用户态并行文件系统性能优化研究*

邓通亮[1,2,3]，陈宸[1,2,3]，殷树[1†]

(1 上海科技大学信息科学与技术学院，上海 201210； 2 中国科学院上海微系统与信息技术研究所，上海 200050；3 中国科学院大学，北京 100049)

(2020 年 3 月 23 日收稿；2020 年 5 月 11 日收修改稿)

邓通亮,陈宸,殷树. 用户态并行文件系统性能优化研究[J]. 中国科学院大学学报,doi:10.7523/j.ucas. 2020. 0027.

摘 要 用户态文件系统框架（FUSE）因其相对简单快捷的开发特性被广泛应用于新型文件系统的开发过程中。但是 FUSE 框架在处理应用的 I/O 请求时存在多次用户态和内核态切换、上下文切换以及额外的内存拷贝操作，造成一定的性能开销。为减小 FUSE 对 I/O 性能的影响，提出一种绕过 FUSE 的方案。该方案利用动态链接技术，通过将 I/O 请求的接收和转发功能实现在用户空间的方式消除 FUSE 框架的接入。该方案针对并行日志文件系统（PLFS）进行了应用实现和测试，结果表明，该方案对于大块读操作的性能提升最大可达 131%，对小块写操作的性能最大提升 5 倍左右。

关键词 文件系统；存储系统；FUSE；并行文件系统

中图分类号:TP302 **文献标志码**:A **doi:**10. 7523/j.ucas. 2020. 0027

On the efficiency of user-level parallel file systems

DENG Tongliang[1,2,3], CHEN Chen[1,2,3], YIN Shu[1]

(1 *School of Information Science & Technology, ShanghaiTech University, Shanghai* 201210, *China;* 2 *Shanghai Institute of Microsystem and Information Technology, Chinese Academy of Sciences, Shanghai* 200050, *China;* 3 *University of Chinese Academy of Sciences, Beijing* 100049, *China*)

Abstract A common and easier way to deploy parallel filesystem is using the user-space file system framework (e.g. FUSE). However, it serves as an I/O interposition layer which crosses user and kernel space that may introduce significant overhead due to the user-kernel mode switches, context switches and additional memory copies. This paper designs a method based on dynamic linking techniques to bypass FUSE, and implements the interposition layer in user space. This method is implemented and evaluated in

* 国家自然科学基金（61402158）资助

† 通信作者，E-mail：yinshu@shanghaitech.edu.cn

PLFS (Parallel Log-structured File System) which is a popular parallel file system for checkpointing. The experiments and evaluation show that it can improve 131% of the read performance when the transfer block size is large, and improve 5 times write performance at small transfer block size and guarantee no significant write performance downgrade in other scenarios.

Keywords file system; storage system; FUSE; parallel file system

在高性能计算系统中，并行文件系统的性能表现起着至关重要的作用[1-3]。目前存在不少的并行文件系统，例如 PVFS[4]、PLFS[5]和 Lustre[6]等。高性能计算系统由多个部分组成，应用的工作负荷往往复杂多变，系统中的薄弱环节易成为性能瓶颈，需要对其进行针对性的优化。例如，文献[7]对并行文件系统的小文件性能进行优化，文献[8]对文件系统中元数据的访问进行优化。

一般的，文件系统 (例如 ext4) 在操作系统内核中实现。但是，由于开发内核态文件系统难度大，不少开发者选择使用 FUSE[9] (Filesystem In Userspace，用户态文件系统框架) 作为文件系统和内核之间的中间层，在用户态开发文件系统[10-11]。通过 FUSE，使用者可以在不修改内核的情况下部署文件系统，开发者可以在不进入内核的情况下调试文件系统。常见的基于 FUSE 的用户态文件系统有 PLFS[5]、GlusterFS [12]和 Sshfs[13]等。

然而，在基于 FUSE 实现的用户态文件系统中，由于 FUSE 的介入，处理每一个 I/O 请求都会额外引入多次用户态和内核态之间的运行态切换 (下文简称：运行态切换)、上下文切换和内存拷贝，加上 FUSE 内部的等待队列和 I/O 请求重排，存在一定的性能损失。Bent 等[5]发现 FUSE 能够造成 20%左右的性能下降。Vangoor 等[14]研究了 ext4 上 FUSE 的性能特点，表明 FUSE 在不同的工作负荷下导致不同程度的性能下降，最大下降 83%。最近研究者发现 Intel 处理器中存在 Meltdown[15]等一系列漏洞，操作系统发布的补丁程序增强了用户态和内核态内存空间的隔离性，导致运行态切换的开销增加。

