



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

FUSE（Filesystem in Userspace）是目前广受欢迎的一个用户态文件系统框架，但 FUSE 在高频负载场景下受限于频繁的用户态/内核态切换和请求提交时的锁争用，性能表现不佳。

为此，我们提出 eFuse，一个结合 eBPF 技术的 FUSE 性能优化方案。其中包括通过 eBPF map 实现文件元数据和文件内容在内核中的缓存，在内核 eBPF 程序中对 FUSE 请求快速处理，设计更为合理的内核请求队列结构。优化请求处理和调度管理，提升各个负载场景下 FUSE 的性能和可扩展性，初赛阶段性能提升约 3 倍。上述设计不改变原本 FUSE 的架构和接口标准，能够实现对现有 FUSE 的完全兼容。

我们设计五大技术目标模块以及目前完成情况如下：

- 目标 1：FUSE 内核模块扩展
- 目标 2：FUSE 元数据请求优化
- 目标 3：FUSE I/O 请求的特殊优化
- 目标 4：基于内核修改的多核优化
- 目标 5：负载监控与请求均衡

目标编号	完成情况	说明
1	100%	1. 完成对 FUSE 内核模块的扩展。 2. 后续可能随着需求变化，进一步扩展。
2	100%	1. 通过 eBPF 在内核快速处理 FUSE 请求。 2. 优化 inode、目录、权限、路径等相关操作。
3	100%	1. 针对文件 I/O 请求的页数优化 2. 读写性能提升 1.5~3 倍，平均延迟显著降低。
4	80%	1. 为每个核心构建独立 ringbuf 管道。 2. 实现多核 CPU 环境的适配。
5	20%	1. 利用 eBPF 动态分析请求负载并进行策略调整。
总计	80%	1. 完成对 FUSE 请求处理路径、调度管理的优化。 2. 实现对 FUSE 工作性能的大幅提升。 3. 将持续完善现有工作并考虑新增其他优化模块。

# 目 录

摘要 .....	II
<b>1 概述 .....</b>	<b>1</b>
1.1 背景及意义 .....	1
1.2 目标 .....	1
1.3 行动项 .....	2
1.4 完成情况 .....	3
1.5 开发历程 .....	4
1.6 团队分工 .....	6
<b>2 现有研究调研 .....</b>	<b>7</b>
2.1 ExtFUSE .....	7
2.2 FUSE-BPF .....	7
2.3 RFUSE .....	8
<b>3 整体架构设计 .....</b>	<b>9</b>
3.1 设计理念 .....	9
3.2 设计架构介绍 .....	10
<b>4 模块设计和实现 .....</b>	<b>11</b>
4.1 FUSE 内核模块扩展 .....	11
4.2 FUSE 元数据请求优化 .....	18
4.3 FUSE I/O 请求优化 .....	21
4.4 多核优化模块 .....	27
<b>5 性能测试 .....</b>	<b>39</b>
5.1 综合性能测试 .....	39
<b>6 总结与展望 .....</b>	<b>42</b>
参考文献 .....	44

# 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[1] 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. 支持直通路经：eBPF 直接读取文件内容。 2. 支持缓存路经：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经选择 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 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

在初赛阶段,我们团队的开发历程如下:

表 1-2 开发历程表

日期	开发内容
<b>start - 3.12</b>	<ol style="list-style-type: none"><li>1. 小组调研讨论赛题。</li><li>2. 确认选题,构思项目优化方向,讨论可行性问题。</li><li>3. 深度学习 eBPF 原理以及开发流程。</li><li>4. 收集近些年有关 FUSE 性能优化的相关工作。</li><li>5. 初步搭建虚拟机开发环境,选择基于的内核版本。</li></ol>
<b>3.13 - 3.26</b>	<ol style="list-style-type: none"><li>1. 调研 ExtFUSE、RFUSE 等相关研究。</li><li>2. 阅读 FUSE 优化相关论文,阅读相关开源项目代码。</li><li>3. 尝试复现 ExtFUSE、JFUSE 项目,对其做初步性能测试。</li><li>4. 与往届学长和老师讨论后续优化方向。</li><li>5. 调研 FUSE-BPF 项目,了解堆栈式文件系统思想。</li><li>6. 阅读 lanbda-io 论文,了解计算存储和 eBPF 扩展优化。</li><li>7. 初步确定三大优化方向。</li></ol>
<b>3.27 - 4.9</b>	<ol style="list-style-type: none"><li>1. 根据后续接口需要修改 libfuse 库。</li><li>2. 整理代码框架,搭建项目代码仓库。</li><li>3. 学习 FUSE 的工作流程,阅读内核中 FUSE 相关代码。</li><li>4. 在内核 FUSE 请求提交路径上设置 eBPF 挂载点。</li><li>5. 能够成功在期望的位置触发 eBPF 程序。</li></ol>
<b>4.10 - 4.23</b>	<ol style="list-style-type: none"><li>1. 尝试初步实现部分 FUSE 请求的绕过操作</li><li>2. 使用 eBPF map 实现数据在内核 eBPF 程序中的缓存。</li><li>3. 对目前现有的问题进行 debug。</li></ol>

4.24 - 5.14	<ol style="list-style-type: none"><li>1. 尝试初步实现部分 FUSE 请求的绕过操作</li><li>2. 使用 eBPF map 实现数据在内核 eBPF 程序中的缓存。</li><li>3. 对目前现有的问题进行 debug。</li></ol>
4.24 - 5.14	<ol style="list-style-type: none"><li>1. 初步实现重要的元数据请求的 eBPF 绕过程序。</li><li>2. 进行阶段性性能测试，性能有一定程度的提升。</li><li>3. 梳理 read 请求的内核提交路径，选取合适的挂载点。</li><li>4. 初步完成 I/O 请求 map 缓存通道的架构，存在栈溢出。</li><li>5. 考虑修改内核，对 eBPF 验证器做放宽限制处理。</li></ol>
5.15 - 6.4	<ol style="list-style-type: none"><li>1. 考虑多核优化方向，调研 RFUSE 相关项目。</li><li>2. 决定使用 eBPF helper 解决指针限制等问题。</li><li>3. 为直通路径设计实现设计并注册 eBPF helper 函数。</li><li>4. 针对基于的用户态文件系统做协调修改，维护 map。</li></ol>
6.5 - 6.18	<ol style="list-style-type: none"><li>1. 完成直通路径和 map 缓存路径的实现。</li><li>2. 基于两条绕过路径，设计并实现绕过算法。</li><li>3. 考虑将对用户态文件系统修改做模块化处理。</li><li>4. 做阶段性性能测试，性能大幅优化。</li></ol>
6.19 - 6.30	<ol style="list-style-type: none"><li>1. 进一步对内核 FUSE 部分做修改，尝试实现多核优化。</li><li>2. 为每个核心构建环形管道，代替原本的请求队列。</li><li>3. 需要进一步修改 libfuse 库，适配后续的多核优化工作。</li><li>4. 设计三种常见的模拟负载场景，进行综合性能测试。</li><li>5. 进行撰写文档、完善仓库等初赛准备工作。</li></ol>



## 1.6 团队分工

在初赛阶段，我们团队的分工如下：

表 1-4 团队分工表

姓名	分工内容
许辰涛	<ul style="list-style-type: none"><li>• 调研现有 FUSE 优化方案</li><li>• eBPF 开发技术栈学习</li><li>• 项目仓库与依赖子仓库搭建</li><li>• ExtFUSE 的 Linux 内核移植</li><li>• 将 libfuse 适配 eFUSE</li></ul>
冯可逸	<ul style="list-style-type: none"><li>• 开发环境搭建</li><li>• eBPF 开发技术栈学习</li><li>• 使用 eBPF 实现 FUSE 请求绕过</li><li>• FUSE I/O 缓存/直通 实现</li><li>• 初赛阶段综合性测试</li></ul>
赵胜杰	<ul style="list-style-type: none"><li>• 理解 rfuse 算法思想和代码架构</li><li>• 理解 extfuse 算法思想和代码架构</li><li>• 复现 rfuse 项目</li><li>• 在 6.5.0 内核上修改内核 fuse 模块</li><li>• 修改用户库 libfuse 整合 rfuse 和 extfuse</li></ul>

## 2 现有研究调研

该部分简要阐述了对现有的 FUSE 优化方案的调研，分析其优缺点和适用场景，为 eFUSE 的设计提供参考，更加详细的内容参见 eFUSE 项目主仓库下 doc 文件夹。

### 2.1 ExtFUSE

ExtFUSE 是一个基于 Linux FUSE 文件系统的性能增强框架，核心思路是在 `fuse_request_send` 函数中挂载自定义 eBPF 程序，通过内核态快速判断请求是否可以直接响应，从而避免进入用户态，提高元数据请求（如 `lookup/getattr`）的处理效率。[2]其实现依赖于对 Linux 内核的修改，包括新增 eBPF 程序类型 `BPF_PROG_TYPE_EXTFUSE`、在 FUSE 初始化流程中加载程序，并通过 `extfuse_request_send` 尝试优先执行 eBPF 路径。当请求命中缓存时，直接构造响应返回；若未命中，则退回至常规用户态流程。

