西 安 邮 电 大 学

bt_fuse 开发设计文档

项目成员：

姓名	年级	E-mail
尚凡	研一	2444576154@qq.com
杨传江	研一	2024056367@qq.com
谢佳月	大二	mufengyaa@gmail.com

项目导师：郑昱笙

指导老师：陈莉君

2024 年 5 月

目录

1 概述

1.1 项目背景及意义

FUSE（Filesystem in Userspace）文件系统允许用户在用户空间创建文件系统，这种设计提供了极大的灵活性。然而，由于频繁的用户态和内核态切换，FUSE 文件系统也引入了显著的性能开销。

本项目旨在分析并提出对现有 FUSE (Filesystem in Userspace) 文件系统的性能优化方案。通过技术改进，减少这些开销，提高文件系统的效率和响应速度，特别是在高频 I/O 操作场景中。

1.2 项目目标

我们希望在 bpftime 工具的帮助下，通过精确的性能监测和分析，尽量减少 FUSE 文件系统在用户态和内核态之间的不必要切换。频繁的态切换是 FUSE 系统性能瓶颈的主要原因之一，通过减少这些开销，我们期望显著提升文件系统的效率和响应速度。

在初赛期间，我们期望完成以下目标的一部分，包括初步的态切换分析、设计并实现优化后的 inode 管理模块，以及进行初步的性能测试和评估。这将为后续的优化工作奠定坚实的基础，并确保我们能够在较短时间内看到优化效果。通过这些努力，我们相信能够为 FUSE 文件系统带来显著的性能提升，使其在高频 I/O 操作场景中表现更加出色。

目标	完成情况	目标说明
bpftime 实现截断 sys_enter_openat 系统调用	完成 90%	通过进程 pid 和 open 的文件描述符，获取到 open 的 pathname
fuse 文件系统实现	完成 50%	实现 create、umtime、getattr、mknod 这几个

		回调函数
inode 管理和优化	完成 40%	1、实现 inode 的增、查、改 2、fuse 回调函数可以成功使用 inode 管理的功能
inode 缓存	未完成	1、实现 inode 缓存，优化缓存算法 2、动态调整缓存大小
inode 并发控制	未完成	1、使用读写锁或其他方式实现并发控制
性能检测与调优，并且完成可视化。	未完成	1、开发监控工具来跟踪 inode 缓存的性能 2、进行性能测试，以识别当前实现的限制，并基于测试结果调整策略

表 1：项目完成情况

1.3 项目开发历程

- **需求分析**
  - ✧ 确定选题，进行题目分析。
  - ✧ 确定优化目标和性能瓶颈。
- **技术调研**
  - ✧ 搭建开发环境。
  - ✧ 研究 FUSE 文件系统和 bpftime 工具，分析优化方案的可行性。
- **方案设计**
  - ✧ 对实现内核 bypass 机制，进行构思，确定 bpftime+inode 管理功能这个

方案。

- ✧ 进行架构设计，对整个方案进行说明，出初稿。
- ✧ 制定减少态切换和优化 inode 管理的具体技术方案。
- ✧ 对方案问题进行集体讨论，提出问题，并进行解决。

- **编码实现**
  - ✧ 实现 bpftime 提取 pathname，并且验证路径，进行系统调用截断。
  - ✧ 实现 inode 管理。创建 inode，查找 inode 等功能。
  - ✧ 对 fuse 文件系统进行重新设计，并实现调用 inode 管理功能。
  - ✧ 在终端对 fuse 文件系统的用户注册函数进行测试。比如：touch，检测是否会触发我们自定义的 inode 管理功能。
- **文档编写**
  - ✧ 编写项目报告和技术文档。