为优化用户态文件系统的性能，Ishiguro 等[16]提出一种通过 FUSE 的内核模块直接与内核文件系统交互的方法，优化用户态文件系统的性能，但该方法需要对内核进行修改，可移植性较差。Zhu 等[17]提出一种将 FUSE 移植到用户空间的方法，虽然消除了运行态切换和上下文切换，但是由于只有基于 libsysio[18] API 进行开发的应用程序才能使用该方法，存在一定的可扩展性问题。Rajgarhia 和 Gehani[19]通过 JNI (Java Native Interface) 实现了基于 JAVA 的 FUSE 接口，并且给出性能测试，但是缺乏深入的性能结果分析。文献[20]基于特定应用的 I/O 工作负荷特征，对高性能文件系统进行优化和深入的结果分析。

本文利用动态链接技术，在用户态并行文件系统中实现了一种绕过 FUSE 的方法，消除 FUSE 的性能影响，对比了用户态并行文件系统在基于 FUSE 和绕过 FUSE 后的性能特点。该方法可以运用于所有动态链接了标准 I/O 库的应用，不需要对应用进行任何的修改就可以提高 I/O 性能。实验结果表明，该方法可以显著提高 I/O 性能。其中，在传输块较大时，读性能最大提高 131%，写性能和有 FUSE 时性能相仿；传输块较小时，写性能最大提高 5 倍左右。

1 背景与问题分析

1.1 用户态文件系统框架

文件系统通常在内核中实现，应用的 I/O 请求通过系统调用的方式陷入内核态，利用其更高的执行权限完成 I/O。与此同时，存在一类拦截 I/O 请求

的中间层文件系统，它们向应用提供文件的抽象，并支持 POSIX 文件 I/O 语义，但是不实际存储文件，而是将文件的内容重新组织并存储到底层存储系统中，这样的中间层称为可堆叠文件系统[21]。FUSE 作为应用和用户态文件系统之间的桥梁拦截并转发 I/O 请求，因此可以使用 FUSE 开发此类可堆叠文件系统。

图 1[14] 展示了 FUSE 的基本架构与操作流程。FUSE 由两部分组成，内核模块和用户态守护进程，两者通过虚拟设备/dev/fuse 进行通信。在基于 FUSE 的文件系统上触发 I/O 请求时，内核虚拟文件系统 (Virtual Filesystem, VFS) 将该请求转发至 FUSE 内核模块，内核模块再将其重定向到对应的用户态守护进程，守护进程处理该请求，请求结果沿着原路返回。由此可见,处理一次 I/O 请求最少需要四次运行态切换，两次上下文切换和两次内存拷贝；如果守护进程涉及本地文件系统的读写，那么至少还需要两次运行态切换和一次内存拷贝。传统的内核文件系统处理一次 I/O 请求一般只需要两次运行态切换和一次内存拷贝，而不需要上下文切换。相比之下，FUSE 两次不可避免的上下文切换会带来较大的性能损失[22]。

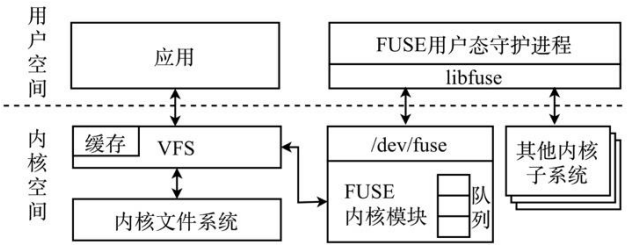

图 1 FUSE 的基本架构

Fig.1 FUSE high-level architecture

1.2 并行日志文件系统

为应对软硬件的错误，高性能计算系统会采用各种容错方案，例如检查点技术[23]。通过定期的将中间结果保存到存储系统，当系统发生错误并且被重置后，应用从可以从最近的检查点中恢复状态并且继续运行。

PLFS[5] (Parallel Log-structured File System, 并行日志文件系统) 通过在存储栈中引入一个中间层的方式对检查点的写性能进行优化。PLFS 重新组织存储模式，将 N-1 读写模式转成 N-N 读写模式，同时利用日志型文件系统利于写操作的特点[24]，极大的提高了检查点的写入性能。PLFS 专用于并行 I/O，并且进行了大量的优化，可以通过 FUSE 挂载到目录树，因此能够在大量的 I/O 工作负荷下对 FUSE 进行测试，充分反映本文所设计方法的优势与不足。图 2 给出了通过 FUSE 挂载 PLFS 后的 I/O 操作流程。