ExtFUSE 使用 eBPF map 存储 inode 和 dentry 的元数据信息，结合两个辅助函数 `bpf_extfuse_read_args` 和 `bpf_extfuse_write_args`，实现对 FUSE 请求参数和结果的读写操作。在用户态，文件系统需在初始化时加载 eBPF 程序，并在创建或访问文件时更新 map 以维持缓存有效性。该框架显著降低了上下文切换频率，提升了文件系统在只读类请求下的性能。但由于其依赖自定义内核、修改较多，部署成本较高，且对非元数据请求的优化有限，在实际应用中仍需权衡使用场景与维护复杂度。

### 2.2 FUSE-BPF

FUSE-BPF 是一种将 FUSE 扩展为堆栈式文件系统的实验性方案，利用 eBPF 技术在内核中实现对文件系统请求的预处理与后处理。与传统 FUSE 不同，FUSE-BPF 允许请求无需进入用户态即可由内核直接处理，从而降低上下文切换开销。其核心思路是将 eBPF 程序嵌入到 FUSE 请求路径中，对请求内容进行拦截、分析和快速处理，或在请求完成后修改结果，实现如路径过滤、元数据缓存、访问控制等功能。

FUSE-BPF 的工作模式类似堆栈式文件系统：用户请求首先被堆叠的 FUSE 层捕获，eBPF 程序决定是否处理或修改该请求，随后再传递到底层绑定文件系统（backing file system）完成实际操作。该设计重用已有文件系统的核心功能，仅在逻

辑路径上插入灵活的处理机制，具有良好的可扩展性与组合能力。当前实现仍处于原型阶段，支持的操作类型较少，程序编写复杂，且缺乏统一的加载与安全机制，实际部署尚不成熟。但该方案为内核态处理用户态文件系统请求提供了新的思路，对 eFUSE 的功能路径设计和可编程扩展机制具有借鉴意义。

## 2.3 RFUSE

RFUSE 是一个针对 FUSE 性能瓶颈的优化框架，旨在提升其在多核系统上的并发处理能力与传输效率。[3]RFUSE 采用内核态与用户态之间的环形缓冲区通信机制，代替传统基于系统调用的请求队列方式。每个 CPU 核心对应一个独立的环形通道，RFUSE 守护进程通过 `mmap()` 将这些通道映射到用户空间，从而实现高效、低延迟的共享内存通信。

该设计借鉴了 `io_uring` 的双缓冲结构理念，引入提交队列（SQ）与完成队列（CQ）的思想，在每个管道中设置专属工作线程，处理来自内核的请求并向内核写回结果。同时，RFUSE 提供了混合轮询机制：在短时间内进行忙轮询，若无响应则自动进入休眠状态，从而在 CPU 占用与延迟之间取得平衡。

与原生 FUSE 架构相比，RFUSE 避免了共享 `pending queue` 引发的锁争用问题，显著提升了高并发下的扩展性与吞吐率。其调度策略也更具弹性，针对异步请求密集的情况可自动跨核重分配任务，缓解单核压力。此外，RFUSE 保持与 FUSE 完全兼容，仅需文件系统连接 `librfuse` 即可使用，部署成本低、工程可行性高。

相较于其他方案，RFUSE 避免了 FUSE-passthrough 的功能受限问题，不依赖于 eBPF 的编程限制，也未采用 XFUSE [4] 中“多通道但无调度”的片面策略，体现了对现代硬件架构和系统负载的良好适配性。其通信架构、线程模型与缓冲队列设计为 eFuse 项目提供了参考。

## 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：

- **性能优先**：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

传统 FUSE 的工作流程和优化后的 eFuse 工作流程如下图所示：

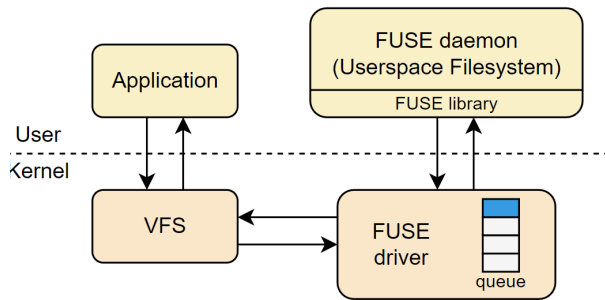


图 3-1 FUSE 工作流程示意图

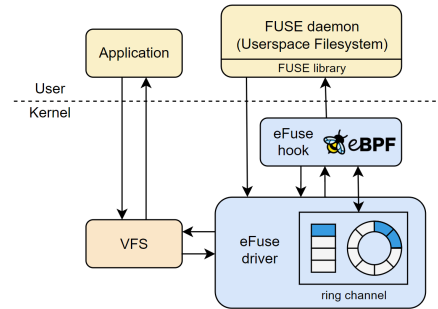


图 3-2 eFuse 工作流程示意图

与原始 FUSE 工作流程相比，eFuse 在内核态对 FUSE 内核驱动进行了针对性的改造和扩展，同时新增 eBPF 模块，设置 eBPF 挂载点和 eBPF 程序，从而提升 FUSE 请求的处理效率和可扩展性。eFuse 核心设计架构包括以下几个关键部分：

- **eBPF 程序**

在内核态预先设计 eBPF 程序挂载点，针对常见的文件系统操作提供专用的快速处理通道。eBPF 程序在挂载点最请求参数和元数据进行快速检查，在命中场景下直接构造回复响应，快速返回结果，避免传统 FUSE 的内核态/用户态切换。

- **专用环形管道设计**

在每个 CPU 核心上独立分配环形管道(ring channel)，完全代替传统 FUSE 的共享请求队列。每个核心独立处理自己的请求，避免锁的集中争用和上下文同步开销，显著改善高负载场景下的性能。

- **负载监控与均衡**

利用 eBPF 的动态追踪能力，实时监控各个核心的请求负载情况。根据负载情况动态调整请求分配策略，实现请求的均衡分配，避免某个核心过载而其他核心空闲。[5]

- **完善的回退机制**

当某个请求不适合在 eBPF 路径处理，或 eBPF 执行过程中发生错误，eFuse 会自动回退到传统 FUSE 的工作路径，确保请求能够正确处理并返回结果。回退机制保证了系统的健壮性和稳定性。

## 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。

```

1 struct fuse_init_out {
2     uint32_t    major;
3     uint32_t    minor;
4     uint32_t    max_readahead;
5     uint32_t    flags;
6     uint16_t    max_background;
7     uint16_t    congestion_threshold;
8     uint32_t    max_write;
9     uint32_t    time_gran;
10    uint16_t    max_pages;
11    uint16_t    map_alignment;
12    uint32_t    flags2;
13    uint32_t    max_stack_depth;
14    uint32_t    efuse_prog_fd; //添加字段
15    uint32_t    unused[5];
16 };

```

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。

```

1 static void process_init_reply(struct fuse_mount *fm, struct
2 fuse_args *args,
3                               int error)
4 {
5     struct fuse_conn *fc = fm->fc;
6     struct fuse_init_args *ia = container_of(args, typeof(*ia),
7 args);

```

```

6     struct fuse_init_out *arg = &ia->out;
7     bool ok = true;
8
9     if (error || arg->major != FUSE_KERNEL_VERSION)
10        ok = false;
11    else {
12        unsigned long ra_pages;
13
14        process_init_limits(fc, arg);
15
16        if (arg->minor >= 6) {
17            .....
18            if (flags & FUSE_FS_EFUSE) {
19                // 加载 eBPF 程序
20                efuse_load_prog(fc, arg->efuse_prog_fd);
21            }
22        }
23        .....
24    }
25 }

```

```

1 int efuse_load_prog(struct fuse_conn *fc, int fd)
2 {
3     struct bpf_prog *prog = NULL;
4     struct bpf_prog *old_prog;
5     struct efuse_data *data;
6
7     BUG_ON(fc->fc_priv);
8
9     data = kmalloc(sizeof(*data), GFP_KERNEL);
10    if (!data)
11        return -ENOMEM;
12
13    prog = bpf_prog_get(fd);
14    if (IS_ERR(prog)) {
15        kfree(data);
16        return -1;
17    }
18
19    old_prog = xchg(&data->prog, prog);
20    if (old_prog)
21        bpf_prog_put(old_prog);
22
23    fc->fc_priv = (void *)data;
24    return 0;
25 }

```

#### 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。

```

1  ssize_t fuse_simple_request(struct fuse_mount *fm, struct fuse_args
2  *args)
3  {
4      struct fuse_conn *fc = fm->fc;
5      struct fuse_req *req;
6      ssize_t ret;
7
8      if (args->force) {
9          atomic_inc(&fc->num_waiting);
10         req = fuse_request_alloc(fm, GFP_KERNEL | __GFP_NOFAIL);
11
12         if (!args->nocreds)
13             fuse_force_creds(req);
14
15         __set_bit(FR_WAITING, &req->flags);
16         __set_bit(FR_FORCE, &req->flags);
17     } else {
18         WARN_ON(args->nocreds);
19         req = fuse_get_req(fm, false);
20         if (IS_ERR(req))
21             return PTR_ERR(req);
22     }