1.4 项目团队成员分工

尚凡

- 团队队长
  - ✧ 作为整个团队的主心骨，负责项目的整体规划和协调，确保各个部分顺利进行。
  - ✧ 监督各个部分的工作进展，及时解决问题和调整计划，确保各个部分顺利进行。
- 框架设计
  - ✧ 设计项目的整体架构，确保系统设计合理、可扩展。
  - ✧ 通过架构图、流程图和文档，详细描述系统的各个组件及其相互关系，确保设计的每个细节都经过深思熟虑和验证。
- 技术引导
  - ✧ 引导团队成员理解和掌握项目技术，提供技术支持和培训。
  - ✧ 在项目开发过程中，随时为团队成员提供技术支持，解答他们在编

码和调试中遇到的问题。

- inode 管理
  - ✧ **设计：**负责 inode 管理模块的设计，包括数据结构和算法的选择。
  - ✧ **编写：**实现 inode 管理模块的代码，确保其功能完整和高效。
  - ✧ **测试：**编写测试用例，进行单元测试和集成测试，验证 inode 管理模块的正确性和性能。
  - ✧ **扩展：**根据项目需求，持续优化和扩展 inode 管理模块，提升其功能和性能。
- 报告编写
  - ✧ 提供部分报告资料，分工让组员完成报告编写

杨传江

- Fuse 代码
  - ✧ 负责根据项目的整体架构和设计文档，编写 FUSE 文件系统的核心代码，实现文件系统的主要功能。
  - ✧ 实现用户空间与内核空间的交互接口，确保数据在两者之间的传递高效可靠。
  - ✧ 根据需求增加新功能模块，优化现有代码，提高文件系统的性能和可扩展性。
- 框架设计
  - ✧ 参与项目的整体架构设计和实现构思
- 报告编写

  撰写本报告的后三部分
  - ✧ **实现方式：**包括技术选型、架构设计、关键模块的实现细节等。
  - ✧ **测试方式：**描述项目的测试方法和过程，详细说明测试环境的搭建、测试用例的设计、测试结果的分析与验证等。
  - ✧ **未来展望：**总结项目的当前成果，提出未来的优化方向和扩展计划，展望项目在实际应用中的前景和潜在的改进措施。

谢佳月

- eBPF 程序设计
  - ✧ 负责 eBPF 程序的设计，确定如何利用 eBPF 技术监控和优化文件系统操作。
- eBPF 编写和测试
  - ✧ 实现 eBPF 程序，确保其高效、稳定地运行在内核态。
  - ✧ 编写测试用例，验证 eBPF 程序的功能和正确性。
- 框架设计
  - ✧ 参与项目的整体架构设计和实现构思
- 文档编写
  - ✧ 技术文档：撰写 eBPF 程序的技术文档，详细描述设计思路、实现细节和使用方法。
- 报告编写

  撰写本报告的前三部分
  - ✧ **背景介绍：**包括为什么选择优化 FUSE 文件系统，当前存在的主要问题，以及这些问题对系统性能的影响
  - ✧ **项目目标：**详细阐述项目的具体目标，以及实现这些目标将对 FUSE 文件系统带来哪些具体的改进和优势。
  - ✧ **开发历程：**记录项目的开发过程，包括从需求分析到方案设计，再到编码实现和测试的各个阶段。
  - ✧ **技术介绍：**对项目中使用的关键技术进行简单介绍，让读者对项目的技术基础有一个全面的了解。
  - ✧ **系统整体框架：**详细介绍系统的整体架构设计，包括系统的各个模块及其相互关系。解释每个模块的功能和作用，以及模块之间的交互方式。通过架构图和流程图，直观展示系统的设计思路和实现方法，为后续部分的详细描述奠定基础。

