哈爾濱工業大學(深圳)

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

eFUSE 设计开发文档

基于 eBPF 加速的高性能用户态文件系统

队伍名称 FastPoke

所属赛题 proj289

项目成员 许辰涛、冯可逸、赵胜杰

院内导师 夏文、李诗逸

项目导师 郑昱笙

所属高校 哈尔滨工业大学（深圳）

2025 年 6 月

摘　　要

这是一段摘要

目　　录

1 概述

1.1 背景及意义

FUSE（Filesystem in Userspace）是一种允许在用户态构建文件系统的 linux 机制，使开发者能够在不必修改内核源码的条件下，便捷且灵活地开发自定义文件系统，极大地降低了开发门槛，简化了开发流程，提高了内核安全性。然而，FUSE 的性能瓶颈一直备受诟病，尤其在高频繁元数据操作、大量小文件读写等场景下，内核态与用户态频繁切换成为主要性能瓶颈，限制了其在特定的高性能场景下的适用性。

在 FUSE 内部的实现中，来自 VFS (虚拟文件系统) 层的所有请求都被放入共享的待处理队列 (pending queue) 中，并由 FUSE 守护进程逐个提取。这种调度方式在某些高并发的场景下会导致严重的锁争用。在多核环境下，无法充分发挥多核处理器的并行优势，使得系统在面对大规模的 I/O 任务时吞吐率首先，处理时延较高，无法充分利用带宽的潜力。

eBPF（extended Berkeley Packet Filter）是 Linux 的一项强大特性，允许开发者在不修改内核源码的情况下向内核注入用户定义逻辑，已广泛应用于网络、安全、追踪等领域，eBPF 为解决和优化上述 FUSE 的性能问题提供了新的可能和方向。近年来，已有多项研究探索将 eBPF 引入文件系统以提升其性能，例如 ExtFuse、Fuse-BPF、XRP 等。我们期望通过本项目，进一步探索基于 eBPF 的 FUSE 加速路径，实现低延迟、高吞吐、具有良好扩展性的用户态文件系统。

1.2 目标

eFUSE 是一个尝试将 eBPF 深度集成到 FUSE 文件系统中的创新项目，旨在重构 FUSE 的传统执行路径和请求调度方式，以提高用户态文件系统的运行效率，同时保留 FUSE 的灵活性和安全性的优势。借助 eBPF 这一灵活的特性，对特定的文件系统进行性能优化，实现以下三大目标：

- **减少内核态与用户态之间的频繁切换**：在内核中直接处理部分 FUSE 请求（如 LOOKUP、READ 等），避免传统 FUSE 工作流程中频繁的内核/用户态切换，提高请求处理效率。
- **设计高效的 I/O 和元数据缓存机制**：利用 eBPF 的 map 数据结构实现元数据和读写数据的缓存机制，降低磁盘的访问频率。

- **实现跨核高并发优化与负载均衡机制**：针对 FUSE 共享请求队列带来的并发限制，设计更为合理、更适合多核的请求调度方式，并结合 eBPF 进行负载监控，避免锁的集中争用。

1.3 行动项

为实现上述目标，进一步将本项目分为五大技术目标模块：

表 1-1 目标技术模块

实现内容	说明
目标 1：FUSE 内核模块扩展	1. 支持新的 eBPF 程序类型。 2. 扩展 FUSE 挂载点支持。 3. 设计并注册文件系统相关 helper 函数。
目标 2：FUSE 元数据请求优化	1. 优化 inode、目录、权限、路径等相关操作。 2. 使用 eBPF map 实现元数据缓存。 3. 实现内核态与用户态高效协调访问。 4. 内核/用户态切换次数显著下降。
目标 3：FUSE I/O 请求的特殊优化	1. 支持直通路径：eBPF 直接读取文件内容。 2. 支持缓存路径：将内容存入 eBPF map 缓存。 3. 设计请求调度策略实现直通与缓存路径选择 4. 读写性能提升 1.5~3 倍。
目标 4：基于内核修改的多核优化	1. 为每个核心构建独立 ringbuf 管道代替请求队列。 2. 实现可扩展的核间通信机制。 3. 实现多核 CPU 环境的适配。
目标 5：负载监控与请求均衡	1. 利用 eBPF 动态分析请求负载。 2. 根据 ringbuf 状态进行调度策略调整。 3. 针对不同的负载情况实现合理的请求分配。

我们将上述目标拆分为以下若干行动项：

- 行动项 1：进行背景知识调研，了解 FUSE 的核心性能瓶颈。
- 行动项 2：搭建开发环境。
- 行动项 3：FUSE 内核驱动扩展、加载 eBPF 程序、设置挂载点入口。

