带来了平均约2.06倍、最高约3.32倍的性能提升.

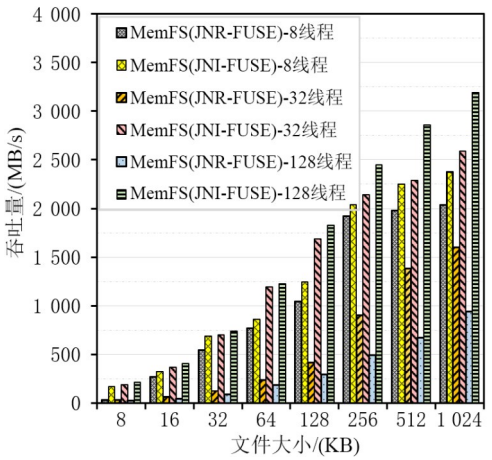
在4线程当每个文件为256 KB时,MemFS(JNI-FUSE)相比MemFS(JNR-FUSE)带来了平均约3.71倍、最高约5.78倍的性能提升. MemFS(JNR-FUSE)和MemFS(JNI-FUSE)分别在4线程与64线程并发度下取得最大吞吐量,MemFS(JNI-FUSE)的最大吞吐量相比MemFS(JNR-FUSE)在128 KB和256 KB文件大小下分别提高63.5%和16%. 在高并发场景下,JNI-FUSE框架显著降低了跨语言函数回调开销,使MemFS(JNI-FUSE)可以以较少的FUSE工作线程数处理高并发读取,大幅削弱了FUSE工作线程管理所带来的影响,从而获得比JNR-FUSE更好的性能.

本文进一步通过8线程和64线程并发度读取大量32 KB小文件的场景,对比了JNI-FUSE和JNR-FUSE所使用的FUSE工作线程数量的差异,实验结果如图12所示. 在8线程和64线程并发度下,JNI-FUSE使用的工作线程数全程分别稳定在10和20左右. 而JNR-FUSE

使用的工作线程数在8线程并发度下同样稳定在10个工作线程,但在64线程并发度下则呈现大幅波动,每次波动意味着大量工作线程的创建和销毁,其最高时甚至同时存在120个工作线程. 实验结果表明JNI-FUSE比JNR-FUSE的读取速度更快,在同样读取总规模为10 GB文件的负载下,在8线程和64线程并发度下JNI-FUSE均在第50 s时提前完成测试,而JNR-FUSE则分别需要60 s与256 s才能完成,JNI-FUSE的性能显著优于JNR-FUSE,尤其是在高并发场景下.




(*a*) 高并发读取性能评估




(*b*) 小文件读取性能评估

图11　不同跨语言FUSE框架下MemFS文件系统的读取性能评估

在小文件场景中,本文在保持MemFS数据总量为10 GB的情况下生成多组不同规模的小文件,并分别使用8个、32个和128个线程并发读取. 图10(*b*)展示了小文件读取场景的性能测试结果. 随着单个文件规模的增长,MemFS(JNI-FUSE)和MemFS(JNR-FUSE)的吞吐量都有所增长,且在较大文件规模下MemFS(JNI-FUSE)的吞吐量大幅领先于MemFS(JNR-FUSE). 当文件越小时,MemFS(JNI-FUSE)的性能优势更加明显. 使用8个线程并发读取时,MemFS(JNI-FUSE)取得了平均1.16倍、最高约5.44倍的性能提升;使用32个线程并发读取时,MemFS(JNI-FUSE)取得了平均约3.98倍、最高约5.77倍的性能提升;使用128个线程并发读取时,MemFS(JNI-FUSE)取得了平均约6.04倍、最高约9.03倍的性能提升. 读取的文件越小,文件系统元数据操作在总耗时中的比例就越高,跨语言FUSE框架本身的性能开销所占比重越大,JNI-FUSE相比JNR-FUSE的优势更加明显.

### 5.2.2　写入性能评估

本文在普通服务器上分别对比了MemFS(JNI-FUSE)与MemFS(JNR-FUSE)在高并发和小文件场景下的写入性能. MemFS初始时不存储任何数据,文件数量为0. 在随后的测试中,测试用写入线程将会持续向MemFS中写入文件. 在高并发场景中,分别限制单个文件大小为128 KB和256 KB,总量对应为10 GB和20 GB;在小文件场景中,固定使用32个和128个写入线程,每次写入不同大小的文件,保持MemFS数据总量为10 GB.