1.5 项目开发历程

时间	已完成工作	下一步计划
开始~03-24	1、讨论赛题 2、确定一批候选题目	1、确定题目 2、联系导师 3、讨论赛题，欲制定比赛方案
03-25~03-31	1、确定赛题，联系导师 2、学习 fuse 文件系统 3、学习 bpftime 4、看 extfuse 的论文	1、跑通 fuse 文件系统 2、了解 bpftime 机制 3、整合所有资料，出初稿
04-01~04-07	1、完成初稿 2、和同学讨论框架	1、解决关于初稿提出的问题 2、继续探索这个框架的可实现性
04-08~04-14	1、解决初稿提出的问题 2、学习 fuse 文件系统接口	1、自己写一个简单的 fuse 文件系统,调用 fuse 的接口 2、学习 bpftime
04-15~04-21	1、搭建 bpftime 环境 2、跑简单的 fuse 文件系统	1、学习 inode 管理 2、学习文件系统
04-22~04-28	1、完成 inode 管理的初稿撰写 2、完成学习文件系统部分内容	1、完成 indoe 管理部分内容的初稿 2、理解文件系统里面对 inode 的相关介绍
04-29~05-05	1、完成 inode 管理结构体的设计 2、完成文件系统对	1、用 bpftime 把 pathname 提取出来 2、对 inode 和 fuse 文件

	inode 的理解	系统的交互进行构建。
05-06~05-12	1、完成提取 pathname 的代码设计 2、完成 inode 和 fuse 文件系统的交互	1、实现 bpftime 到 fuse 文件系统回调函数的跳转 2、撰写文档
05-13~05-22	1、实现 bpftime 与 fuse 文件系统的交互 2、实现 inode 的简单设计	1、完成比赛文档的撰写

表 2：项目开发历程

2 现有工具了解

2.1 FUSE 技术

FUSE 允许在用户空间创建文件系统，提供了开发简便、灵活性高和安全性强的优势。用户可以编写普通的用户态程序来实现文件系统功能，而无需修改内核代码。

FUSE 是一个用户空间文件系统的框架，包括以下组件:

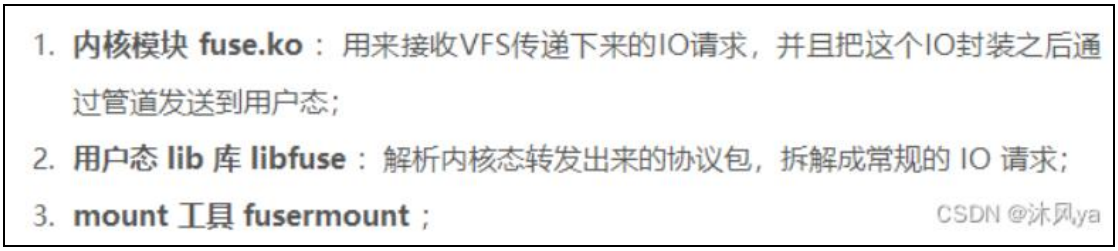


图 1：组件描述图

这三个组件帮助我们在用户态实现文件系统，并且让 io 可以在内核态/用户态文件系统之间自由穿梭。

调用流程图：

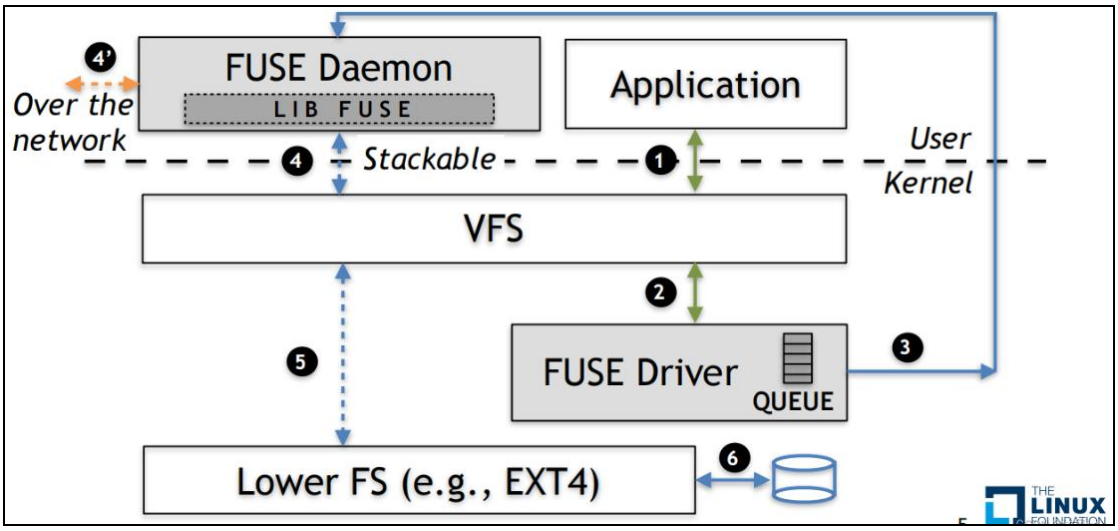


图 2：调用流程图

简化版流程图：

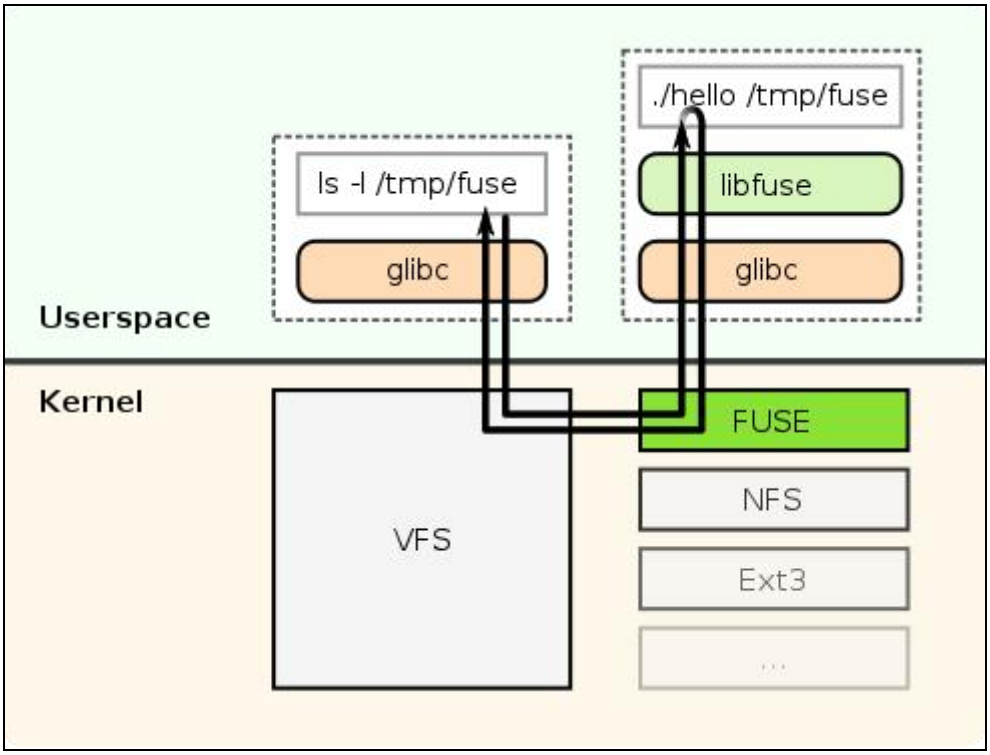


图 3：简化版流程图

然而，FUSE 的设计也带来了性能开销，尤其是在频繁的用户态和内核态切换时，这种开销尤为显著。这是因为每次文件操作都需要在用户态和内核态之间进行多次切换，导致延迟增加和性能下降。