- 行动项 4：实现并注册内核 eBPF helper 辅助函数。
- 行动项 5：实现 FUSE 元数据请求绕过路径和回退机制。
- 行动项 6：在用户态和内核中协调访问。
- 行动项 7：实现 FUSE I/O 请求 map 缓存绕过路径。
- 行动项 8：实现 FUSE I/O 请求直通绕过路径。
- 行动项 9：实现 FUSE I/O 请求中的自适应调度算法。
- 行动项 10：FUSE 请求绕过机制的安全性评估和处理。
- 行动项 11：为 FUSE 内核设计更为合理的请求队列数据结构。
- 行动项 12：通过 eBPF 实现对请求队列的负载监控和请求均衡。
- 行动项 13：模拟常见的负载场景并进行性能评估。

1.4 完成情况

在初赛阶段，针对上述行动项的完成情况如下：

- 行动项 1 （完成）：讨论并选定可行的 FUSE 优化方向。
- 行动项 2 （完成）：在虚拟机中搭建测试环境，基于 linux 6.5 开发。
- 行动项 3（完成）：使指定文件系统在挂载时自动加载 eBPF 程序，完成 eBPF 程序在送往用户态文件系统时的自动触发。
- 行动项 4（完成）：在内核中设计并注册合适的 eBPF helper 函数，便于后续开发，同时须确保 eBPF 程序安全性。
- 行动项 5 （完成）：实现 LOOUP、GETATTR 等元数据请求的绕过机制，大幅降低文件系统在运行时的内核态/用户态切换次数。
- 行动项 6（完成）：对指定的用户态文件系统做一定的修改，使其与 eBPF 程序协调配合，管理 eBPF map 中的数据内容。
- 行动项 7 （完成）：实现以 READ、WRITE 为主的文件 I/O 请求的 eBPF map 缓存机制，加快请求的处理速度。
- 行动项 8 （完成）：实现以 READ、WRITE 为主的文件 I/O 请求的 eBPF 直通路径，作为对缓存机制的补充。
- 行动项 9（完成）：设计并实现自适应路径选择算法，使系统在不同的负载情况下预测并选择较优的路径，读写性能提升 1.5~3 倍。
- 行动项 10（完成）：对完成的请求绕过机制进行安全性检查，防止文件读取越界等情况发生，进行处理和优化。

- 行动项11（完成）:在多核环境下为每个核心分配环形管道，代替原先的请求队列。
- 行动项 12 （进行中）
- 行动项 13 （基本完成）:设计模拟常见的负载场景测试。

1.5 开发历程

1.6 团队分工

2 现有研究调研

2.1 FUSE

2.2 eBPF

3 整体架构设计

3.1 设计理念

eFuse 项目旨在将 eBPF 深度集成到 FUSE 文件系统中，重构 FUSE 的传统执行路径和请求调度方式，以提高用户态文件系统的运行效率，同时保留 FUSE 的灵活性和安全性的优势。借助 eBPF 这一灵活的特性，对特定的文件系统进行性能优化。

针对传统 FUSE 的性能瓶颈，在尽可能不改变 FUSE 架构和接口标准的前提下，eFuse 尝试实现对用户态文件系统运行效率提升的同时，在各类实际负载情况下，都能保持优秀且稳定的运行性能。

一方面，对于传统 FUSE 文件系统处理流程中频繁的内核态/用户态切换，eFuse 尝试通过 eBPF 技术，在内核中快速处理部分 FUSE 请求以减少切换次数，提升请求处理的效率。另一方面，对于传统 FUSE 文件系统在高并发场景下的请求调度问题，eFuse 尝试在内核驱动中设计更为合理的请求队列结构，避免锁的集中争用，同时结合 eBPF 进行负载监控，避免请求处理的瓶颈。

eFuse 的核心设计理念为：

- **性能有限**：在不改变 FUSE 架构和接口标准的前提下，尽可能提升用户态文件系统的运行效率。
- **兼容性**：完全保持与现有 FUSE 文件系统的兼容性，确保用户态文件系统的接口和行为不变。
- **安全性**：在实现绕过机制的同时，确保系统的安全性和稳定性，避免文件读取越界等问题，确保在性能优化的同时不引入安全隐患。
- **可扩展性**：提供 eBPF 的可编程能力，设计灵活的 eBPF 程序和 map 结构，便于针对实际负载场景定义和加载专用逻辑。
- **高并发支持**：设计合理的请求调度和负载均衡机制，充分利用多核处理器的并行能力，提升系统在高并发场景下的吞吐率和响应速度。
- **易用性和可维护性**：提供简单易用的接口和配置方式，便于用户快速上手和使用 eFuse，同时易于后续维护和迭代。

3.2 设计架构介绍

4 模块设计和实现

4.1 FUSE 内核模块扩展

为实现“4.2 FUSE 元数据请求优化”和“4.3 FUSE I/O 请求优化”的优化功能，需要首先对内核中的 FUSE 驱动模块做一些必要的扩展和修改，以支持后续相关的优化逻辑，实现特定的功能并为用户态和 eBPF 程序提供接口。

具体修改内容包括，添加自定义 eBPF 程序类型、自动加载 eBPF 程序、设置 eBPF 挂载点、实现相关 eBPF helper 函数等。这些内核层面的改动旨在为用户态 FUSE 文件系统和内核态的 eBPF 程序提供接口支持和运行环境，确保后续的绕过机制能够正确工作。