图13(*a*)展示了高并发写入场景的性能测试结果,其整体增长趋势类似于高并发读取场景;图13(*b*)展示了小文件写入场景性能测试结果. 小文件写入时,文件越小两者性能差距反而越小,这与小文件读取相反. 这是因为写入操作是元数据高度密集型操作[9],性能瓶颈主要在FUSE层,无法体现JNR-FUSE和JNI-FUSE框架本身的性能差距. 在高并发写入场景中,当每个文件为128 KB时,MemFS(JNI-FUSE)相比MemFS(JNR-FUSE)带来了平均约1.15倍、最高约1.36倍的性能提升;当每个文件为256 KB时,MemFS(JNI-FUSE)相比MemFS(JNR-FUSE)带来了平均约3.26倍、最高约5.07倍的性能提升. 从两个框架在实验中所取得的最高写入吞吐量来看,MemFS(JNI-FUSE)在128 KB和256 KB文件大小下的最高吞吐量相比MemFS(JNR-FUSE)分别提高了5.0%和256.8%.

在小文件写入场景中,使用8个线程并发写入时,MemFS(JNI-FUSE)相比MemFS(JNR-FUSE)带来了平均约2.41倍、最高约4.86倍的性能提升;使用32个线程并发写入时,MemFS(JNI-FUSE)相比MemFS(JNR-FUSE)带来了平均约1.49倍、最高约3.08倍的性能提升;使用128个线程并发写入时,MemFS(JNI-FUSE)相比MemFS(JNR-FUSE)带来了平均约5.02、最高约6.59倍的性能提升.

### 5.2.3　优化技术效果评估

JNI-FUSE框架采用了延迟分离优化与元信息缓存优化等两种优化技术来提升性能. 本节通过比较优化前后跨语言FUSE过程的性能,以评估本文所提优化技

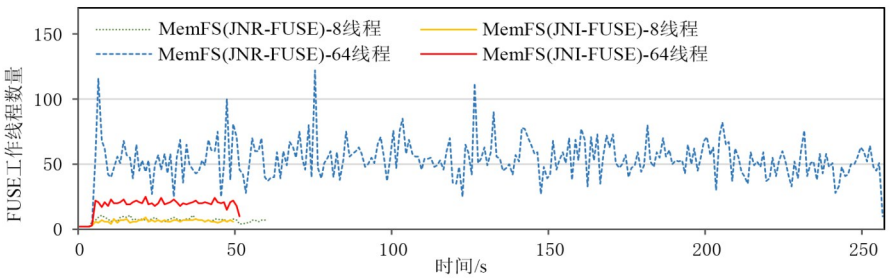
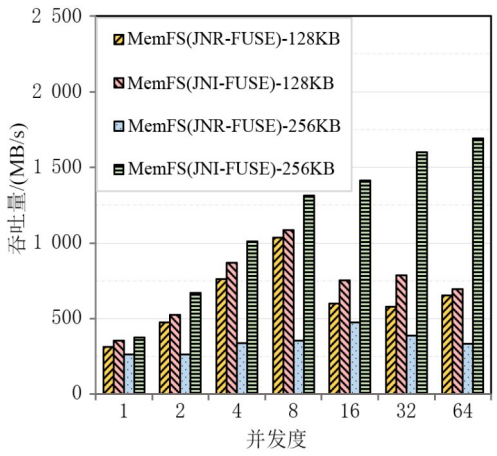
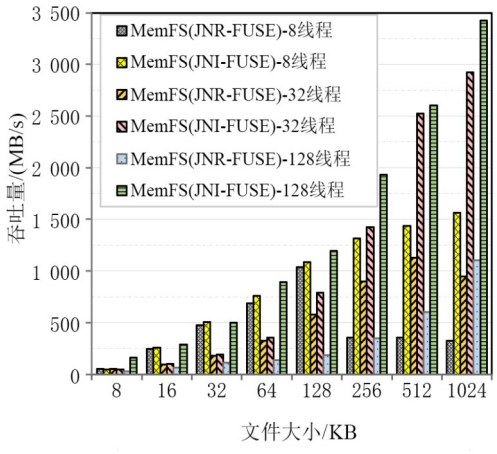