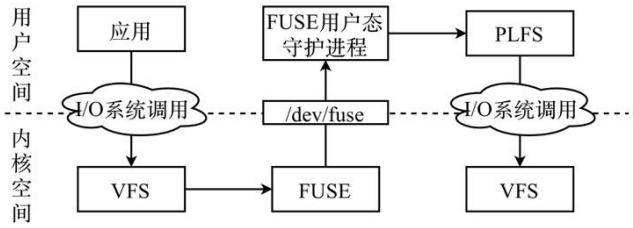

图 2 PLFS 通过 FUSE 的 I/O 操作流程

Fig.2 I/O routine of PLFS via FUSE

2 绕过 FUSE 的实现

PLFS 提供了一组 API，应用可以通过 API 直接使用 PLFS 的功能。绕过 FUSE 的一种方法是将应用源代码中所有 I/O 相关的函数调用替换成对 PLFS API 的调用，然后将源代码重新编译得到直接使用 PLFS 的可执行文件。该方法需要得到应用的源代码，并且对其有比较深入的理解，对于大多数应用来说局限性较大。

另一种方法是采用动态链接库覆盖用户态 I/O 函数的方式,将 I/O 操作重定向到 PLFS 内部。本文基于 PLFS API 设计并实现了一个动态链接库 libplfs，通过 libplfs 能够在不需要对应用进行修改的情况下，达到绕过 FUSE 的目的。与此同时，在 libplfs 中收集相关数据，分析 FUSE 性能表现背后的原因。

2.1 链接与加载

动态链接库能够在运行时被链接到进程的地址

空间，因此可以定义和 I/O 调用具有相同函数签名的函数，然后编译成动态链接库，利用该动态链接库对应用中的 I/O 调用进行高效地重定向，最终实现将 I/O 请求转发到 PLFS 中，把核心操作都放在用户态中完成。

LD_PRELOAD[25]环境变量用于指定最先加载的动态链接库，让自定义的动态链接库拥有甚至比标准库高的最高优先级，达到优先加载和链接自定义函数库的目的。

本文的方法是将 PLFS API 封装到 libplfs 动态链接库中，利用 LD_PRELOAD 实现优先加载和链接 libplfs 中的 I/O 库函数。

2.2 文件映射表与文件状态管理

由于 POSIX API 与 PLFS API 之间存在差异，上层应用通过 POSIX 文件描述符进行 I/O 操作，而在 PLFS 上进行 I/O 操作所使用的文件描述符具有其自身特点，因此需要在 libplfs 中实现从 POSIX 文件描述符到 PLFS 文件描述符的转换，并且对文件的状态进行管理。

本文通过两个内存中的全局哈希表 (时间复杂度 O(1)) 实现文件映射关系的管理，分别为 fd_table 和 path_table。fd_table 中的 key 为上层应用所使用的 POSIX 文件描述符 fd，value 为 plfs_file 结构体指针 (见图 3)。fd_table 的作用是向应用提供 POSIX 文件描述符，保证可以在不对应用进行修改的情况下绕过 FUSE 并且使用 PLFS。path_table 哈希表用于对同一文件打开多次的情况，结合引用计数，实现将多个 POSIX 文件描述符 fd 映射到同一个 plfs_file 对象。fd_table 中的键值对在 libplfs 中的 open 函数内完成插入 (伪代码 1 中第 10 和 16 行)，在 close 函数中进行删除，在 libplfs 中重载的其他 I/O 函数内，通过查询 fd_table 哈希表实现从 POSIX 文件描述符到 PLFS 文件描述符的转换。path_table 仅在 libplfs 中的 open 函数 (伪代码 1 中第 8、9、10 和 15 行) 和 close 函数内使用。

Pseudocode 1　open() function in libplfs

```
1  int open(const char* path, int flags, …) {
2    int (open_*)(const char*, int, …);
3    open_ = (int (*) (const char*, int, ...)) dlsym
     (RTLD_NEXT, "open");
4    if (!is_plfs_path(path)) //打开常规文件
5      return open_(path, flags);
6    //打开 PLFS 中的文件
7    int fd = tmpfile(); //在 tmpfs 中创建临时文
     件，得到上层应用所使用的文件描述符 fd
8    if (path_table.count(path) != 0) { //该路径所
     表示的文件已经被打开
9      path_table[path]->ref_num++;
10     fd_table[fd] = path_table[path];
11   } else { //第一次打开该文件
12     Plfs_file* pf = malloc(sizeof(Plfs_file));
13     pf->plfs_fd = plfs_open(path, flags);
14     … //初始化 pf 中的其他成员变量
15     path_table[path] = pf;
16     fd_table[fd] = pf; } //建立映射关系
17   return fd; } //向上层应用返回 fd
```