4.1.1 eBPF 程序加载

该部分实现了在用户态 FUSE 文件系统挂载并初始化时，自动加载并注册提前编译好的 eBPF 程序二进制流。需要修改结构体 fuse_init_out，使用其中的一项未用字段来存放 eBPF 程序文件的文件描述符。

```
struct fuse_init_out {
    uint32_t    major;
    uint32_t    minor;
    uint32_t    max_readahead;
    uint32_t    flags;
    uint16_t    max_background;
    uint16_t    congestion_threshold;
    uint32_t    max_write;
    uint32_t    time_gran;
    uint16_t    max_pages;
    uint16_t    map_alignment;
    uint32_t    flags2;
    uint32_t    max_stack_depth;
    uint32_t    efuse_prog_fd;  //添加字段
    uint32_t    unused[5];
};
```

在 process_init_reply 中，通过传入的 fuse_init_out，获取 eBPF 程序文件 fd 并加载相应的 efuse eBPF 程序。

```
static void process_init_reply(struct fuse_mount *fm, struct
fuse_args *args,
                   int error)
{
    struct fuse_conn *fc = fm->fc;
    struct fuse_init_args *ia = container_of(args, typeof(*ia),
args);
```

```
    struct fuse_init_out *arg = &ia->out;
    bool ok = true;

    if (error || arg->major != FUSE_KERNEL_VERSION)
        ok = false;
    else {
        unsigned long ra_pages;

        process_init_limits(fc, arg);

        if (arg->minor >= 6) {
            ……
            if (flags & FUSE_FS_EFUSE) {
                // 加载 eBPF 程序
                efuse_load_prog(fc, arg->efuse_prog_fd);
            }
        }
        ……
    }
}
```

```
int efuse_load_prog(struct fuse_conn *fc, int fd)
{
    struct bpf_prog *prog = NULL;
    struct bpf_prog *old_prog;
    struct efuse_data *data;

    BUG_ON(fc->fc_priv);

    data = kmalloc(sizeof(*data), GFP_KERNEL);
    if (!data)
        return -ENOMEM;

    prog = bpf_prog_get(fd);
    if (IS_ERR(prog)) {
        kfree(data);
        return -1;
    }

    old_prog = xchg(&data->prog, prog);
    if (old_prog)
        bpf_prog_put(old_prog);

    fc->fc_priv = (void *)data;
    return 0;
}
```

4.1.2 自定义 eBPF 程序类型

该部分注册并添加了新的自定义 eBPF 程序类型 BPF_PROG_TYPE_EFUSE，在后续“4.2 FUSE 元数据请求优化”和“4.3 FUSE I/O 请求优化”中，将为绝大部分常见的、可优化性强的 FUSE 请求设定专门的 eBPF 程序，以进行用户态绕过操作或 eBPF map 维护操作，而他们的 eBPF 程序类型皆为自定义的 BPF_PROG_TYPE_EFUSE 类型。这么做有如下好处：

1. 可以为自定义的 eBPF 程序类型限制使用用途，增强安全性，起到隔离作用。一方面，在后续的 eBPF 程序设计中，为了实现目标功能，可能需要略微放宽验证器限制，单独为一个类型放宽限制可以提高系统整体的安全性，也更便于维护。另一方面，可以单独设置该类型只允许在特定的 FUSE 请求路径上加载，能够有效防止 eBPF 程序误用。
2. 可以灵活设计相关接口，注册专门的 eBPF helper 函数支持，这些 helper 函数只能在指定的 eBPF 程序类型中使用，以减少通用类型中冗余的 helper 函数，避免权限过大带来的安全隐患。
3. 可以自由设定 eBPF 程序的输入数据结构。现有的 eBPF 程序类型可能无法完全满足需求。同时，使用自定义的程序类型也能够提高语义清晰度，便于后续的维护和持续开发。

4.1.3 简单请求的 eBPF 挂载点设置

绝大部分的 FUSE 请求（以简单的元数据请求为主），在向上调用的过程中会经过函数 fuse_simple_request，我们在此处统一设置挂载点，根据该 FUSE 请求会话控制块（fc）和具体请求内容和结构（req），作为 eBPF 程序的输入，触发对应的 eBPF 程序。