图 12 不同跨语言 FUSE 框架下 FUSE 工作线程数量对比




(*a*) 高并发写入测试




(*b*) 小文件写入测试

图 13 不同跨语言 FUSE 框架下 MemFS 文件系统的写入性能评估

术的有效性. 以是否采用延迟分离技术与元信息缓存技术为标准,跨语言函数回调方式可以划分为 Baseline,Cache,Defer 和 Cache+Defer 四种. Baseline 方式采用 JNI 提供的基本跨语言调用过程,包含工作流程中步骤 1 至步骤 6 的所有过程. Cache 方式在 Baseline 的基础上使用元信息缓存优化. Defer 方式在 Baseline 的基础上,额外加入延迟线程分离优化. Cache+Defer 方式在 Baseline 的基础上,同时使用元信息缓存优化和延迟线程分离优化.

图 14 对比了在不同并发度下,分别采用以上四种方式执行 10 万次跨语言调用的总体耗时. 图 14 实验结果表明不同方式的总体耗时差异较大:元信息缓存(Cache)和延迟分离优化(Defer)都能提升跨语言调用的性能,延迟分离优化效果最为明显,可带来两个数量级的性能提升. 对比 Baseline,元信息缓存和延迟分离优化联合使用能够带来平均约 150 倍的性能提升. 现有 JNR-FUSE 采用的跨语言函数回调是未充分优化的,它并未缓存元信息和延迟线程分离,因此只能达到 Baseline 的性能. JNI-FUSE 通过优化跨语言函数回调机制,可获得显著的性能提升.

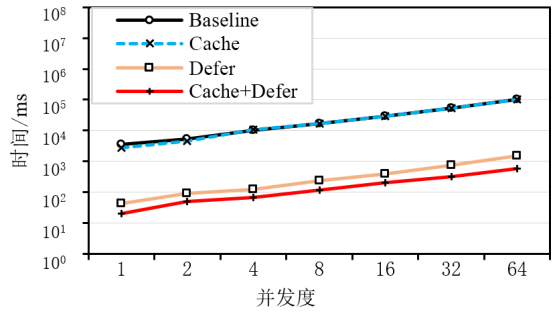



图 14 跨语言函数回调优化技术性能对比(纵轴采用对数坐标)

### 5.3 分布式文件系统端到端性能评估

本文在普通服务器上测试了广为使用的开源分布式文件系统 Alluxio 基于 JNI-FUSE 提供的 POSIX 接口的端到端性能. 本文将 JNI-FUSE 集成到 Alluxio 中(记为 Alluxio(JNI-FUSE)),并采用 Alluxio 原本基于 JNR-FUSE 的实现[26](记为 Alluxio(JNR-FUSE))作为对比. 实验中采用大型真实图片数据集 SVHN[27]的 extra 部分和大型真实图片数据集 COCO[28]的测试集部分作为工作负载. SVHN 的 extra 部分包含 202 355 张图片文件,总大小约 3.6 GB,平均每张图片 19 KB,其中最小的图片不到 1 KB. COCO 的测试集部分包含 40 504 张图片文件,总大小约 6.2 GB,平均每张图片大小 160 KB,其中最小的图片不到 8 KB,最大图片接近 900 KB. Alluxio Worker 缓存大小设置为 30 GB,因此这两个数据集都可以完全缓存在内存中. 为了加快 Alluxio 读取数据的速度,本文开启了 Alluxio 的元数据缓存优化,以减

少元数据操作RPC请求.

在测试时,本文预先将两种数据集分别加载进Alluxio中,接着分别用Alluxio(JNR-FUSE)和Alluxio(JNI-FUSE)把数据集挂载到本地文件系统以测试读取性能,测试结果如图15所示. 在并发度不大于8时,两者的吞吐量基本与并发度呈线性增长关系,两个框架的性能基本持平;当并发度大于8时,Alluxio(JNR-FUSE)出现了严重的性能下降,而Alluxio(JNI-FUSE)仅是降低了吞吐量随并发度的增长速度. 因为JNI-FUSE比JNR-FUSE更为高效稳定,最终在读取SVHN的extra部分时,Alluxio(JNI-FUSE)相比Alluxio(JNR-FUSE)取得了平均2.71倍、最高7.94倍的性能提升;读取COCO的测试集部分时,对比Alluxio(JNR-FUSE),Alluxio(JNI-FUSE)取得了平均1.90倍、最高3.90倍的性能提升. 在两个数据集上,Alluxio(JNR-FUSE)均在并发度为8时取得最大吞吐量,而Alluxio(JNI-FUSE)则均在64线程并发时取得最大吞吐量,Alluxio(JNI-FUSE)的最大吞吐量相比Alluxio(JNR-FUSE)分别提升19.4%和64.3%.