以 libplfs 中重载的 open 函数为例 (伪代码 1)，在打开某一文件时，通过路径参数判断该文件是否位于 PLFS 文件系统中，如果是，则在 tmpfs 上创建一个 POSIX API 所使用的的文件描述符 fd，然后打开该 PLFS 文件，利用 fd_table 和 path_table 记录 fd 与 PLFS 文件描述符之间的映射关系，为接下来应用通过 POSIX 文件描述符 fd 触发 PLFS 上的 I/O 操作提供翻译功能。伪代码 1 中涉及的 Plfs_file 结构体中具体成员信息见图 3。

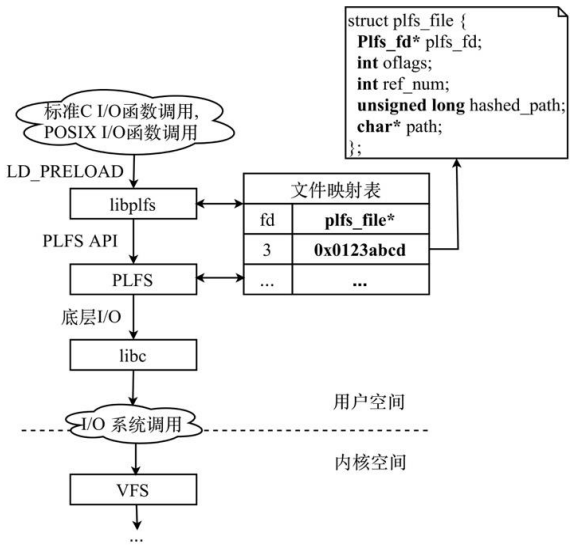

图 3 libplfs 的处理流程

Fig.3 Process Flow of libplfs

2.3 libplfs 的实现

libplfs 的主要功能是在用户空间维护一个文件状态映射表，将对 C 标准库和 POSIX I/O 函数的调用转换成对 PLFS API 的调用。libplfs 的具体函数调用流程进行见图 3。

本文通过 open(), read(), write()和 close()这四个具有代表性的 I/O 函数来对 libplfs 的实现进行说明。在打开一个文件时，libplfs 在本地文件系统上创建一个临时的虚拟文件，为应用进程分配一个文件描述符 fd。接着 libplfs 调用 plfs_open()函数创建一个 PLFS API 使用的文件对象 plfs_file，并且建立虚拟文件描述符 fd 与 plfs_file 的映射关系。当应用通过文件描述符 fd 读写文件时，libplfs 利用 plfs_file 文件对象直接调用 PLFS API，同时将应用传递过来的内存地址、请求大小和文件偏移量传递给 PLFS API；在 PLFS 上完成 I/O 请求后对虚拟文件描述符 fd 的当前偏移量进行对应的调整。在关闭文件时，释放包括映射表中对应项在内的所有内存资源。对于并发的 I/O 请求，例如若干个线程同时打开同一个文件，通过引用计数的方式记录打开的次数，避免创建冗余的对象，提高内存利用率。

在进行 I/O 操作之前，libplfs 对接收的参数进行判断，区分该操作是一般的 I/O 操作还是在 PLFS 上相关的 I/O 操作。如果是对 PLFS 的操作，那么 libplfs 调用对应 PLFS API 处理该请求，否则通过调用 dlsym()函数得到 libc 中对应 I/O 函数的指针并调用它，如伪代码 1 中的第 3 和第 5 行。

libplfs 重载了 69 个几乎所有 libc 中的 I/O 库函数。用户可以简单的通过设置 LD_PRELOAD 环境变量，将目标程序中的 I/O 操作重定向到 libplfs 动态链接库中，进而直接调用 PLFS API，达到绕过 FUSE 的目的。该方法对所有在运行时动态链接了 libc 库中 I/O 相关函数的可执行文件都适用。

3 实验与分析

3.1 实验平台

测试集群由 5 个节点组成，其中 1 个作为主节点，4 个以 HDD (Hard Disk Drive, 机械硬盘) 为介质的存储节点。节点配置如表 1 所示。

表 1 集群节点配置

Table 1 Cluster node specification

项目	参数	
节点	主节点(x1)	存储节点(x4)
CPU	Intel(R) Xeon(R) CPU E5-2603 v4 @ 1.70GHz	
内存	16GB DDR4 2400MHz	
存储	2 * 1TB HDD	4 * 1TB HDD
OS	Linux Mint, 4.8.0-53-generic 内核	