若 eBPF 程序正确执行并返回正确，则在该函数中可以直接返回，若失败则触发回退机制，即继续原始 FUSE 的工作路径，保证请求应答的正确性。

```
ssize_t fuse_simple_request(struct fuse_mount *fm, struct fuse_args
*args)
{
    struct fuse_conn *fc = fm->fc;
    struct fuse_req *req;
    ssize_t ret;

    if (args->force) {
        atomic_inc(&fc->num_waiting);
        req = fuse_request_alloc(fm, GFP_KERNEL | __GFP_NOFAIL);

        if (!args->nocreds)
            fuse_force_creds(req);

        __set_bit(FR_WAITING, &req->flags);
        __set_bit(FR_FORCE, &req->flags);
    } else {
        WARN_ON(args->nocreds);
        req = fuse_get_req(fm, false);
        if (IS_ERR(req))
            return PTR_ERR(req);
    }
```

```
22
        /* Needs to be done after fuse_get_req() so that fc->minor is
23
    valid */
24      fuse_adjust_compat(fc, args);
25      fuse_args_to_req(req, args);
26
27      // 调用相应的eBPF程序
28      if ((ret = efuse_request_send(fc, req)) != -ENOSYS)
29          return ret;
30
31      if (!args->noreply)
32          __set_bit(FR_ISREPLY, &req->flags);
33      __fuse_request_send(req);
34      ret = req->out.h.error;
35      if (!ret && args->out_argvar) {
36          BUG_ON(args->out_numargs == 0);
37          ret = args->out_args[args->out_numargs - 1].size;
38      }
39      fuse_put_request(req);
40
41      return ret;
42  }
```

4.1.4 特殊请求的 eBPF 挂载点设置

对于部分特殊、相对复杂的 FUSE 请求，并不会经过 fuse_simple_request 函数，需要特殊处理，这里以常见的 READ 请求为例说明。

首先，可以参照 fuse_simple_request 函数，设计类似的 fuse_read_request 函数，保证各类 FUSE 请求对应内核挂载点的统一性，便于后续维护处理。

```
    ssize_t fuse_read_request(struct fuse_mount *fm, struct fuse_args
1
    *args)
2   {
3       struct fuse_conn *fc = fm->fc;
4       struct fuse_req *req;
5
6       if (args->force) {
7           atomic_inc(&fc->num_waiting);
8           req = fuse_request_alloc(fm, GFP_KERNEL | __GFP_NOFAIL);
9
10          if (!args->nocreds)
11              fuse_force_creds(req);
12
13          __set_bit(FR_WAITING, &req->flags);
14          __set_bit(FR_FORCE, &req->flags);
15      } else {
16          WARN_ON(args->nocreds);
17          req = fuse_get_req(fm, false);
18          if (IS_ERR(req))
19              return PTR_ERR(req);
20      }
21
```

```
    /* Needs to be done after fuse_get_req() so that fc->minor is
valid */
    fuse_adjust_compat(fc, args);
    fuse_args_to_req(req, args);

    return efuse_request_send(fc, req);
}
```

分析 READ 请求的函数调用路径，选择在 fuse_file_read_iter 函数的位置插入挂载点，由于各参数的数据结构与先前其他的 FUSE 请求不一致，需要进行一定的处理，同时，部分参数（如 count）在当前阶段并未获取，需要提前处理计算。

调用先前的 fuse_read_request 函数触发 READ 相关的 eBPF 程序，若成功得到结果，则直接填入返回区域（to），若失败，则同样继续原始 FUSE 的工作路径，保证请求应答的正确性和完整性。

```
static ssize_t fuse_file_read_iter(struct kiocb *iocb, struct
iov_iter *to)
{
    struct file *file = iocb->ki_filp;
    struct fuse_file *ff = file->private_data;
    struct inode *inode = file_inode(file);

    if (fuse_is_bad(inode))
        return -EIO;

    if (FUSE_IS_DAX(inode))
        return fuse_dax_read_iter(iocb, to);

    if (!(ff->open_flags & FOPEN_DIRECT_IO)) {

        /* ======== EFUSE hook for read start ======== */
        struct fuse_io_priv io = FUSE_IO_PRIV_SYNC(iocb);
        struct file *file2 = io.iocb->ki_filp;
        struct fuse_file *ff2 = file2->private_data;
        struct fuse_mount *fm = ff2->fm;
        struct fuse_conn *fc = fm->fc;
        unsigned int max_pages = iov_iter_npages(to, fc->max_pages);
        struct fuse_io_args *ia = fuse_io_alloc(&io, max_pages);
        loff_t pos = iocb->ki_pos;
        size_t count = min_t(size_t, fc->max_read,
iov_iter_count(to));
        fl_owner_t owner = current->files;

        fuse_read_args_fill(ia, file2, pos, count, FUSE_READ);
        struct fuse_args *args = &ia->ap.args;
        args->in_numargs = 2;
        args->in_args[1].size = sizeof(file2);
        args->in_args[1].value = &file2;
        if (owner != NULL) {
            ia->read.in.read_flags |= FUSE_READ_LOCKOWNER;
```

```
            ia->read.in.lock_owner = fuse_lock_owner_id(fc, owner);
        }

        //  分配输出缓冲区供 BPF 写入
        void *bpf_output_buf = kzalloc(count, GFP_KERNEL);
        if (!bpf_output_buf) {
            kfree(ia);  // 清理已分配 fuse_io_args
            goto fallback;
        }
        args->out_args[0].size = count;
        args->out_args[0].value = bpf_output_buf;
        args->out_numargs = 1;
        ssize_t ret = fuse_read_request(fm, args);

        // 如果 BPF 成功处理 read 请求，直接从 args->out 中获取数据
        if (ret >= 0) {
            void *data = args->out_args[0].value;
            size_t data_size = args->out_args[0].size;

            if (data && data_size > 0) {
                ssize_t copied = copy_to_iter(data, data_size, to);
                iocb->ki_pos += copied;
                kfree(bpf_output_buf);
                kfree(ia);
                return copied;
            } else {
                kfree(bpf_output_buf);
                kfree(ia);
                return 0;
            }
        }
        kfree(bpf_output_buf);
        kfree(ia);
        /* ======== EFUSE hook for read end ======== */

        fallback:
        return fuse_cache_read_iter(iocb, to);
    } else {
        return fuse_direct_read_iter(iocb, to);
    }
}
```