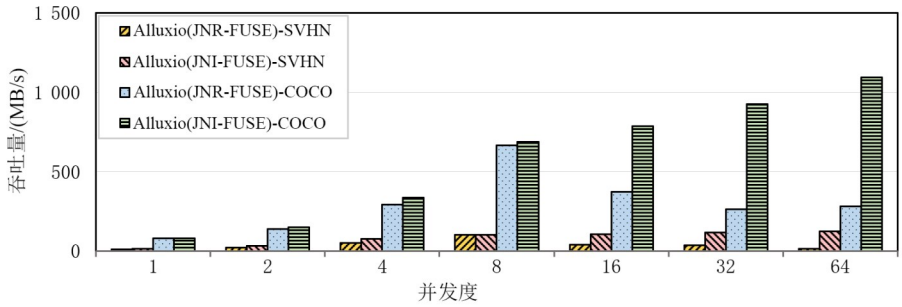


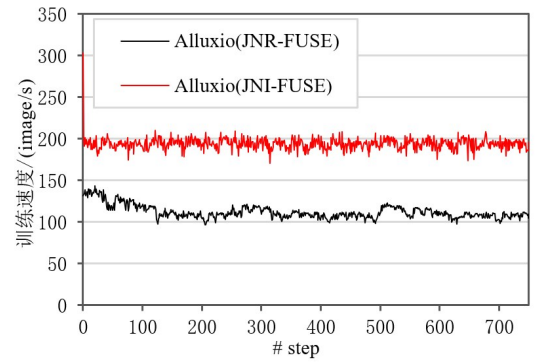

图15　Alluxio端到端性能测试

### 5.4　深度学习训练应用性能评估

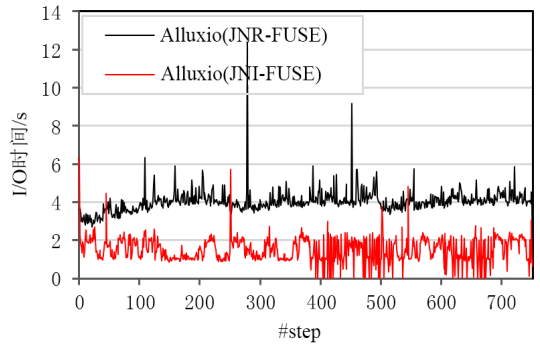
最后本文在Kubernetes环境下的GPU计算服务器上利用大规模深度学习训练应用评估了本文优化后的跨语言FUSE框架对整个训练应用的性能影响. 本文使用深度学习训练框架PyTorch[29]官方提供的PyTorch容器,运行其自带测试Demo程序[30]对ImageNet数据集[31] ILSVRC子集的CLS-LOC部分进行训练;使用深度学习框架Horovod[32]官方提供的Horovod容器,运行其自带测试Demo程序[33]对CIFAR-10数据集[34]进行训练. ImageNet和CIFAR-10都是大规模图像分类数据集. ImageNet数据集ILSVRC子集的CLS-LOC部分总大小约150 GB,包含了约120万张图片. CIFAR-10数据集处理为Horovod自带测试Demo所需的格式后,总大小为7.2 GB,包含了60 000张图片. 本文将两种数据集分别存入Alluxio分布式文件系统中,并分别利用Alluxio(JNR-FUSE)和Alluxio(JNI-FUSE)挂载Alluxio中的数据集到本地文件系统,然后指定挂载点为数据源,并进行深度学习训练. 深度学习框架在训练过程中采用的并发读取线程数通常由硬件配置(例如CPU数量)、用户给出的训练参数(例如batch size等)、深度学习框架配置(例如最大并发线程数等)等共同确定.