PLFS 作为一个可堆叠文件系统，实际不存储数据，而是将数据重新组织到底层文件系统中，向应用提供一个虚拟文件的接口。CephFS[26]是常见的高性能的文件系统，本文使用它作为 PLFS 的后端存储。4 个存储节点组成一个 Ceph 对象存储池，其中 1 个节点还充当元数据服务器。主节点通过 Ceph 的内核模块将 CephFS 挂载到主节点的目录树中，将该挂载点目录设置成 PLFS 的后端存储。

3.2 实验设计

测试 libplfs 时，在 LD_PRELOAD 的作用下，

所有 I/O 函数的调用被重定向到 libplfs 中，该测试探究 libplfs 下 PLFS 的性能，记作 no-FUSE。在有 FUSE 并且默认设置的情况下进行测试，作为控制组 FUSE-with-cache。在有 FUSE 时，不使用内核页缓存的方式下进行测试，探究 FUSE 内核模块中页缓存对性能的影响，记作 FUSE-direct-IO。后端 CephFS 文件系统通过 Ceph 的内核模块挂载，并且启用缓存功能，采用默认配置。

本文通过文件大小、传输块大小和并发度 3 个参数的组合配置进行测试。每组实验重复 10 次，实验参数配置见表 2。

表 2　实验参数设置

Table 2　Experiment's parameter setup

项目	参数
I/O 方法	MPI I/O 读(read)，MPI I/O 写(write)
并发线程数	1, 4
文件大小	128MB, 512MB, 1GB, 2GB
传输块大小	1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 512KB, 1MB, 2MB

本文采用 LANL 开发的 fs_test[27]作为基准测试工具。为了便于叙述，本文对测试点采用形如“4-512m-16k-read”的命名方式，表示 4 个线程并发读同一个大小为 512 MB 的文件，传输块大小为 16 KB。在读测试之前，我们先完成对应的写测试以此来达到系统预热的效果，目的是更好的利用后端存储的缓存，避免后端存储成为性能瓶颈，从而掩盖 FUSE 的性能影响，造成实验数据偏差。每个测试点完成后，清空缓存中的数据。

得到测试结果之后，我们对一些测试点使用系统剖析工具进行深入分析，探究 FUSE 如何影响 I/O 性能，以及 libplfs 的优势。

3.3　实验结果

本文选择具有代表性的结果来进行说明，类别为小文件和大文件、串行和并发、读和写、小传输块大小(1 KB, 4 KB, 16 KB)和大传输块大小(64 KB, 256 KB, 1 MB)。

传输块大小与缓存利用率直接相关，决定了在给定文件大小条件下系统调用和内核函数调用的数量。选择传输块大小作为 x 轴。每一个实验运行 10 次，结果去掉最大最小值后取平均。

图 4 展示读测试的结果。从结果中可以得出，在所有传输块大小下，no-FUSE 总是比 FUSE-direct-IO 的带宽要高。当传输块大小大于 32 KB 时，no-FUSE 的带宽最高，最大提升 131%。当传输块大小小于 32 KB 时，FUSE-with-cache 带宽最高。FUSE-with-cache 的带宽总是比 FUSE-direct-IO 高。变化趋势方面，FUSE-direct-IO 与 no-FUSE 的变化趋势类似，而 FUSE-with-cache 的变化趋势比较平稳。单线程与多线程下读性能相仿。

图 5 展示写测试的结果。从结果中可以得出，在所有传输块大小下，FUSE-direct-IO 的带宽总是比 FUSE-with-cache 高。当传输块大小小于 64 KB 时，no-FUSE 带宽最高，最大提升 5.7 倍。当传输块大小大于 64 KB 时，no-FUSE 被 FUSE-direct-IO 超过，两者性能差距大部分位于 1%至 8%之间，但两者都比 FUSE-with-cache 高。图 6 给出了写测试中 no-FUSE 对比 FUSE-with-cache 所带来的的性能提升结果。结果表明，当传输块大小为 1 KB 时，四线程写测试的性能提升更大，当传输块大小大于 1 KB 时，单线程写测试提升更加明显。换句话说，在并发小请求的写工作负荷下，绕过 FUSE 带来的性能提升更加明显。