```



```

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。

```

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

```

```

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     return efuse_request_send(fc, req);
27 }

```

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。

```

1  static ssize_t fuse_file_read_iter(struct kiocb *iocb, struct
2  iov_iter *to)
3  {
4      struct file *file = iocb->ki_filp;
5      struct fuse_file *ff = file->private_data;
6      struct inode *inode = file_inode(file);
7
8      if (fuse_is_bad(inode))
9          return -EIO;
10
11     if (FUSE_IS_DAX(inode))
12         return fuse_dax_read_iter(iocb, to);
13
14     if (!(ff->open_flags & FOPEN_DIRECT_IO)) {
15         /* ===== EFUSE hook for read start ===== */
16         struct fuse_io_priv io = FUSE_IO_PRIV_SYNC(iocb);
17         struct file *file2 = io.iocb->ki_filp;
18         struct fuse_file *ff2 = file2->private_data;
19         struct fuse_mount *fm = ff2->fm;
20         struct fuse_conn *fc = fm->fc;
21         unsigned int max_pages = iov_iter_npages(to, fc->max_pages);
22         struct fuse_io_args *ia = fuse_io_alloc(&io, max_pages);
23         loff_t pos = iocb->ki_pos;
24         size_t count = min_t(size_t, fc->max_read,
25         iov_iter_count(to));
26         fl_owner_t owner = current->files;
27
28         fuse_read_args_fill(ia, file2, pos, count, FUSE_READ);
29         struct fuse_args *args = &ia->ap.args;
30         args->in_numargs = 2;
31         args->in_args[1].size = sizeof(file2);
32         args->in_args[1].value = &file2;
33         if (owner != NULL) {
34             ia->read.in.read_flags |= FUSE_READ_LOCKOWNER;

```

```

34         ia->read.in.lock_owner = fuse_lock_owner_id(fc, owner);
35     }
36
37     // 分配输出缓冲区供 BPF 写入
38     void *bpf_output_buf = kzalloc(count, GFP_KERNEL);
39     if (!bpf_output_buf) {
40         kfree(ia); // 清理已分配 fuse_io_args
41         goto fallback;
42     }
43     args->out_args[0].size = count;
44     args->out_args[0].value = bpf_output_buf;
45     args->out_numargs = 1;
46     ssize_t ret = fuse_read_request(fm, args);
47
48     // 如果 BPF 成功处理 read 请求, 直接从 args->out 中获取数据
49     if (ret >= 0) {
50         void *data = args->out_args[0].value;
51         size_t data_size = args->out_args[0].size;
52
53         if (data && data_size > 0) {
54             ssize_t copied = copy_to_iter(data, data_size, to);
55             iocb->ki_pos += copied;
56             kfree(bpf_output_buf);
57             kfree(ia);
58             return copied;
59         } else {
60             kfree(bpf_output_buf);
61             kfree(ia);
62             return 0;
63         }
64     }
65     kfree(bpf_output_buf);
66     kfree(ia);
67     /* ===== EFUSE hook for read end ===== */
68
69     fallback:
70     return fuse_cache_read_iter(iocb, to);
71 } else {
72     return fuse_direct_read_iter(iocb, to);
73 }
74 }

```

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。

```

1  if (type == READ_MAP_CACHE) {
2      if (size != sizeof(struct efuse_cache_in))
3          return -EINVAL;
4
5      struct efuse_cache_in *in = (struct efuse_cache_in *)src;
6
7      memcpy(req->out.args[0].value + in->copied, in->data->data + in-
8          >data_offset, in->copy_len);
9
10     req->out.args[0].size = in->copied + in->copy_len;
11     return in->copied + in->copy_len;
12 }

```

```

1  if (type == READ_PASSTHROUGH) {
2      if (size != sizeof(struct efuse_read_in))
3          return -EINVAL;
4
5      struct efuse_read_in *in = (struct efuse_read_in *)src;
6
7      if (!req || in->size <= 0)
8          return -EINVAL;
9
10     if (req->in.numargs < 2) {
11         return -EINVAL;
12     }
13     struct file *filp = *(struct file **)req->in.args[1].value;
14     if (!filp) {
15         return -EINVAL;
16     }
17
18     loff_t file_size = i_size_read(file_inode(filp));
19     if (in->offset >= file_size) {
20         req->out.args[0].size = 0;
21         return 0; // 读取偏移超出文件大小，返回0表示EOF
22     }
23
24     if (in->size <= 0) {
25         req->out.args[0].size = 0;
26         return 0;
27     }
28
29     size_t to_read = in->size;
30     if (in->offset + to_read > file_size)
31         to_read = file_size - in->offset;
32

```

```

33     if (numargs < 1 || req->out.args[0].size < to_read) {
34         return -EINVAL;
35     }
36
37     if (in->offset + to_read > file_size) {
38         return -EINVAL;
39     }
40
41     outptr = req->out.args[0].value;
42     loff_t pos = in->offset;
43     ret = kernel_read(filp, outptr, to_read, &pos); // 需要严格限制读取
范围
44
45     if (ret < 0) {
46         memset(outptr, 0, in->size);
47         return ret;
48     }
49
50     // 更新实际读取的大小
51     req->out.args[0].size = ret;
52     return ret;
53 }

```

## 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。

```

1 struct {
2     __uint(type, BPF_MAP_TYPE_HASH);
3     __uint(max_entries, MAX_ENTRIES);
4     __type(key, lookup_entry_key_t);
5     __type(value, lookup_entry_val_t);
6     __uint(map_flags, BPF_F_NO_PREALLOC);
7 } entry_map SEC(".maps");
8
9 struct {
10    __uint(type, BPF_MAP_TYPE_HASH);
11    __uint(max_entries, MAX_ENTRIES);
12    __type(key, lookup_attr_key_t);
13    __type(value, lookup_attr_val_t);
14    __uint(map_flags, BPF_F_NO_PREALLOC);
15 } 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。

```

1 int SEC("efuse") fuse_main_handler(void *ctx)
2 {
3     struct efuse_req *args = (struct efuse_req *)ctx;
4     __u32 opcode = 0;
5     bpf_core_read(&opcode, sizeof(opcode), &args->in.h.opcode);
6     bpf_tail_call(ctx, &handlers, opcode);
7     return UPCALL;
8 }

```

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。

```

1 HANDLER(FUSE_GETATTR)(void *ctx)
2 {
3     lookup_attr_key_t key = {0};
4     int ret = gen_attr_key(ctx, IN_PARAM_0_VALUE, "GETATTR", &key);
5     if (ret < 0)
6         return UPCALL;
7
8     /* get cached attr value */
9     lookup_attr_val_t *attr = bpf_map_lookup_elem(&attr_map, &key);
10    if (!attr)
11        return UPCALL;
12
13    /* check if the attr is stale */
14    if (attr->stale) {
15        /* what does the caller want? */
16        struct fuse_getattr_in inarg;
17        ret = bpf_efuse_read_args(ctx, IN_PARAM_0_VALUE, &inarg,
18        sizeof(inarg));
19        if (ret < 0)
20            return UPCALL;

```

```

21     /* check if the attr that the caller wants is stale */
22     if (attr->stale & inarg.dummy)
23         return UPCALL;
24 }
25
26 /* populate output */
27 ret = bpf_efuse_write_args(ctx, OUT_PARAM_0, &attr->out,
28 sizeof(attr->out));
29 if (ret)
30     return UPCALL;
31 return RETURN;
32 }

```

### 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 缓存路径和直通过程，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。

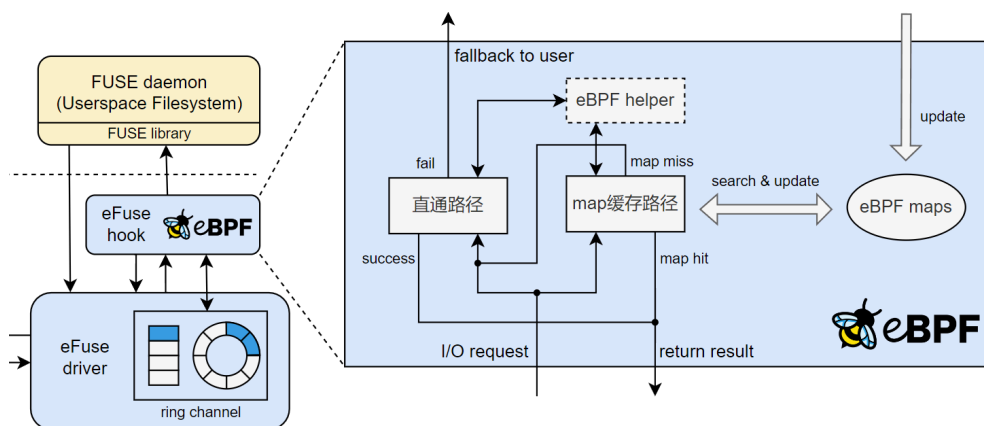


图 4-1 FUSE I/O 请求绕过路径示意图



### 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。

```

1 struct {
2     __uint(type, BPF_MAP_TYPE_LRU_HASH);
3     __uint(max_entries, MAX_ENTRIES);
4     __type(key, read_data_key_t);
5     __type(value, read_data_value_t);
6 } 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，或尝试直通过程。