本地文件系统和 FUSE 文件系统的 IO 性能对比图：

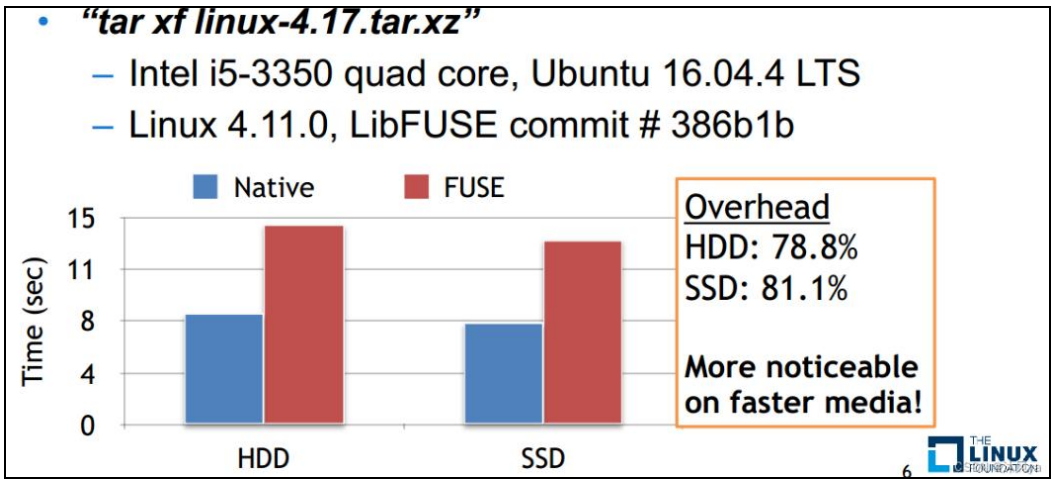


图 4：IO 对比图

2.2 ebpf 技术

eBPF（extended Berkeley Packet Filter）最初用于网络数据包过滤，但近年来其功能已扩展到可以在内核态执行任意类型的安全、高效的代码。

当今的 Linux 内核正在向一个新的内核模型演化：用户定义的应用程序可以在内核态和用户态同时执行。通过 eBPF，用户可以编写并在内核中动态加载运行自定义的程序，而无需修改/重新编译内核源代码，从而实现了一种灵活而安全的内核扩展方式。

它具有以下优点：

- 高性能：eBPF 程序可以在内核态直接执行，减少用户态和内核态之间的切换。
- 灵活性：支持动态加载和卸载，适应多种应用场景。
- 安全性：通过验证和限制机制，确保 eBPF 程序在内核中运行的安全性。