在PyTorch的实验中,训练过程中采用的数据并发读取线程数由PyTorch框架根据硬件配置、训练参数等计算得到,在本实验中为8个并发线程. 因为完整的训练过程耗时很长,但前3个Epoch训练过程已经足够稳定,之后的Epoch不会产生大的性能差异,因此在评估中取前3个Epoch的性能进行对比. 图16(*a*)比较了每10步迭代的平均训练速度,图16(*b*)对比了每10步迭代中花费在数据读取上的平均耗时. 在整个训练过程中,JNR-FUSE平均每步训练有约88.1%的时间花费在了数据I/O上,而JNI-FUSE将这部分数据I/O的时间降低到了55.6%,JNI-FUSE的每步数据I/O时间比JNR-FUSE更短,因此使用JNI-FUSE时GPU资源利用率更高,训练速度更快. 在深度学习训练任务中,JNI-FUSE相比JNR-FUSE平均减少了62.7%的数据I/O时间,最终取得平均1.73倍的训练速度提升.

在Horovod测试中,选取前100次迭代的性能进行对比,控制数据并发读取线程数分别为8线程与32线程. 图17(*a*)比较了每步迭代的平均训练速度,图17(*b*)对比了每步迭代中花费在数据读取上的平均耗时. 在整个训练过程中,当并发预取线程数分别为8和32线程时,JNR-FUSE平均每步训练有分别约14.1%和59.0%的时间花费在了数据I/O上,而JNI-FUSE将这部分数据I/O的时间降低到了7.0%和11.1%,JNI-FUSE的每步数据I/O时间比JNR-FUSE更短,因此使用JNI-FUSE时GPU资源利用率更高、训练速度更快. 在深度学习训练任务中,8线程和32线程并发读取情况下,JNI-FUSE相比JNR-FUSE平均分别减少了54.1%和86.7%的数据I/O时间,最终取得了平均1.06倍和1.50倍的训练速度提升.

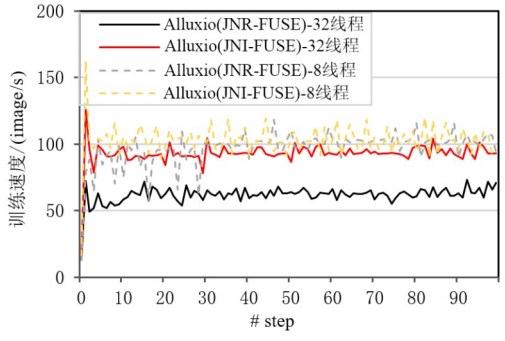



(*a*) 训练速度

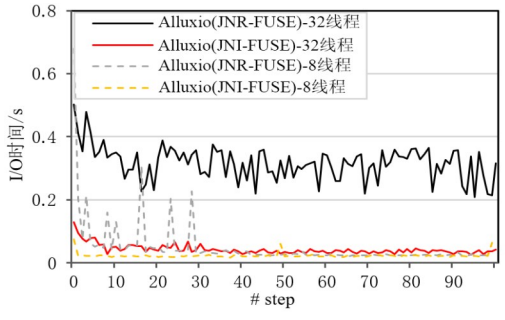



(*b*) I/O耗时

图16 PyTorch自带Demo程序在ImageNet数据集ILSVRC子集的CLS-LOC部分上的训练性能对比




(*a*) 训练速度




(*b*) I/O耗时

图17 Horovod自带Demo程序在CIFAR-10数据集上的训练性能对比

## 6 总结

现在很多大数据文件存储系统依赖FUSE提供POSIX兼容性,但跨语言FUSE框架的性能瓶颈缺乏系统性的研究与分析. 本文系统性地评估分析了现有分布式文件系统采用的跨语言FUSE框架JNR-FUSE的性能问题,进而总结出提升跨语言FUSE框架性能的可能优化方向,包括降低跨语言函数回调开销、跨语言数据拷贝开销与优化FUSE工作线程管理机制等.

基于上述优化方向,本文面向Java语言设计并实现了一个轻量化的跨语言FUSE框架JNI-FUSE. 相比现有跨语言FUSE框架JNR-FUSE,JNI-FUSE能够显著提升小文件和高并发场景下的吞吐量. 本文将JNI-FUSE集成到了分布式文件系统Alluxio,并在真实Kubernetes环境中测试了端到端性能及其对深度学习训练负载的影响. 实验结果显示本文提出的方法显著提升了文件读写吞吐量,降低了深度学习训练耗时. 本文设计实现的FUSE框架已开源,并被Alluxio官方社区接受集成并作为默认配置使用.