```

1 static __always_inline
2 int read_from_cache(void *ctx, uint64_t fh, uint64_t offset,
3 uint32_t size)
4 {
5     read_data_key_t data_key = {};
6     uint64_t copied = 0;
7     uint64_t aligned_offset = offset & ~(uint64_t)
8 (DATA_MAX_BLOCK_SIZE - 1);
9     uint64_t end_offset = (offset + size + DATA_MAX_BLOCK_SIZE - 1)
10 & ~(uint64_t)(DATA_MAX_BLOCK_SIZE - 1);
11     uint64_t off = aligned_offset;
12
13     for (int i = 0; i < MAX_LOOP_COUNT; i++) {
14         data_key.file_handle = fh;
15         data_key.offset = off;
16
17         read_data_value_t *data =
18 bpf_map_lookup_elem(&read_data_map, &data_key);
19         if (!data) {
20             return -1;
21         }
22     }
23 }

```

```

19         uint64_t data_offset = (off == aligned_offset) ? (offset -
aligned_offset) : 0;
20         if (data_offset >= data->size || data->size == 0) {
21             struct efuse_cache_in bpf_cache_in = {
22                 .copied = copied,
23                 .data_offset = data_offset,
24                 .copy_len = 0,
25                 .data = data
26             };
27             int ret = bpf_efuse_write_args(ctx, READ_MAP_CACHE,
&bpf_cache_in, sizeof(bpf_cache_in));
28             if (ret < 0)
29                 return -1;
30             return ret;
31         }
32
33         uint32_t copy_len = data->size - data_offset;
34         if (copied + copy_len > size)
35             copy_len = size - copied;
36
37         if (copied + copy_len > size)
38             return -1;
39
40         struct efuse_cache_in bpf_cache_in = {
41             .copied = copied,
42             .data_offset = data_offset,
43             .copy_len = copy_len,
44             .data = data
45         };
46         int ret = bpf_efuse_write_args(ctx, READ_MAP_CACHE,
&bpf_cache_in, sizeof(bpf_cache_in));
47         if (ret < 0)
48             return -1;
49
50         copied += copy_len;
51
52         if (data->is_last || data->size < DATA_MAX_BLOCK_SIZE)
53             break;
54
55         off += DATA_MAX_BLOCK_SIZE;
56         if (off >= end_offset)
57             break;
58     }
59
60     return 0;
61 }

```

### 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。

为解决上述问题，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。

```
1 static __always_inline
2 int read_passthrough(void *ctx, uint64_t fh, uint64_t offset,
3   uint32_t size)
4 {
5     struct efuse_read_in bpf_read_in = {
6         .fh = fh,
7         .offset = offset,
8         .size = size
9     };
10    int ret = bpf_efuse_write_args(ctx, READ_PASSTHROUGH,
11      &bpf_read_in, sizeof(bpf_read_in));
12    if (ret < 0) {
13        return -1;
14    }
15    return 0;
16 }
```

### 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。

```

1  HANDLER(FUSE_READ)(void *ctx)
2  {
3      int ret;
4      struct fuse_read_in readin;
5      u32 pid = bpf_get_current_pid_tgid() >> 32;
6      ret = bpf_efuse_read_args(ctx, IN_PARAM_0_VALUE, &readin,
7      sizeof(readin));
8      if (ret < 0)
9          return UPCALL;
10
11     lookup_attr_key_t key = {0};
12     ret = gen_attr_key(ctx, IN_PARAM_0_VALUE, "READ", &key);
13     if (ret < 0)
14         return UPCALL;
15
16     /* get cached attr value */
17     lookup_attr_val_t *attr = bpf_map_lookup_elem(&attr_map, &key);
18     if (!attr)
19         return UPCALL;
20
21     #ifndef HAVE_PASSTHRU
22         if (attr->stale & FATTR_ATIME)
23             return UPCALL;
24     #endif
25
26     uint64_t file_handle = readin.fh;
27     uint64_t offset = readin.offset;
28     uint32_t size = readin.size;
29
30     // 调度选择部分
31     u32 stat_key = 0;
32     read_stat_t *stat = bpf_map_lookup_elem(&read_stat_map,
33     &stat_key);
34     if (!stat)
35         return UPCALL;
36
37     // 前 TEST_CNT 次：探测阶段
38     if (stat->total_cnt < TEST_CNT) {
39         __u64 t1 = bpf_ktime_get_ns();
40         int r1 = read_from_cache(ctx, file_handle, offset, size);
41         __u64 t2 = bpf_ktime_get_ns();
42         if (r1 == 0) {
43             stat->cache_time_sum += (t2 - t1);
44             stat->cache_cnt++;
45         }
46
47         t1 = bpf_ktime_get_ns();
48         int r2 = read_passthrough(ctx, file_handle, offset, size);
49         t2 = bpf_ktime_get_ns();
50         if (r2 == 0) {
51             stat->passthrough_time_sum += (t2 - t1);
52             stat->passthrough_cnt++;
53         }
54     }
55 }

```

```

52
53     stat->total_cnt++;
54
55     if (r1 == 0)
56         return RETURN;
57     if (r2 == 0)
58         return RETURN;
59     return UPCALL;
60 }
61
62 // 选择阶段
63 if (stat->total_cnt == TEST_CNT) {
64     if ( stat->cache_cnt != stat->passthrough_cnt ) {
65         stat->prefer_cache = stat->cache_cnt > stat-
66 >passthrough_cnt; // 1:缓存 0:直通
67     } else {
68         __u64 avg_cache = stat->cache_time_sum / (stat-
69 >cache_cnt ? 1);
70         __u64 avg_pt = stat->passthrough_time_sum / (stat-
71 >passthrough_cnt ? 1);
72         stat->prefer_cache = avg_cache < avg_pt; // 1:缓存 0:直通
73     }
74 }
75
76 // 后续轮内请求，使用选中的路径
77 stat->total_cnt++;
78
79 if (stat->prefer_cache) {
80     ret = read_from_cache(ctx, file_handle, offset, size);
81 } else {
82     ret = read_passthrough(ctx, file_handle, offset, size);
83 }
84
85 if (stat->total_cnt > ROUND_CNT) {
86     // 重置统计信息
87     stat->cache_time_sum = 0;
88     stat->passthrough_time_sum = 0;
89     stat->cache_cnt = 0;
90     stat->passthrough_cnt = 0;
91     stat->total_cnt = 0;
92 }
93
94 if (ret == 0) {
95     return RETURN;
96 }
97
98 #ifdef HAVE_PASSTHRU
99     return RETURN;
100 #else
101     return UPCALL;
102 #endif
103 }

```

## 4.4 多核优化模块

为实现多核优化，需要对内核 FUSE 驱动模块进行扩展和修改，同时考虑接口问题也需要对用户层 libfuse 也进行扩展和修改。

具体修改内容包括：替换原始 fuse 请求队列，引入了多个请求队列（如挂起队列、中断队列、忘记队列和完成队列），修改所有内核 fuse 有关请求函数，编写轮询函数，修改用户层有关请求函数。这些内核和用户层面的改动旨在为用户态和内核态的 FUSE 文件系统提供接口支持和运行环境，确保后续的多核机制能够正确工作。

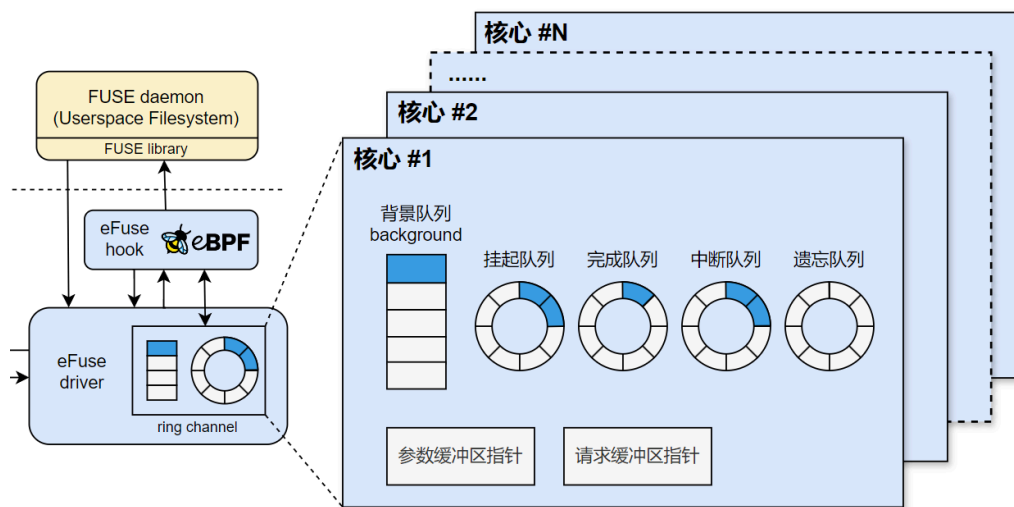


图 4-2 eFuse 请求队列

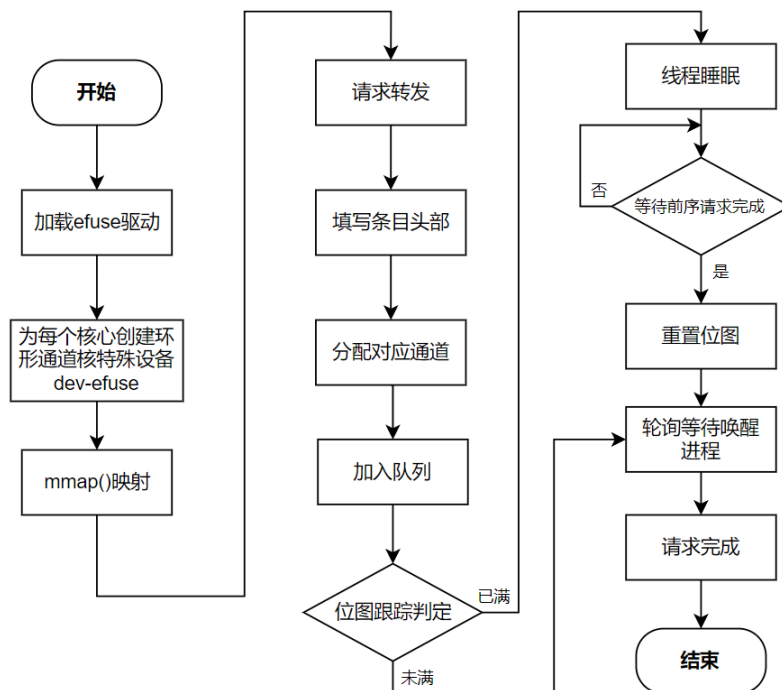


图 4-3 eFuse 多核优化模块工作流程图

#### 4.4.1 队列优化

原始 fuse 请求队列结构体 `struct fuse_iqueue` 是 FUSE 模块中的核心数据结构之一，用于管理文件系统请求队列。它包含多个字段来跟踪请求的状态、同步和异步操作等。例如，`connected` 字段表示连接是否已建立，`lock` 字段用于保护结构体成员的并发访问，`waitq` 用于让进程等待可用请求，`reqctr` 用于生成唯一的请求 ID，`pending` 列表用于存储挂起的请求等。此外，它还包含设备特定的回调和私有数据字段，以支持不同设备的特定功能。