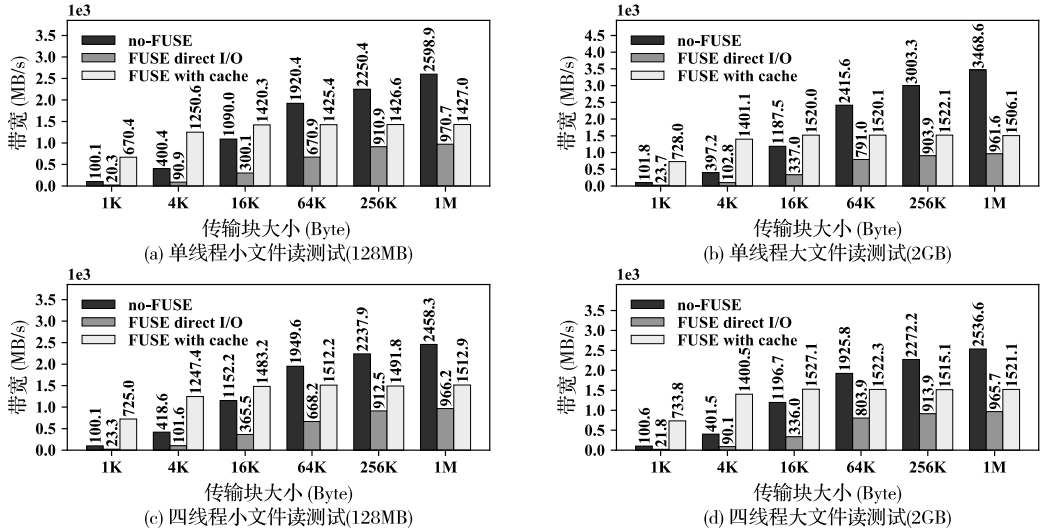

图 4 读性能比较

Fig.4 Read performance comparison

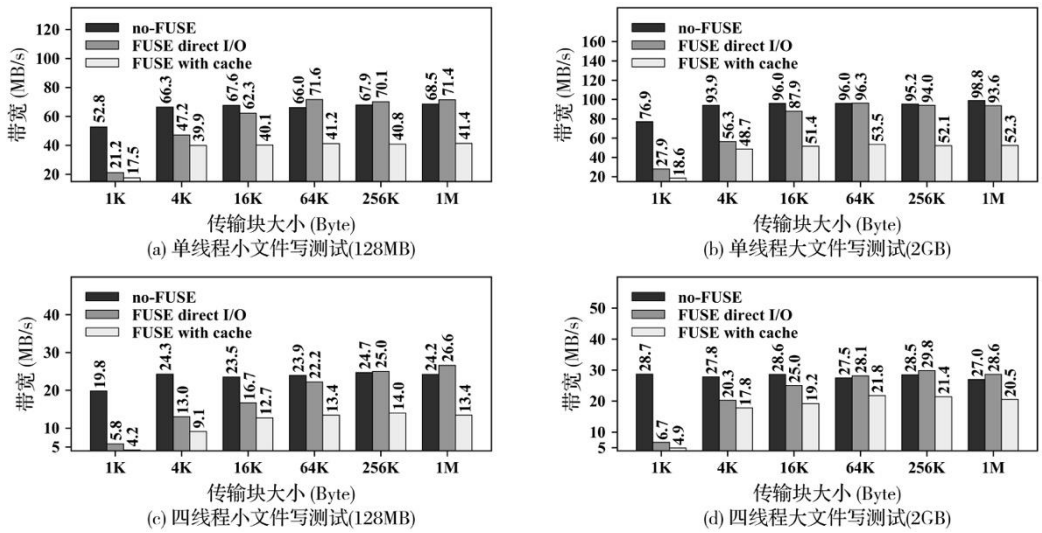

图 5 写性能比较

Fig.5 Write performance comparison

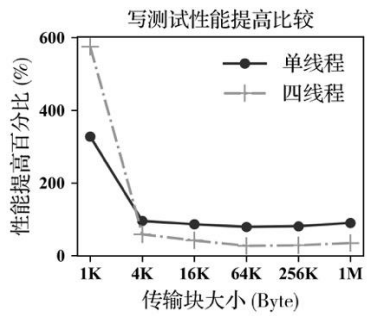

图 6 写性能提高比较

Fig.6 Write performance improvement comparison

3.4 结果分析

FUSE 的内核模块可以利用内核页缓存，进行预取和写回操作，进而影响性能。通过 libplfs 绕过 FUSE 之后，运行态切换、系统调用和内存拷贝的减少有利于提升性能。

本节借助 Linux 系统提供的系统调优和调试工具对得到的结果进行分析和研究，所涉及的工具包括 perf 和 ftrace。我们利用这些工具统计分析 I/O 过程中系统调用操作的数量、内核函数调用的情况以

及页缓存的使用率。

3.4.1 读操作与页缓存利用率

FUSE 的主要特点之一是可以利用内核页缓存。Linux 内核将 FUSE 内核模块和底层文件系统当作两个不同的文件系统，它们在内核有独立的缓存空间，因此 FUSE 存在双重缓存的问题。换句话说，对一个文件进行读操作将导致同一份内容被缓存两次。在 CephFS 缓存被清空后运行“4-2g-1k-read”测试，页缓存使用情况见表 3。