对于 WRITE 等其他较为复杂的 FUSE 请求，也需要做类似的特殊处理。

4.1.5 相关 helper 函数实现

由于 eBPF 程序中较为严格的验证器限制，后续目标功能的实现收到一定阻碍，为此，需要在 FUSE 内核驱动模块中设计并注册相关的 helper 函数，便于后续 eBPF 程序的实现。

需要使用 helper 函数辅助实现（即无法直接用 eBPF 程序实现的功能）的部分主要集中在 FUSE I/O 请求优化部分，由于需要处理和返回的字符串较大，eBPF 验

证器会对指针、栈等操作作出限制，故 map 缓存路径相关的字符串拼接操作需要通过 helper 函数辅助完成。

另外，由于直通路径对安全性和边界情况较为敏感，故选择把直通相关的操作统一包装到 helper 函数中完成，一方面可以在其中较为灵活的操作字符串指针，另一方面可以集成包装严格的边界限制和安全性限制，避免文件读取越界等情况发生。

```
if (type == READ_MAP_CACHE) {
    if (size != sizeof(struct efuse_cache_in))
        return -EINVAL;

    struct efuse_cache_in *in = (struct efuse_cache_in *)src;

    memcpy(req->out.args[0].value + in->copied, in->data->data + in->data_offset, in->copy_len);

    req->out.args[0].size = in->copied + in->copy_len;
    return in->copied + in->copy_len;
}
```

```
if (type == READ_PASSTHROUGH) {
    if (size != sizeof(struct efuse_read_in))
        return -EINVAL;

    struct efuse_read_in *in = (struct efuse_read_in *)src;

    if (!req || in->size <= 0)
        return -EINVAL;

    if (req->in.numargs < 2) {
        return -EINVAL;
    }
    struct file *filp = *(struct file **)req->in.args[1].value;
    if (!filp) {
        return -EINVAL;
    }

    loff_t file_size = i_size_read(file_inode(filp));
    if (in->offset >= file_size) {
        req->out.args[0].size = 0;
        return 0; // 读取偏移超出文件大小，返回0表示EOF
    }

    if (in->size <= 0) {
        req->out.args[0].size = 0;
        return 0;
    }

    size_t to_read = in->size;
    if (in->offset + to_read > file_size)
        to_read = file_size - in->offset;

```

```
    if (numargs < 1 || req->out.args[0].size < to_read) {
        return -EINVAL;
    }

    if (in->offset + to_read > file_size) {
        return -EINVAL;
    }

    outptr = req->out.args[0].value;
    loff_t pos = in->offset;
    ret = kernel_read(filp, outptr, to_read, &pos); // 需要严格限制读取范围

    if (ret < 0) {
        memset(outptr, 0, in->size);
        return ret;
    }

// 更新实际读取的大小
    req->out.args[0].size = ret;
    return ret;
}
```

4.2 FUSE 元数据请求优化

在 FUSE 协议中，元数据请求指不涉及具体文件内容读写，而是访问、修改文件系统结构或属性的请求操作，通常用于访问 inode、文件/目录路径、目录结构、权限、时间戳、符号链接等元信息。

在诸多 FUSE 元数据请求中，我们为其中出现频次较高的（或有必要的）请求操作进行绕过优化。通过在请求指令进入用户态文件系统前的位置挂载对应的 eBPF 函数，当 VFS 发出对应的请求时，就触发该 eBPF 函数。

为了实现绕过操作，需要使用 eBPF map 实现元数据内容的缓存。在各个 eBPF 函数内部通过查找 eBPF map 的方式尝试对请求进行快速处理，若完成请求即可直接返回结果并实现用户态文件系统的调用。同时，需要在内核态（eBPF 程序）和用户态（文件系统）中协调配合，以保证 eBPF map 中数据的正确性和高命中率。

对于各个 FUSE 请求对应的 eBPF 函数，主要的功能分为访问 map、维护 map 两类，在初赛阶段我们对如下 FUSE 元数据请求设计了单独的 eBPF 函数。

表 4-1　FUSE 请求的 eBPF 函数设计

FUSE 请求	操作码	说明
LOOKUP	1	访问 map 并绕过。
GETATTR	3	访问 map 并绕过。
SETATTR	4	维护 map。
GETXATTR	22	访问 map 并绕过。
FLUSH	25	维护 map 并绕过。
RENAME	12	维护 map。
RMDIR	11	维护 map。
UNLINK	10	维护 map。
READ	15	特殊处理，实现绕过。
WRITE	16	特殊处理，可选绕过。