```

1  static __always_inline
2  struct fuse_iqueue {
3      /** Connection established */
4      unsigned connected;
5
6      /** Lock protecting accesses to members of this structure */
7      spinlock_t lock;
8
9      /** Readers of the connection are waiting on this */
10     wait_queue_head_t waitq;
11
12     /** The next unique request id */
13     u64 reqctr;
14
15     /** The list of pending requests */
16     struct list_head pending;
17
18     /** Pending interrupts */
19     struct list_head interrupts;
20
21     /** Queue of pending forgets */
22     struct fuse_forget_link forget_list_head;
23     struct fuse_forget_link *forget_list_tail;
24
25     /** Batching of FORGET requests (positive indicates FORGET
26     batch) */
27     int forget_batch;
28
29     /** 0_ASYNC requests */
30     struct fasync_struct *fasync;
31
32     /** Device-specific callbacks */
33     const struct fuse_iqueue_ops *ops;
34
35     /** Device-specific state */
36     void *priv;
37 };

```

新 `efuse` 请求队列结构体是 `efuse` 模块中的核心数据结构，用于管理增强型文件系统请求队列。它包含多个环形缓冲区，用于分别处理请求、中断、忘记和完成操作。这些环形缓冲区可以提高数据流处理的效率，减少内存分配和释放的开销。

它还具有动态参数缓冲区和动态请求缓冲区，支持用户空间和内核空间之间的数据传递。此外，efuse\_iqueue 包括位图机制来管理缓冲区状态，以及复杂的阻塞控制字段来实现同步请求和背景请求的拥塞控制，适用于更复杂的场景和性能要求。

```

1 struct efuse_iqueue {
2     int riq_id;
3     /** Pending queue */
4     struct ring_buffer_1 pending;
5     /** Interrupt queue */
6     struct ring_buffer_2 interrupts;
7     /** Forget queue */
8     struct ring_buffer_3 forgets;
9     /** Complete queue */
10    struct ring_buffer_1 completes;
11
12    /** Dyanmic argument buffer */
13    struct efuse_arg *uarg; // user address
14    struct efuse_arg *karg; // kernel address
15
16    /** Dynamic request buffer */
17    struct efuse_req *ureq; // user address
18    struct efuse_req *kreq; // kernel address
19
20    /** Connection established */
21    unsigned connected;
22
23    /** wait queue for requests to wait to receive a request buffer
24    */
25    wait_queue_head_t waitq;
26
27    /** Lock protecting accesses to members of this structure */
28    spinlock_t lock;
29
30    /** The next unique request id */
31    u64 reqctr;
32
33    /** Device specific state */
34    void *priv;
35
36    struct {
37        unsigned long bitmap_size;
38        unsigned full;
39        unsigned long *bitmap;
40    } argbm;
41
42    struct {
43        unsigned long bitmap_size;
44        unsigned full;
45        unsigned long *bitmap;
46    } reqbm;
47
48    wait_queue_head_t idle_user_waitq;
49
50    /** synchronous request congestion control */
51    int num_sync_sleeping;

```



```

51
52  /** background request congestion control */
53  struct list_head bg_queue;
54  spinlock_t bg_lock;
55
56  unsigned max_background;
57  unsigned congestion_threshold;
58  unsigned num_background;
59  unsigned active_background;
60  int blocked;
61
62  /** waitq for congested asynchronous requests*/
63  wait_queue_head_t blocked_waitq;
64  };

```

上述新 efuse 结构体与传统 FUSE 的区别：传统 FUSE 通常只有一个请求队列，而 Efuse 引入了多个队列，以支持更复杂的请求管理。Efuse 使用位图来管理参数和请求缓冲区的分配和释放，这在传统 FUSE 中是不存在的。

例如，挂起队列环形缓冲区（ring\_buffer\_1）和后台队列的条目结构（efuse\_bg\_entry）结构体如下所示，环形缓冲区支持内核空间 and 用户空间的共享，通过 kaddr 和 uaddr 分别表示内核地址和用户地址。这种设计允许用户空间直接访问和操作缓冲区，减少了内核和用户空间之间的数据拷贝，提高了性能。

```

1  struct ring_buffer_1 {
2      uint32_t tail;
3      uint32_t head;
4      uint32_t mask;
5      uint32_t entries;
6      struct efuse_address_entry *kaddr; // kernel address
7      struct efuse_address_entry *uaddr; // user address
8  };
9
10 struct efuse_bg_entry {
11     struct list_head list;
12     uint32_t request;
13     int32_t riq_id;
14 };

```

原始 fuse 请求结构 struct fuse\_req 是 FUSE 中用于表示请求的核心结构体，其构造过程主要包括初始化 list 和 intr\_entry 字段为链表头，分配并初始化 args 字段，设置 count 字段为 1，清零 flags 字段，填写 in 字段中的请求输入 header 信息，初始化 out 字段为默认值，调用 init\_waitqueue\_head 函数初始化 waitq 字段，在启用 virtio-fs 时分配物理连续缓冲区给 argbuf 字段，最后将所属的 fuse\_mount 结构体指针赋值给 fm 字段。整个构造过程根据具体的请求类型和需求进行相应的初始化设置，为后续的请求处理做好准备。

```

1  struct fuse_req {
2      /** This can be on either pending processing or io lists in
3          fuse_conn */
4      struct list_head list;
5
6      /** Entry on the interrupts list */
7      struct list_head intr_entry;
8
9      /** Input/output arguments */
10     struct fuse_args *args;
11
12     /** refcount */
13     refcount_t count;
14
15     /** Request flags, updated with test/set/clear_bit() */
16     unsigned long flags;
17
18     /** The request input header */
19     struct {
20         struct fuse_in_header h;
21     } in;
22
23     /** The request output header */
24     struct {
25         struct fuse_out_header h;
26     } out;
27
28     /** Used to wake up the task waiting for completion of request*/
29     wait_queue_head_t waitq;
30
31     #if IS_ENABLED(CONFIG_VIRTIO_FS)
32         /** virtio-fs's physically contiguous buffer for in and out args
33         */
34         void *argbuf;
35     #endif
36
37     /** fuse_mount this request belongs to */
38     struct fuse_mount *fm;
39 };

```

新 efuse 请求结构 struct efuse\_req 定义了 符合 efuse 队列的请求结构，包含请求的输入头 (in)、输出头 (out)、请求缓冲区索引 (index)、队列 ID (riq\_id)、请求标志 (flags)、引用计数 (count)、等待队列 (waitq) 等。包含了请求的参数空间 (args) 和布尔标志位 (如 force、noreply、nocreds 回调函数指针 (end))，用于处理请求完成后的操作。

主要区别：增加了更多的字段和标志位，例如 eFuse 引入了引用计数 (refcount\_t count) 和等待队列 (wait\_queue\_head\_t waitq)，用于更好地管理请求的生命周期和同步操作。