表 3 清空缓存后测试缓存使用情况

Table 3 Cache information when cache cleaned

	FUSE-with-cache	FUSE-direct-IO	no-FUSE
页缓存命中率	65.49%	95.21%	72.56%
缓存空间占用	4 096 MB	2 048 MB	2 048 MB

表 3 结果显示，当后端存储缓存被清空时，双重缓存使 FUSE-with-cache 的缓存命中率最低；当后端缓存被预先填充时，FUSE-with-cache 的缓存命中率可以提高到 80%左右。

在读测试之前，先进行写测试然后卸载 PLFS，保证读测试时 CephFS 的缓存处于有效状态。重新运行“4-128m-1k-read”和“4-128m-1m-read”测试项，统计 VFS、FUSE 和 CephFS 上内核函数的调用情况，结果见表 4。

表 4 内核函数调用数量

Table 4 Kernel function call count

	FUSE-with-cache	FUSE-direct-IO	no-FUSE
测试名称	4-128m-1k-read		
vfs_read	533 191	1 579 679	531 067
fuse_*read_iter	524 297	524 297	0
ceph_read_iter	1 506	524 297	524 297
测试名称	4-128m-1m-read		
vfs_read	11 919	19 052	11 067
fuse_*read_iter	512	512	0
ceph_read_iter	2,717	6,492	3,049

当传输块大小为 1KB 时，FUSE-with-cache 在缓存的作用下，使 Ceph 上的读操作只被触发了 1 506 次；FUSE-direct-IO 和 no-FUSE 的情况下，Ceph 上的读操作是其 350 倍之高。当传输块大小为 1 MB 时，它们之间的差距缩小了很多，只有 2.4 和 1.1 倍。结果表明，当传输块大小比较小时，在内核页缓存的作用下，FUSE-with-cache 的性能最高。

3.4.2 系统调用(运行态切换)

I/O 通过系统调用完成，系统调用数量与运行态切换数量正相关。当传输块大小很小时，一个 I/O 请求的实际数据传输时间很短，导致运行态切换开销占比很大。除此之外，系统调用数量与用户空间和内核空间之间的内存拷贝次数存在密切联系。对“4-128m-1k-write”和“4-128m-1m-read”两项测试中触发的系统调用次数进行统计，得到表 5。

表 5 系统调用情况

Table 5 System call count

	FUSE-direct-IO	no-FUSE
测试名称	4-128m-1k-write	
系统调用数量	7 037 259	1 611 635
带宽	5.38 MB/s	18.92 MB/s
测试名称	4-128m-1m-read	
系统调用数量	91 872	34 267
带宽	1 006 MB/s	2 327 MB/s

表 5 展示了测试的带宽和系统调用数量。在传输块大小为 1KB 的写测试中，FUSE-direct-IO 的系统调用数量是 no-FUSE 的 4.4 倍，然而每一个 I/O 请求的数据传输时间很短，大量系统调用使运行态切换的总开销占比很高，导致 no-FUSE 的写性能是 FUSE-direct-IO 的 3.5 倍。对于传输块大小为 1 MB 的读测试，no-FUSE 的系统调用数量为 34 267 远小

于 FUSE-direct-IO 的 91 872,导致 no-FUSE 的读性能提升了 131%。

4 结束语

本文研究了 FUSE 在并行文件系统中的性能问题，基于动态链接的机制，实现了一种绕过 FUSE 的 libplfs 动态链接库，并且在 PLFS 文件系统进行实验和验证。实验结果表明，通过 libplfs 绕过 FUSE 后，可以在保证写性能的前提下，当传输块大小较大时，读性能提高最大可达 131%；当传输块大小较小时，写性能提高最大 5 倍左右。对结果进行研究分析，发现内核页缓存和大量的系统调用是 FUSE 影响性能的重要因素。基于 FUSE 的文件系统在传输块大小较小时，双重缓存给文件系统的读性能带来了一定的性能提升，但同时大量的系统调用使其写性能降低 2.5 倍以上。对于并行文件系统，并且存在大量大块 I/O，利用 libplfs 的方法可以带显著的性能提升。我们后续将对基于 FUSE 的不同文件系统进行研究，开展关于 FUSE 系统性能优化的通用性方面的探索，扩展现有的优化方案。

参考文献

[1] Bland B. Titan-early experience with the titan system at oak ridge national laboratory[C]//2012 SC Companion: High Performance Computing, Networking Storage and Analysis. Washington, DC, USA: IEEE Press, 2012: 2189-2211.