4.2.1　eBPF map 实现

为实现上述 FUSE 元数据请求的绕过功能，设计如下两个 eBPF map，用来存储文件相关元数据，并可以在 eBPF 程序中快速读取并返回，实现绕过功能。

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, MAX_ENTRIES);
    __type(key, lookup_entry_key_t);
    __type(value, lookup_entry_val_t);
    __uint(map_flags, BPF_F_NO_PREALLOC);
} entry_map SEC(".maps");

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, MAX_ENTRIES);
    __type(key, lookup_attr_key_t);
    __type(value, lookup_attr_val_t);
    __uint(map_flags, BPF_F_NO_PREALLOC);
} attr_map SEC(".maps");
```

定义了 entry_map 和 attr_map 两个 eBPF map，分别用于存储文件或目录的 inode 信息和属性信息。

entry_map 可以实现通过父目录 inode 和文件名，获取子文件（目录）inode 以及相关 flags 标识符信息，可以用来在 eBPF 程序中快速判断路径项是否存在，是否可以提前命中并返回，用于在 LOOKUP 等 FUSE 请求时实现快速处理并绕过的功能。该 map 在 UNLINK 等请求的 eBPF 程序和用户态文件系统中的 CREATE 等请求中都需要进行维护，保证该 map 的正确性和时效性。

attr_map 可以通过文件或目录的 inode 来获取该文件或目录的属性以及缓存控制信息，可以用来快速获取 inode 的属性缓存，判断该 inode 是否有效等，用于在 LOOKUP、GETATTR 等 FUSE 请求时实现快速处理并绕过的功能。该 map 在用户态文件系统中的 CREATE 等请求中都需要进行维护，保证该 map 的正确性和时效性。

4.2.2 eBPF 程序实现

首先为诸多 FUSE 请求对应的 eBPF 程序设计管理函数 handler，当 FUSE 请求在内核进入挂载点时，统一会进入该 handler 函数，在该函数内通过操作符 opcode 进一步确认需要执行的 eBPF 程序并通过 bpf_tail_call 调用，其中 bpf_tail_call 不会返回。若不存在相应操作符的 eBPF 程序，则会直接退回到原始 FUSE 的逻辑。

```
int SEC("efuse") fuse_main_handler(void *ctx)
{
    struct efuse_req *args = (struct efuse_req *)ctx;
    __u32 opcode = 0;
    bpf_core_read(&opcode, sizeof(opcode), &args->in.h.opcode);
    bpf_tail_call(ctx, &handlers, opcode);
    return UPCALL;
}
```

这里以 GETATTR 为例来说明 eBPF 函数如何实现绕过逻辑。首先从输入的 req 结构中获取对应文件的 inode 信息，并查找 attr_map，若成功找到对应信息且信息有效，则可以直接返回给 VFS 从而不需要进一步执行用户态文件系统逻辑，减少了内核/用户态切换的次数并大大加速了请求处理速度。

```
HANDLER(FUSE_GETATTR)(void *ctx)
{
    lookup_attr_key_t key = {0};
    int ret = gen_attr_key(ctx, IN_PARAM_0_VALUE, "GETATTR", &key);
    if (ret < 0)
        return UPCALL;

    /* get cached attr value */
    lookup_attr_val_t *attr = bpf_map_lookup_elem(&attr_map, &key);
    if (!attr)
        return UPCALL;

    /* check if the attr is stale */
    if (attr->stale) {
        /* what does the caller want? */
        struct fuse_getattr_in inarg;
        ret = bpf_efuse_read_args(ctx, IN_PARAM_0_VALUE, &inarg,
sizeof(inarg));
        if (ret < 0)
            return UPCALL;

```

```
        /* check if the attr that the caller wants is stale */
        if (attr->stale & inarg.dummy)
            return UPCALL;
    }

    /* populate output */
    ret = bpf_efuse_write_args(ctx, OUT_PARAM_0, &attr->out,
sizeof(attr->out));
    if (ret)
        return UPCALL;

    return RETURN;
}
```

4.3 FUSE I/O 请求优化

FUSE I/O 请求区别于先前的 FUSE 元数据请求，这类请求针对文件具体内容的操作，即对时间文件字节数据的读写，以常见的 READ、WRITE 请求为代表。这类请求的数据量通常较大，通常发生在打开文件后，在常见的负载情况下，对整体文件系统的性能影响更大。

对 FUSE I/O 请求的绕过处理更为复杂，一方面，文件内容的长度不固定且跨度较大，通常远大于元数据请求，简单的 map 缓存机制难以覆盖实际场景。另一方面，对于某些特定的负载场景，如对单一文件的频繁读写、连续对不同文件的读写等，map 的命中率显著降低，性能下降，甚至出现额外开销过大的问题。

为解决上述情况，我们设计了两条绕过路径：map 缓存路径和直通路径，同时设计自适应调度算法，使系统能够根据先前的工作情况，预测并判断哪条路径更快实现请求，从而使该系统在各种负载情况下都能实现较高的性能，在确保接口灵活性的同时，大大提升 I/O 性能和稳定性。

4.3.1 map 缓存路径

为实现上述 eBPF map 缓存路径，设计如下的 eBPF map，用来存储文件数据，并可以在 eBPF 程序中快速读取并返回，实现绕过功能。