```

1  struct efuse_req {
2      /** Request input header **/
3      struct {
4          uint64_t unique;
5          uint64_t nodeid;
6          uint32_t opcode;
7          uint32_t uid;
8          uint32_t gid;
9          uint32_t pid;
10         uint32_t arg[2];    // Location of in operation-specific
11         argument
12         uint32_t arglen[2]; // Size of in operation-specific argument
13     } in;                    // 48
14     /** Request output header **/
15     struct {
16         int32_t error;
17         uint32_t arg;    // Location of out operation-specific argument
18         uint32_t arglen; // Size of out operation-specific argument
19         uint32_t padding;
20     } out; // 16
21
22     /** request buffer index **/
23     uint32_t index; // 4
24     int32_t riq_id;
25     /** Request flags, updated with test/set/clear_bit() **/
26     unsigned long flags; // 8
27
28     /** fuse_mount this request belongs to **/
29     struct fuse_mount *fm; // 8
30     /** refcount **/
31     refcount_t count; // 4
32     /** Used to wake up the task waiting for completion of request **/
33     wait_queue_head_t waitq; // 24
34
35     struct {
36         uint8_t argument_space[112];
37     } args; // 112
38
39     bool force : 1;
40     bool noreply : 1;
41     bool nocreds : 1;
42     bool in_pages : 1;
43     bool out_pages : 1;
44     bool out_argvar : 1;
45     bool page_zeroing : 1;
46     bool page_replace : 1;
47     bool may_block : 1;
48
49     struct efuse_pages *rp;
50     void (*end)(struct fuse_mount *fm, struct efuse_req *r_req, int
51     error);
52 };

```

#### 4.4.2 内核 FUSE 有关请求函数

**efuse\_send\_init 函数**，用于向用户空间发送 Fuse 文件系统的初始化请求。Fuse 文件系统的初始化过程，使内核空间能够与用户空间的文件系统进行通信协商，确定双方支持的功能和参数，从而建立有效的文件系统连接。

通过 `efuse_get_req` 函数获取一个请求结构体。然后，构造初始化请求的参数，包括协议版本、最大读取提前大小等，并设置各种功能支持的标志位，如异步读取、POSIX 锁定、大写入等。接着，设置请求的类型为 `FUSE_INIT`，并将该请求标记为后台和异步请求。最后，调用 `efuse_simple_background` 函数来异步发送这个初始化请求。如果发送失败，则直接调用 `efuse_process_init_reply` 函数来处理这个失败的情况，传递一个错误码。

```

1 void efuse_send_init(struct fuse_mount *fm)
2 {
3     struct efuse_req *r_req;
4     struct fuse_init_in *inarg;
5
6     r_req = efuse_get_req(fm, true, true);
7     inarg = (struct fuse_init_in *)&r_req->args;
8
9     inarg->major = FUSE_KERNEL_VERSION;
10    inarg->minor = FUSE_KERNEL_MINOR_VERSION;
11    inarg->max_readahead = fm->sb->s_bdi->ra_pages * PAGE_SIZE;
12    inarg->flags |=
13        FUSE_ASYNC_READ | FUSE_POSIX_LOCKS | FUSE_ATOMIC_O_TRUNC |
14        FUSE_EXPORT_SUPPORT | FUSE_BIG_WRITES | FUSE_DONT_MASK |
15        FUSE_SPLICE_WRITE | FUSE_SPLICE_MOVE | FUSE_SPLICE_READ |
16        FUSE_FLOCK_LOCKS | FUSE_HAS_IOCTL_DIR | FUSE_AUTO_INVAL_DATA
17
18        FUSE_DO_READDIRPLUS | FUSE_READDIRPLUS_AUTO | FUSE_ASYNC_DIO
19
20        FUSE_WRITEBACK_CACHE | FUSE_NO_OPEN_SUPPORT |
21        FUSE_PARALLEL_DIROPS | FUSE_HANDLE_KILLPRIV | FUSE_POSIX_ACL
22
23        FUSE_ABORT_ERROR | FUSE_MAX_PAGES | FUSE_CACHE_SYMLINKS |
24        FUSE_NO_OPENDIR_SUPPORT | FUSE_EXPLICIT_INVAL_DATA |
25        FUSE_HANDLE_KILLPRIV_V2 | FUSE_SETXATTR_EXT |
26        EXT_FUSE_FLAGS;
27    #ifdef CONFIG_FUSE_DAX
28        if (fm->fc->dax)
29            inarg->flags |= FUSE_MAP_ALIGNMENT;
30    #endif
31    if (fm->fc->auto_submounts)
32        inarg->flags |= FUSE_SUBMOUNTS;
33
34    r_req->in.opcode = FUSE_INIT;
35    __set_bit(FR_BACKGROUND, &r_req->flags);
36    __set_bit(FR_ASYNC, &r_req->flags);

```

```

34     r_req->end = efuse_process_init_reply;
35
36     if (efuse_simple_background(fm, r_req) != 0)
37     {
38         pr_info("fuse_send_init ia->out: %d", ia-
>out.extfuse_prog_fd);
39         printk("EFUSE: efuse_send_init: efuse_simple_background
failed\n");
40         efuse_process_init_reply(fm, r_req, -ENOTCONN);
41     }
42 }

```

### efuse\_lookup\_init 函数

**efuse\_lookup\_init** 函数是 efuse 文件系统中用于初始化查找操作的函数。这个函数的主要作用是初始化一个查找操作的请求，将查找所需的信息（如父目录节点 ID 和文件名）封装到请求对象中，以便后续发送给用户空间的文件系统进行处理。起到了关键的桥梁作用，连接了内核空间的请求发起和用户空间的请求处理。

```

static void efuse_lookup_init(struct fuse_mount *fm, struct
1 efuse_req *r_req, u64 nodeid, const struct qstr *name){
2     struct efuse_arg* arg;
3     struct efuse_iqueue *riq = efuse_get_specific_iqueue(fm->fc,
r_req->riq_id);
4     unsigned int in_arg = efuse_get_argument_buffer(fm, r_req-
>riq_id);
5     unsigned int out_arg = efuse_get_argument_buffer(fm, r_req-
>riq_id);
6
7     arg = (struct efuse_arg*)&riq->karg[in_arg];
8     memset(arg, 0, sizeof(struct efuse_arg));
9
10    // Copy the name into argument space
11    memcpy(arg, (char*)name->name, name->len+1);
12
13    r_req->in.opcode = FUSE_LOOKUP;
14    r_req->in.nodeid = nodeid;
15    r_req->in.arglen[0] = name->len+1;
16    r_req->in.arg[0] = in_arg;
17    r_req->out.arg = out_arg;
18    r_req->out.arglen = sizeof(struct fuse_entry_out);
19 }

```

#### 4.4.3 轮询函数

在 efuse 文件系统中，轮询功能的实现主要依赖于 `select_round_robin` 函数和 `efuse_get_iqueue_for_async` 函数的协作。`select_round_robin` 函数通过维护一个全局

的原子变量 `rr_id` 来记录当前轮询到的队列索引，每次调用时会更新该变量并返回下一个队列索引，从而实现循环选取多个 `efuse_iqueue` 队列的目的。

而 `efuse_get_iqueue_for_async` 函数则直接调用 `select_round_robin` 函数来获取队列索引，并根据该索引从 `fuse_conn` 的 `riq` 数组中取出对应的 `efuse_iqueue` 对象，为异步请求提供轮询选择的队列。这种设计使得多个请求能够均匀地分布在不同的 `efuse_iqueue` 上，从而实现负载均衡。

- **select\_round\_robin 函数**

```

1 static int select_round_robin(struct fuse_conn *fc){
2     int ret;
3
4     spin_lock(&fc->lock);
5
6     if(atomic_read(&rr_id) == EFUSE_NUM_QUEUE)
7         atomic_set(&rr_id, 0);
8
9     ret = atomic_read(&rr_id);
10    atomic_add(1, &rr_id);
11    spin_unlock(&fc->lock);
12
13    return ret;
14 }
```

- **efuse\_get\_iqueue\_for\_async 函数**