在未来的工作中,将进一步面向更多应用场景优化FUSE库的运行性能. 此外,还将继续深入研究其他编程语言的跨语言FUSE性能优化.

### 参考文献

[1] LI H, GHODOSI A, ZAHARIA M, et al. Tachyon: reliable, memory speed storage for cluster computing frameworks[C]//Proceedings of the 2014 ACM Symposium on Cloud Computing (SoCC). New York: ACM, 2014: 6.1-6.15.

[2] 孙勇. 林菲. 等. 面向云计算的键值型分布式存储系统研究[J]. 电子学报, 2013, 41(7): 1406-1411.
SUN Y, LIN F, WANG B J. Study on the key-value distributed storage system for cloud computing[J]. Acta Electronica Sinica, 2013, 41(7): 1406-1411. (in Chinese)

[3] GNU. POSIX (The Portable Operating System Interface) [EB/OL]. (2016-04-11)[2021-09-28]. https://www.gnu.org/software/libc/manual/html_node/POSIX.html.

[4] Linux Kernel Development Community. FUSE - the linux kernel documentation[EB/OL]. (2021-08-23)[2021-09-28]. https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[5] ALEXANDROV A D, IBEL M, SCHAUSER K E, et al. Extending the operating system at the user level: the UFO global file system[C]//Proceedings of the 1997 USENIX Annual Technical Conference (USENIX ATC). California: USENIX, 1997: 77-90.

[6] MAZIERES D. A toolkit for user-level file systems[C]//

Proceedings of the 2001 USENIX Annual Technical Conference (USENIX ATC). California: USENIX, 2001: 261-274.

[7] WEIL S A, BRANDT S A, MILLER E L, et al. Ceph: a scalable, high-performance distributed file system[C]//Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI). California: USENIX, 2006: 307-320.

[8] Red Hat, Inc. Gluster[EB/OL]. (2021-09-28)[2022-04-08]. https://www.gluster.org/.

[9] VANGOOR B K R, TARASOV V, ZADOK E. To FUSE or not to FUSE: performance of user-space file systems [C]//Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST). California: USENIX, 2017: 59-72.

[10] BIJLANI A, RAMACHANDRAN U. Extension framework for file systems in user space[C]//Proceedings of the 2019 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC). California: USENIX, 2019: 121-134.

[11] ISHIGURO S, MURAKAMI J, OYAMA Y, et al. Optimizing local file accesses for fuse-based distributed storage[C]//Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SC). New York: IEEE, 2012: 760-765.

[12] NARAYAN S, MEHTA R K, CHANDY J A. User space storage system stack modules with file level control[C]// Proceedings of the 12th Annual Linux Symposium in Ottawa. Ottawa: OLS, 2010: 189-196.

[13] TSELOVALNIKOV S. JNR-FUSE[EB/OL]. (2021-04-30)[2021-09-28]. https://github.com/SerCeMan/jnr-fuse.

[14] LEVART P. FUSE-J: java bindings for FUSE[EB/OL]. (2015-08-06) [2021-09-28]. https://sourceforge. net/projects/fuse-j/.

[15] libfuse: The reference implementation of the linux FUSE interface[EB/OL]. (2021-09-06) [2021-09-28]. https://github.com/libfuse/libfuse/.

[16] Oracle, Inc. Java native interface[EB/OL]. (2021-09-28) [2022-04-08]. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/.

[17] DOUBROVKINE D, BLÄSING M. Java native access (JNA)[EB/OL]. (2021-08-23)[2021-09-28]. https://github.com/java-native-access/jna.

[18] SIEGER N, NUTTER C O, JENVEY P, et al. The Java native runtime project[EB/OL]. (2021-03-09) [2021-09-28] https://github.com/jnr.

[19] WANG L, YE S, YANG B, et al. DIESEL: A dataset-based distributed storage and caching system for large-scale deep learning training[C]//Proceedings of the 49th International Conference on Parallel Processing (ICPP). New York: ACM, 2020: 20.1-20.11.