```
struct {
    __uint(type, BPF_MAP_TYPE_LRU_HASH);
    __uint(max_entries, MAX_ENTRIES);
    __type(key, read_data_key_t);
    __type(value, read_data_value_t);
} read_data_map SEC(".maps");
```

read_data_map 可以通过 file_handle 和 offset，获取对应文件中 4KB 大小的字节块，其中 offset 必须以 4KB 对齐。在 eBPF 函数中，可以通过查找 read_data_map 获取

问价内容，并进行拼接得到请求所需的内容并直接返回，即可实现用户态文件系统的绕过操作。在 READ、WRITE 对应的 eBPF 程序中，以及 WRITE 的用户态文件系统操作中，同样需要维护 read_data_map，保证其中数据的正确性和有效性。

设计并实现 read_from_cache 函数，封装实现 map 缓存功能。根据输入的 file_handle、offset、size，判断需要在 read_data_map 中查询的数据块，获取相应数据后，在 helper 函数的辅助下完成拼接并返回。若成功获取结果，即对 map 的访问均命中且未发生其他问题，则可以直接返回并完成绕过，若未能成功，则退回到原始 FUSE 逻辑，或尝试直通路径。

```
static __always_inline
int read_from_cache(void *ctx, uint64_t fh, uint64_t offset,
uint32_t size)
{
    read_data_key_t data_key = {};
    uint64_t copied = 0;
    uint64_t aligned_offset = offset & ~(uint64_t)
(DATA_MAX_BLOCK_SIZE - 1);
    uint64_t end_offset = (offset + size + DATA_MAX_BLOCK_SIZE - 1)
& ~(uint64_t)(DATA_MAX_BLOCK_SIZE - 1);
    uint64_t off = aligned_offset;

    for (int i = 0; i < MAX_LOOP_COUNT; i++) {
        data_key.file_handle = fh;
        data_key.offset = off;

        read_data_value_t *data =
bpf_map_lookup_elem(&read_data_map, &data_key);
        if (!data) {
            return -1;
        }

        uint64_t data_offset = (off == aligned_offset) ? (offset -
aligned_offset) : 0;
        if (data_offset >= data->size || data->size == 0) {
            struct efuse_cache_in bpf_cache_in = {
                .copied = copied,
                .data_offset = data_offset,
                .copy_len = 0,
                .data = data
            };
            int ret = bpf_efuse_write_args(ctx, READ_MAP_CACHE,
&bpf_cache_in, sizeof(bpf_cache_in));
            if (ret < 0)
                return -1;
            return ret;
        }

```

```
        uint32_t copy_len = data->size - data_offset;
        if (copied + copy_len > size)
            copy_len = size - copied;

        if (copied + copy_len > size)
            return -1;

        struct efuse_cache_in bpf_cache_in = {
            .copied = copied,
            .data_offset = data_offset,
            .copy_len = copy_len,
            .data = data
        };
        int ret = bpf_efuse_write_args(ctx, READ_MAP_CACHE,
&bpf_cache_in, sizeof(bpf_cache_in));
        if (ret < 0)
            return -1;

        copied += copy_len;

        if (data->is_last || data->size < DATA_MAX_BLOCK_SIZE)
            break;

        off += DATA_MAX_BLOCK_SIZE;
        if (off >= end_offset)
            break;
    }

    return 0;
}
```

4.3.2 直通路径

使用 map 缓存路径优化能够有效提高 FUSE I/O 请求的性能，但是在某些典型的负载模式下，例如对单一大文件的高频、连续性读写或对多个小文件的快速、频繁、分散性的 I/O 请求等，任然使用 map 缓存方案会出现 map 命中率低，性能明显下降的问题。

为解决上述问腿，我们提出直通路径。即在 eBPF 程序中直接尝试读取磁盘内容并返回。由于 eBPF 验证器的指针限制，该过程同样需要设计合适的内核 helper 函数辅助完成。同时，上述过程需要严格限制磁盘的访问范围，以保障系统的安全性。

设计并实现 read_passthrough 函数，封装实现直通绕过的功能，根据输入的 file_handle、offset、size 并通过 helper 函数辅助完成直接从磁盘获取文件内容的操作，由于直通操作对安全性和边界情况比较敏感，故直通路径的大部分逻辑被封装到 helper 函数中实现，其中对边界条件和文件读取范围进行了十分严格的限制，避免用户的误操作导致的安全漏洞和系统崩溃，helper 函数的具体实现见 **4.1.5 相关 helper 函数实现**。