```

1 struct efuse_iqueue *efuse_get_iqueue_for_async(struct fuse_conn *fc)
2 {
3     int id = 0;
4     id = select_round_robin(fc);
5
6     return fc->riq[id];
7 }
```

#### 4.4.4 用户层有关请求函数

**efuse\_read\_queue 函数**是 `efuse` 文件系统的核心组件，负责从共享内存空间中读取请求（包括普通请求、中断请求和忘记请求），并调用相应的处理函数进行处理。它首先根据 `forget` 参数决定处理忘记请求还是挂起请求，然后从对应的队列中提取请求，生成用户级请求（ULR）对象，并将其加入请求列表。接着，根据请求的操作码调用相应的处理函数（如 `do_getattr`、`do_lookup` 等）来处理请求。

在处理过程中，它会进行一系列检查，如会话是否初始化、操作是否被允许等，确保请求的合法性和安全性。若请求不合法或出现错误，它会进入错误处理流程，向用户回复错误信息。此函数通过优化请求提取和处理流程、支持多队列操作以及

统一的请求处理逻辑等改进，提升了文件系统的并发处理能力和整体性能，增强了系统的健壮性和可维护性。

```

1  bool efuse_read_queue(struct efuse_worker *w, struct efuse_mt *mt,
2  struct fuse_chan *ch, int forget) {
3      struct fuse_session *se = mt->se;
4      int riq_id = mt->riq_id;
5      struct efuse_iqueue *riq = se->riq[riq_id]; // Iqueue
6      struct efuse_address_entry *target_entry;
7      struct efuse_forget_entry *forget_entry;
8      fuse_req_t u_req; // ULR
9      struct efuse_req *r_req;
10     int err;
11     uint64_t nodeid;
12     bool processed = false;
13
14     if(riq->connected == 0){
15         printf("riq connection is lost, id: %d\n", riq_id);
16         fuse_session_exit(se);
17         return processed;
18     }
19
20     // 1.Forget Requests
21     if(forget == 1){
22         pthread_mutex_lock(&se->riq_lock[riq_id]);
23         forget_entry = efuse_read_forgets_head(riq);
24         if(!forget_entry){
25             pthread_mutex_unlock(&se->riq_lock[riq_id]);
26             // efuse_free_req(u_req);
27             goto out;
28         }
29         u_req = efuse_ll_alloc_req(se, riq_id);
30         u_req->nlookup = forget_entry->nlookup;
31         u_req->unique = forget_entry->unique;
32         u_req->w = w;
33         nodeid = forget_entry->nodeid;
34         efuse_extract_forgets_head(riq);
35         pthread_mutex_unlock(&se->riq_lock[riq_id]);
36
37         efuse_ll_ops[FUSE_FORGET].func(u_req, nodeid);
38         processed = true;
39         return processed;
40     }
41     else{
42         // 2. Pending Requests
43         pthread_mutex_lock(&se->riq_lock[riq_id]);
44         target_entry = efuse_read_pending_head(riq);
45         if(!target_entry){
46             pthread_mutex_unlock(&se->riq_lock[riq_id]);
47             // efuse_free_req(u_req);
48             goto out;
49         }
50         u_req = efuse_ll_alloc_req(se, riq_id);
51         u_req->index = target_entry->request;
52         efuse_extract_pending_head(riq);

```

```

52     r_req = &riq->ureq[u_req->index];
53     assert(r_req->riq_id == u_req->riq_id);
54     u_req->w = w;
55     efuse_list_add_req(u_req, &se->efuse_list[riq_id]);
56     pthread_mutex_unlock(&se->riq_lock[riq_id]);
57     processed = true;
58 }
59 #ifdef DEBUG
60     printf("efuse experiment opcode: %s (%i)\n", efuse_opname((enum
61 fuse_opcode) r_req->in.opcode), r_req->in.opcode);
62 #endif
63     GET_TIMESTAMPS(2)
64     u_req->unique = r_req->in.unique;
65     u_req->ctx.uid = r_req->in.uid;
66     u_req->ctx.gid = r_req->in.gid;
67     u_req->ctx.pid = r_req->in.pid;
68     u_req->ch = ch ? fuse_chan_get(ch) : NULL;
69     err = EIO;
70     if(!se->got_init){
71         enum fuse_opcode expected;
72
73         expected = se->cuse_data ? CUSE_INIT : FUSE_INIT;
74         if (r_req->in.opcode != expected)
75             goto reply_err;
76     } else if (r_req->in.opcode == FUSE_INIT || r_req->in.opcode ==
77 CUSE_INIT)
78         goto reply_err;
79     err = EACCES;
80     if (se->deny_others && r_req->in.uid != se->owner && r_req-
81 >in.uid != 0 &&
82         r_req->in.opcode != FUSE_INIT && r_req->in.opcode !=
83 FUSE_READ &&
84         r_req->in.opcode != FUSE_WRITE && r_req->in.opcode !=
85 FUSE_FSYNC &&
86         r_req->in.opcode != FUSE_RELEASE && r_req->in.opcode !=
87 FUSE_READDIR &&
88         r_req->in.opcode != FUSE_FSYNCDIR && r_req->in.opcode !=
89 FUSE_RELEASEDIR &&
90         r_req->in.opcode != FUSE_NOTIFY_REPLY &&
91         r_req->in.opcode != FUSE_READDIRPLUS)
92         goto reply_err;
93     err = ENOSYS;
94     if (r_req->in.opcode >= FUSE_MAXOP || !efuse_ll_ops[r_req-
95 >in.opcode].func)
96         goto reply_err;
97     GET_TIMESTAMPS(3)
98     if (r_req->in.opcode == FUSE_WRITE && se->op.write_buf) {
99         err = efuse_prep_write_buf(u_req, se, &w->fbuf, w->ch);
100         if(err < 0)

```