[2] Livermore's HPC Innovation Center. Catalyst [EB/OL]. (2017-10-08)[2018-01-01]. http://computation.llnl.gov/computers/catalyst.

[3] Oak Ridge National Laboratory. Summit[EB/OL]. (2018-07-30)[2019-08-07]. https://www.olcf.ornl.gov/summit/.

[4] Ross R B, Thakur R. PVFS: A parallel file system for Linux clusters[C]//Proceedings of the 4th annual Linux showcase and conference. CA, USA: USENIX Association, 2000: 391-430.

[5] Bent J, Gibson G, Grider G, et al. Plfs: A checkpoint filesystem for parallel applications[C]//Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. New York, USA: IEEE Press, 2009: 1-12.

[6] Braam P. The Lustre Storage Architecture[EB/OL]. (2019-03-05)[2019-10-15]. http://lustre.org/.

[7] 周恩强, 董永, 张伟,等. 对象存储并行文件系统小文件性能优化研究[J]. 计算机工程与科学, 2013, 35(12): 8-13.

[8] 刘恋, 郑彪, 龚奕利. 分布式文件系统中元数据操作的优化[J]. 计算机应用, 2012, 32(12): 3217-3273.

[9] Szeredi M. Fuse: File system in user space [EB/OL]. (2017-09-27)[2017-10-01]. http://fuse.sourceforge.net/.

[10] Rath N. List of fuse based file systems[EB/OL]. (2019-12-22)[2020-04-15]. https://github.com/libfuse/libfuse/wiki/Filesystems.

[11] Tarasov V, Gupta A, Sourav K, et al. Terra incognita: On the practicality of user-space file systems[C]//7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15). CA, USA: USENIX Association, 2015:15-15.

[12] Red Hat, Inc.. GlusterFS[EB/OL]. (2011-10-07)[2020-05-08]. https://www.gluster.org/.

[13] Hoskins M E. Sshfs: super easy file access over ssh[EB/OL]. (2006-04-28)[2020-05-08]. https://www.linuxjournal.com/article/8904.

[14] 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, Santa Clara, CA, USA: USENIX Association,

2017:59-72.

[15] Lipp M, Schwarz M, Gruss D, et al. Meltdown: Reading kernel memory from user space[C]//27th USENIX Security Symposium. Baltimore, MD, USA: USENIX Association, 2018: 973-990.

[16] Ishiguro S, Murakami J, Oyama Y, et al. Optimizing local file accesses for fuse-based distributed storage[C]//2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Washington, DC, USA: IEEE Press, 2012: 760-765.

[17] Zhu Y, Wang T, Mohror K, et al. Direct-FUSE: Removing the Middleman for High-Performance FUSE File System Support[C]//Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers. New York, USA: ACM Press, 2018:1-8.

[18] Ward L. The SYSIO library[EB/OL]. (2003-02-20)[2019-12-17]. https://sourceforge.net/projects/libsysio/.

[19] Rajgarhia A, Gehani A. Performance and extension of user space file systems[C]//Proceedings of the 2010 ACM Symposium on Applied Computing. New York, USA: ACM Press, 2010: 206-213.

[20] 王卫锋, 杨林. 基于 Hadoop 的邮政寄递大数据分析系统设计与实现[J]. 中国科学院大学学报, 2017, 34(3): 395-400.

[21] Zadok E, Nieh J. FiST: A Language for Stackable File Systems[C]//Proceedings of the Annual Conference on USENIX Annual Technical Conference. CA, USA: USENIX Association, 2000:5-21.

[22] Li C, Ding C, Shen K. Quantifying the cost of context switch[C]//Proceedings of the 2007 workshop on Experimental computer science. New York, USA: ACM Press, 2007: 2-5.

[23] Egwutuoha I P, Levy D, Selic B, et al. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems[J]. The Journal of Supercomputing, 2013, 65(3): 1302-1326.

[24] Rosenblum M, Ousterhout J K. The design and implementation of a log-structured file system[J]. ACM Transactions on Computer Systems (TOCS), 1992, 10(1): 26-52.

[25] Linux Programmer's Manual. ld.so(8)[EB/OL]. (2020-04-30)[2020-05-08]. http://man7.org/linux/man-pages/man8/ld.so.8.html.

[26] 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. CA, USA: USENIX Association, 2006: 307-320.

[27] Los Alamos National Laboratory. LANL fs test [EB/OL]. (2017-05-20)[2017-09-05]. https://github.com/fstest/fs_test.