```c
static __always_inline
int read_passthrough(void *ctx, uint64_t fh, uint64_t offset,
uint32_t size)
{
    struct efuse_read_in bpf_read_in = {
        .fh = fh,
        .offset = offset,
        .size = size
    };
    int ret = bpf_efuse_write_args(ctx, READ_PASSTHROUGH,
&bpf_read_in, sizeof(bpf_read_in));
    if (ret < 0) {
        return -1;
    }
    return 0;
}
```

4.3.3 自适应调度算法

为了使系统能够在不同负载情况下智能选择最合适的绕过路径，我们设计了一套探测+预测型的自适应调度算法。

具体来讲，将若干次 READ 请求设定为一轮，在每轮的前几次请求为初步探测阶段，同时尝试通过两条绕过路径（map 缓存路径和直通路径）完成 FUSE I/O 请求的绕过操作，并记录他们的实际响应实践和性能指标（如延迟、吞吐率、map 命中率等）。在后面的 READ 请求中，首先对前几次收集的性能数据分析，通过历史命中率、实际响应实践等数据，构造简单且有效的时间预测模型，预测走两条路径所需的时间并从中选择合适的路径完成绕过操作。

经过测试，在负载稳定的情况下倾向选择 map 缓存路径，在大跨度、低局部性的请求场景将倾向于使用直通路径。通过这样的自适应调度算法，实现了系统在不同的负载情况下都能维持较高的性能，适应不同类型的负载，能够极大改善文件系统的 I/O 性能和稳定性。

```c
HANDLER(FUSE_READ)(void *ctx)
{
    int ret;
    struct fuse_read_in readin;
    u32 pid = bpf_get_current_pid_tgid() >> 32;
    ret = bpf_efuse_read_args(ctx, IN_PARAM_0_VALUE, &readin,
sizeof(readin));
    if (ret < 0)
        return UPCALL;

    lookup_attr_key_t key = {0};
    ret = gen_attr_key(ctx, IN_PARAM_0_VALUE, "READ", &key);
```

```
    if (ret < 0)
        return UPCALL;

    /* get cached attr value */
    lookup_attr_val_t *attr = bpf_map_lookup_elem(&attr_map, &key);
    if (!attr)
        return UPCALL;

#ifndef HAVE_PASSTHRU
    if (attr->stale & FATTR_ATIME)
        return UPCALL;
#endif

    uint64_t file_handle = readin.fh;
    uint64_t offset = readin.offset;
    uint32_t size = readin.size;

    // 调度选择部分
    u32 stat_key = 0;
    read_stat_t *stat = bpf_map_lookup_elem(&read_stat_map,
&stat_key);
    if (!stat)
        return UPCALL;

    // 前 TEST_CNT 次: 探测阶段
    if (stat->total_cnt < TEST_CNT) {
        __u64 t1 = bpf_ktime_get_ns();
        int r1 = read_from_cache(ctx, file_handle, offset, size);
        __u64 t2 = bpf_ktime_get_ns();
        if (r1 == 0) {
            stat->cache_time_sum += (t2 - t1);
            stat->cache_cnt++;
        }

        t1 = bpf_ktime_get_ns();
        int r2 = read_passthrough(ctx, file_handle, offset, size);
        t2 = bpf_ktime_get_ns();
        if (r2 == 0) {
            stat->passthrough_time_sum += (t2 - t1);
            stat->passthrough_cnt++;
        }

        stat->total_cnt++;

        if (r1 == 0)
            return RETURN;
        if (r2 == 0)
            return RETURN;
        return UPCALL;
    }

    // 选择阶段
    if (stat->total_cnt == TEST_CNT) {
        if ( stat->cache_cnt != stat->passthrough_cnt ) {
            stat->prefer_cache = stat->cache_cnt > stat-
>passthrough_cnt; // 1:缓存 0:直通
        } else {
```

```
            __u64 avg_cache = stat->cache_time_sum / (stat->cache_cnt ?: 1);
            __u64 avg_pt = stat->passthrough_time_sum / (stat->passthrough_cnt ?: 1);
            stat->prefer_cache = avg_cache < avg_pt; // 1:缓存 0:直通
        }
    }

    // 后续轮内请求，使用选中的路径
    stat->total_cnt++;

    if (stat->prefer_cache) {
        ret = read_from_cache(ctx, file_handle, offset, size);
    } else {
        ret = read_passthrough(ctx, file_handle, offset, size);
    }

    if (stat->total_cnt > ROUND_CNT) {
        // 重置统计信息
        stat->cache_time_sum = 0;
        stat->passthrough_time_sum = 0;
        stat->cache_cnt = 0;
        stat->passthrough_cnt = 0;
        stat->total_cnt = 0;
    }

    if (ret == 0) {
        return RETURN;
    }

#ifdef HAVE_PASSTHRU
    return RETURN;
#else
    return UPCALL;
#endif
}
```

4.4 多核扩展模块

4.5 负载监控与请求均衡

5 项目测试

5.1 单核测试

5.2 多核测试

6 总结与展望

参考文献