```
97         goto reply_err;
98         efuse_do_write_buf(u_req, r_req->in.nodeid, &w->fbuf);
99     } else if (r_req->in.opcode == FUSE_NOTIFY_REPLY) {
100         //do_notify_reply(req, r_req->in.nodeid);
101     } else {
102         efuse_ll_ops[r_req->in.opcode].func(u_req, r_req-
103         >in.nodeid);
104     }
105     return processed;
106 reply_err:
107     fuse_reply_err(u_req, err);
108 out:
109     return processed;
110 }
```

## 5 性能测试

### 5.1 综合性能测试

#### 5.1.1 测试方法

综合性能测试使用 **fio** 工具，针对实际使用场景设计并模拟常见的文件画质情况，用于评估 eFuse 在引入后的性能提升效果。

我们设计了三类典型负载场景：

- **负载测试 1：单个文件的小块随机读写**

考察对单个文件的随机读写操作，模拟实际应用中对小文件的频繁访问情况，如数据库查询场景等。

- **负载测试 2：多个小文件的随机读写**

考察对多个小文件的随机读写操作，模拟实际应用中多个小文件的频繁访问情况，如日志处理场景，网页服务器查询场景等。

- **负载测试 3：多个大文件的分散式随机读写**

考察对多个大文件的分散式随机读写操作，模拟实际应用中大文件的分散访问情况，如视频处理场景和大数据场景等。

每个负载场景下，使用 **fio** 工具进行测试，设置不同的文件大小、读写比例、并发数等参数，以全面评估 eFuse 在不同负载下的性能表现。

以上三组场景均采用**读写比 7:3**的随机读写操作进行测试，贴合实际应用负载，通过比较 IOPS、读写吞吐量、平均延迟以及延迟抖动等性能指标，能够综合评估 eFuse 在不同负载下的性能表现。

本测试以原始 FUSE 和其他相关项目 ExtFUSE 作为性能参照，性能测试基于简易用户态文件系统 StackFS。其中，ExtFUSE 是与本项目类似的其他开源研究，同样旨在利用 eBPF 对 FUSE 的性能做优化，我们尝试对其进行复现，并进行相关测试和对比。

## 5.1.2 测试结果

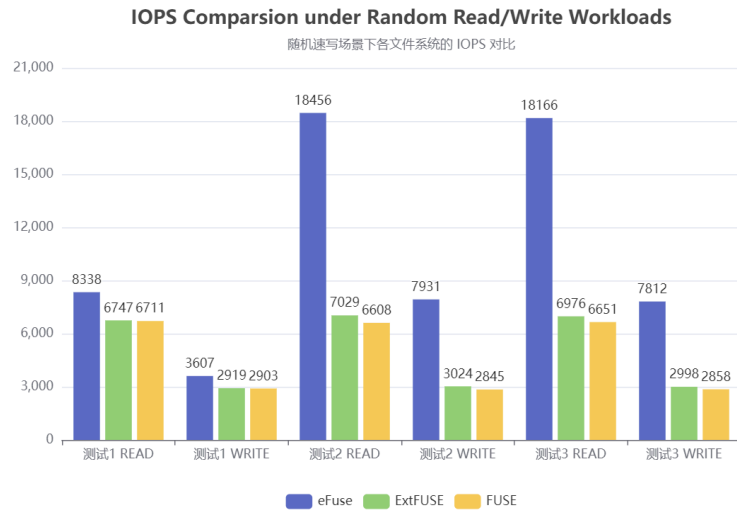


图 5-1 随机读写 IOPS 测试结果

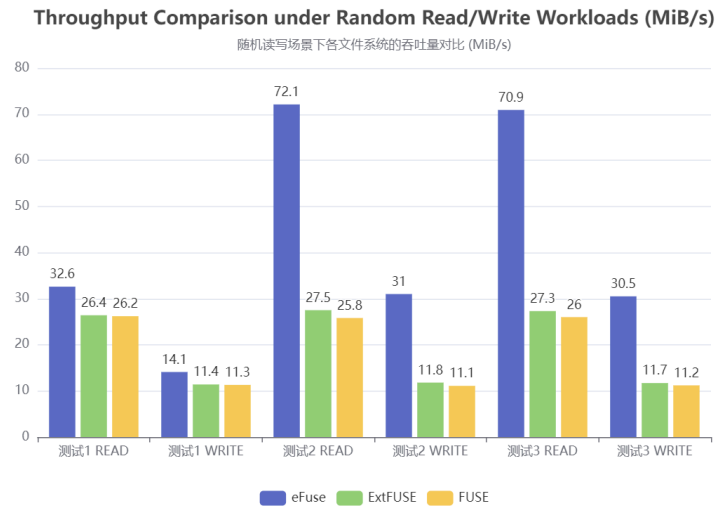


图 5-2 随机读写吞吐量测试结果

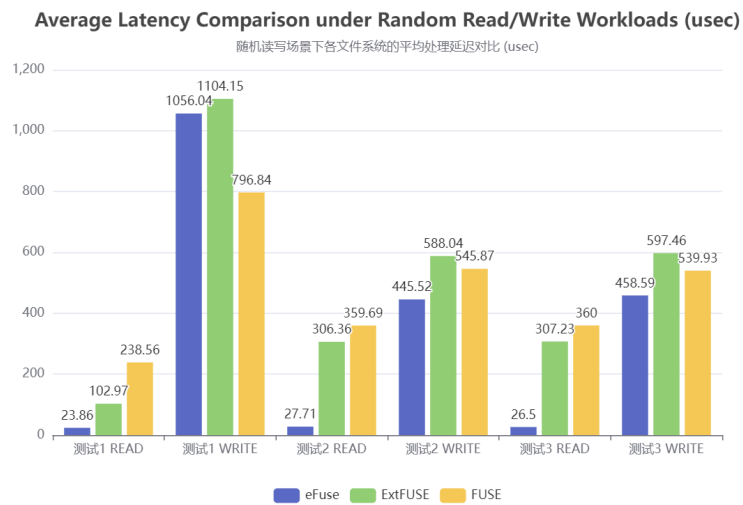


图 5-3 随机读写平均延迟测试结果

### 5.1.3 测试结果分析

从测试结果可以看出，eFuse 在全部三类负载场景下相比 FUSE 和 ExtFUSE 均有显著的性能提升。

在**负载测试 2**（多个小文件的随机读写）中和**负载测试 3**（多个大文件的分散式随机读写）中，eFuse 完全发挥出 eBPF map 缓存并绕过路径的优势，IOPS 和吞吐量均有显著提升，提升约 3 倍，平均延迟和延迟抖动也明显降低。

在**负载测试 1**（单个文件的小块随机读写）中，尽管由于连续对同一个文件进行操作，eBPF map 的缓存命中率略有下降，但 eFuse 依旧凭借直通优化使请求响应速度答复领先于 FUSE 和 ExtFUSE，IOPS 和吞吐量提升约 1.5 倍，平均延迟和延迟抖动也有明显降低。

在写操作方面，受 eBPF map 更新以及同步开销的影响，在单文件负载下，eFuse 的写性能略低于 FUSE 和 ExtFUSE。但结合整体场景计算，这样的开销是完全值得的，吞吐量的提升弥补了这一开销，显示出综合性能的显著改善。

为进一步验证 eFuse 在更高性能目标下的潜力，我们还同时对**内核态文件系统 EXT4** 进行了测试。结果显示，eFuse 在所有负载场景下均能逼近 EXT4，在**负载测试 2**（多个小文件的随机读写），eFuse 借助 eBPF map 缓存和直通路路径的优势，性能在改负载条件下已经超过 EXT4，显示出 eFuse 在高性能文件系统场景下的巨大潜力。

表 5-1 测试 2 下 eFuse 与 EXT4 性能对比

性能指标	eFuse	EXT4
READ IOPS	18456	15122
WRITE IOPS	7931	6481
READ 吞吐	72.1M	58.9M
WRITE 吞吐	31.0M	25.3M

在其他场景下，eFuse 整体性能虽略低于 EXT4，但仍然表现出色，相比 FUSE 和 ExtFUSE 有明显提升。

综合来看，eFuse 在多个实际负载场景中表现出远超前于 FUSE 和 ExtFUSE 的性能，尤其适合小文件、多元混合负载和高并发的场景，能够更好利用现在计算和存储硬件的能力，提供更高的 I/O 性能和更低的延迟。

## 6 总结与展望

eFuse 在初赛阶段，已经实现绝大部分预期功能，包括对 FUSE 请求的绕过优化、环形管道设计等。通过对 FUSE 请求绕过、请求队列的优化，eFuse 在多种负载场景下均表现出色，尤其在小文件和高并发场景下，性能提升显著。

针对 **1.3: 行动项** 部分叙述的五大技术目标模块，在初赛阶段完成进度如下：

表 6-1 目标技术模块

实现内容	完成情况	说明
目标 1: FUSE 内核模块扩展	全部完成 100%	1. 完成对 FUSE 内核模块的扩展。 2. 后续可能随着需求变化，进一步扩展。
目标 2: FUSE 元数据请求优化	全部完成 100%	1. 优化 inode、目录、权限、路径等相关操作。 2. 后续考虑对更多 FUSE 元数据请求设计 eBPF 绕过优化。
目标 3: FUSE I/O 请求的特殊优化	全部完成 100%	1. 支持直通/缓存路径，并实现自适应调度。 2. 读写性能提升 1.5~3 倍。
目标 4: 基于内核修改的多核优化	基本完成 80%	1. 为每个核心构建独立 ringbuf 管道。 2. 实现多核 CPU 环境的适配。
目标 5: 负载监控与请求均衡	进行中 20%	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。

**目标 1** 是所有后续目标的基础，是为了实现相对底层的 eFuse 核心功能的必要修改，修改内容也需要随着后续目标的实现而不断完善和扩展。

**目标 2** 已经有部分相关开源项目进行了类似的设计和实现，eFuse 在此基础上进行了一定扩展和优化，希望实现尽可能覆盖绝大部分的 FUSE 元数据请求，提高统一性和全面性。

**目标 3** 是 eFuse 初赛阶段前半段的主要工作部分，区别与 ExtFUSE 等其他类似开源项目，尝试对更复杂的 FUSE I/O 请求进行优化，同时通过两条路径的设计，使系统能够在不同负载情况下都能维持较高的性能。

**目标 4** 主要针对 FUSE 的请求拥塞现象进行优化, 尝试修改内核驱动中 FUSE 的请求处理逻辑和存储结构, 使其能够支持多核 CPU 环境下的高并发请求处理, 提升整体性能。

**目标 5** 仍在进行中, 构想是通过 eBPF 程序动态分析 FUSE 请求的负载情况, 并根据当前的负载情况和请求队列状态, 调整调度策略和请求处理方式, 以实现更高效的请求处理。

综合来看, eFuse 在初赛阶段已经实现了绝大部分预期功能, 在兼容并保留 FUSE 的可扩展性、便捷性等优势的情况下, 大通过 eBPF 技术大幅提升其性能, 并在多种负载场景下表现出色。后续工作将继续完善各个模块, 进一步优化和扩展 eFuse 的功能。

## 参考文献

- [1] Y. Zhong *et al.*, “XRP: In-Kernel Storage Functions with eBPF,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 375–393. [Online]. Available: <https://www.usenix.org/conference/osdi22/presentation/zhong>
- [2] A. Bijlani and U. Ramachandran, “Extension Framework for File Systems in User space,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, Jul. 2019, pp. 121–134. [Online]. Available: <https://www.usenix.org/conference/atc19/presentation/bijlani>
- [3] K.-J. Cho, J. Choi, H. Kwon, and J.-S. Kim, “RFUSE: Modernizing Userspace Filesystem Framework through Scalable Kernel-Userspace Communication,” in *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, Santa Clara, CA: USENIX Association, Feb. 2024, pp. 141–157. [Online]. Available: <https://www.usenix.org/conference/fast24/presentation/cho>
- [4] Q. Huai, W. Hsu, J. Lu, H. Liang, H. Xu, and W. Chen, “XFUSE: An Infrastructure for Running Filesystem Services in User Space,” in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, USENIX Association, Jul. 2021, pp. 863–875. [Online]. Available: <https://www.usenix.org/conference/atc21/presentation/hsu>
- [5] Z. Yang *et al.*, “ $\lambda$ -IO: A Unified IO Stack for Computational Storage,” in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, Santa Clara, CA: USENIX Association, Feb. 2023, pp. 347–362. [Online]. Available: <https://www.usenix.org/conference/fast23/presentation/yang-zhe>