[20] Oracle, Inc. The invocation API of JNI[EB/OL]. (2021-09-26)[2021-09-28]. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/invocation.html.

[21] Oracle, Inc. JNI copy array[EB/OL]. (2021-09-06)[2021-09-28]. https://docs. oracle. com/javase/8/docs/technotes/guides/jni/spec/functions. html#Get_PrimitiveType_ArrayRegion_routines.

[22] Oracle, Inc. DirectByteBuffer of Java[EB/OL]. (2021-09-21) [2021-09-28]. https://docs. oracle. com/javase/8/docs/api/java/nio/ByteBuffer.html.

[23] Oracle, Inc. JNI critical copy array[EB/OL]. (2018-02-21) [2021-09-28]. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/functions. html#GetPrimitiveArrayCritical_ReleasePrimitiveArrayCritical.

[24] Kubernetes: Production-grade container orchestration[EB/OL]. (2020-09-06)[2021-09-28]. https://kubernetes.io/.

[25] Kubernetes container storage interface (CSI) documentation [EB/OL]. (2018-03-03) [2021-09-28]. https://kubernetes-csi.github.io/docs/introduction.html.

[26] Alluxio, Inc. FUSE-based POSIX API[EB/OL]. (2021-03-09)[2021-09-28]. https://docs.alluxio.io/os/user/stable/en/api/POSIX-API.html.

[27] NETZER Y, WANG T, COATES A, et al. Reading digits in natural images with unsupervised feature learning[EB/OL]. (2020-09-20)[2021-09-28]. http://ufldl.stanford.edu/housenumbers.

[28] COCO: Common objects in context[EB/OL]. (2020-09-26)[2021-09-28]. https://cocodataset.org/.

[29] PyTorch[EB/OL]. (2020-09-02) [2021-09-28]. https://pytorch.org/.

[30] PyTorch. ImageNet training in PyTorch[EB/OL]. (2020-09-02)[2021-09-28]. https://github.com/pytorch/examples/tree/master/imagenet.

[31] DENG J, DONG W, SOCHER R, et al. ImageNet: a large-scale hierarchical image database[C]//Proceedings of 2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. New York: IEEE, 2009: 248-255.

[32] SERGEEV A, BALSO M D. Horovod: fast and easy distributed deep learning in TensorFlow[J/OL]. (2018-02-01) [2021-09-28]. https://arxiv.org/abs/1802.05799.

[33] Tensorflow Horovod2 benchmark[EB/OL]. (2020-09-26) [2021-09-28]. https://github.com/horovod/horovod/blob/master/examples/tensorflow2/tensorflow2_synthetic_benchmark.py.
[34] KRIZHEVSKY A. The CIFAR-10 dataset[EB/OL]. (2021-09-28) [2021-09-28]. http://www.cs.toronto.edu/kriz/cifar.html.

## 作者简介

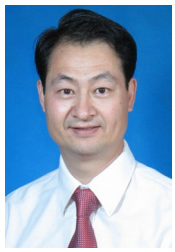
**顾 荣** 男,1988年出生,江苏泰州人. 博士,南京大学特聘研究员. 主要研究方向为大数据与云计算系统,分布式缓存与高效索引系统.
E-mail: gurong@nju.edu.cn

**罗义力** 男,1998年出生. 南京大学计算机科学与技术系硕士研究生. 主要研究方向为分布式存储系统.
E-mail: luoyl@smail.nju.edu.cn

**仇伶玮** 男,1994年出生. 南京大学计算机科学与技术系硕士研究生. 主要研究方向为深度学习系统.
E-mail: mp1933003@smail.nju.edu.cn

**王肇康(通讯作者)** 男,1990年出生,河南郑州人. 博士、讲师、硕士生导师. 主要研究方向为分布式图计算、分布式数据处理、云计算技术.
E-mail: wangzhaokang@nuaa.edu.cn

**戴海鹏** 男,1985年出生,湖南娄底人. 博士、副教授、博士生导师. 主要研究方向为高效数据索引、物联网.
E-mail: haipengdai@nju.edu.cn



**黄宜华** 男,1962年出生,江苏泰州人. 博士、教授、博士生导师. 主要研究方向为大数据系统、自动化机器学习、文本分析处理技术.
E-mail: yhuang@nju.edu.cn