eBPF 技术被广泛应用于性能监控、网络安全和系统跟踪等领域。

eBPF 程序通用数据流程图：

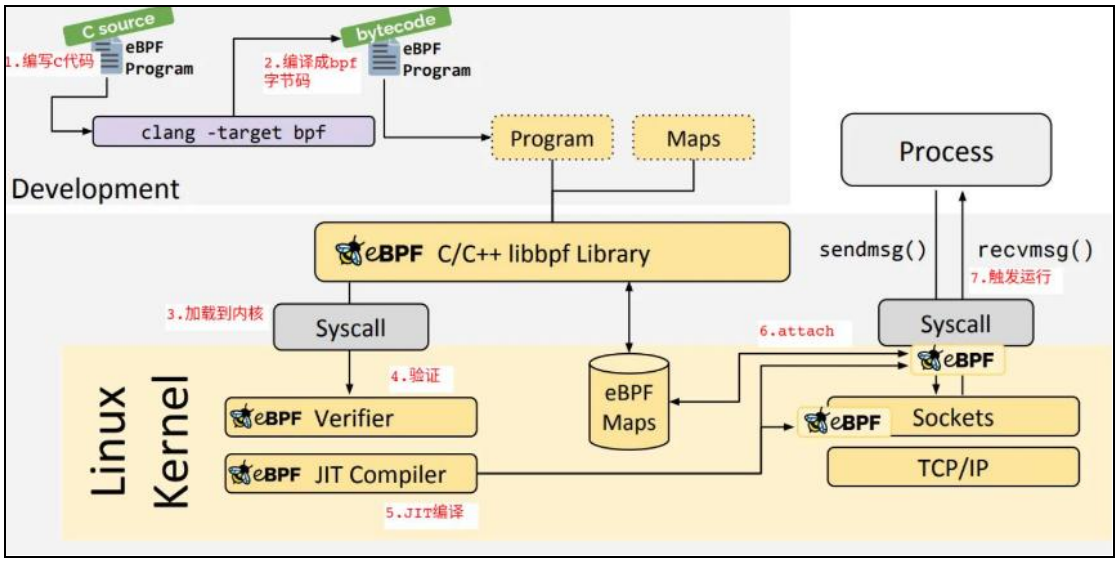


图 5：通用流程图

2.3 ebpf 技术结合 FUSE

结合 eBPF 技术和 FUSE 文件系统，可以在不改变内核代码的情况下，大大提升 FUSE 文件系统的性能和响应速度。

- **监控文件操作：**通过 eBPF 程序监控文件操作，收集性能数据，发现和定位性能瓶颈。
- **优化文件操作：**利用 eBPF 在内核态执行高效的文件操作，减少用户态和内核态之间的切换次数。
- **增强安全性：**通过 eBPF 实现文件操作的过滤和控制，增强文件系统的安全性。

这种结合方案不仅可以提升 FUSE 文件系统的性能，还可以增强其安全性和灵活性，适用于多种高性能和高安全性需求的应用场景。

3 系统整体架构设计

3.1 分析及指定方案

通过分析赛题，我们制订了一个切实可行的工作路线，以优化 FUSE 文件系统的性能。具体方案如下：

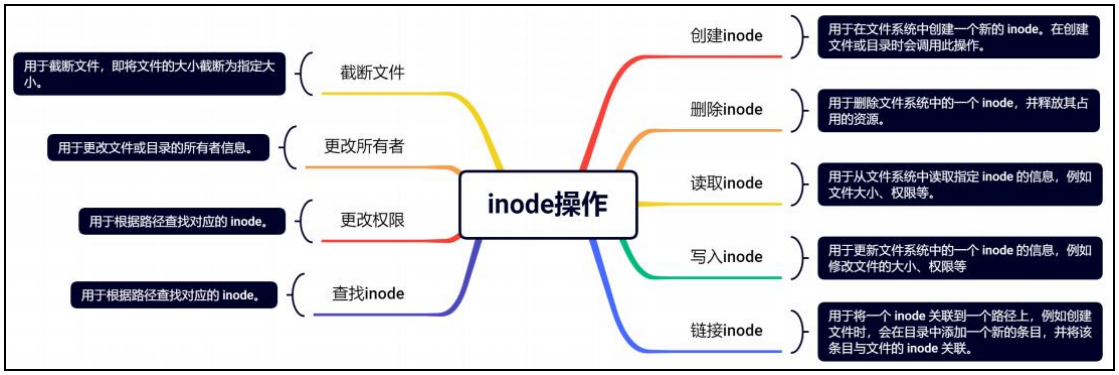
- **利用 bpftime 工具进行系统调用拦截**
  - ✧ 我们通过 bpftime 工具对 FUSE 的系统调用进行拦截，对于符合条件的系统调用引导调用我们自己开发的模块，不再进行系统调用，以此大大减少用户态与内核态的切换次数，提高反应速度。
- **重写 inode 管理模块**
  - ✧ 我们将 inode 管理逻辑单独封装成一个模块。这种设计使得 inode 的管理更加方便，并且显著提高了其操作效率。
  - ✧ 通过重写 inode 管理模块，我们能够优化文件系统的内部数据结构和访问效率。
  - ✧ 

图 6：inode 管理功能

具体的优化流程如下：

- **利用 bpftime 工具来优化 FUSE 的执行过程**
  - ✧ 当 FUSE 进行系统调用时，bpftime 进行拦截，以减少系统调用的次数。
  - ✧ 通过预先获得的 pathname 去判断是否是 fuse 下的目录，如果是则截断系统调用，改走我们自己实现的 inode 管理模块，实现我们自己想要的功能 ； 否则的话继续进入 vfs 模块进行系统调用。
  - ✧ 即利用 Linux 的 inode 管理机制，通过路径名直接获取 inode，减少路径解析的时间复杂度。

如下所示为我们实现的流程图：

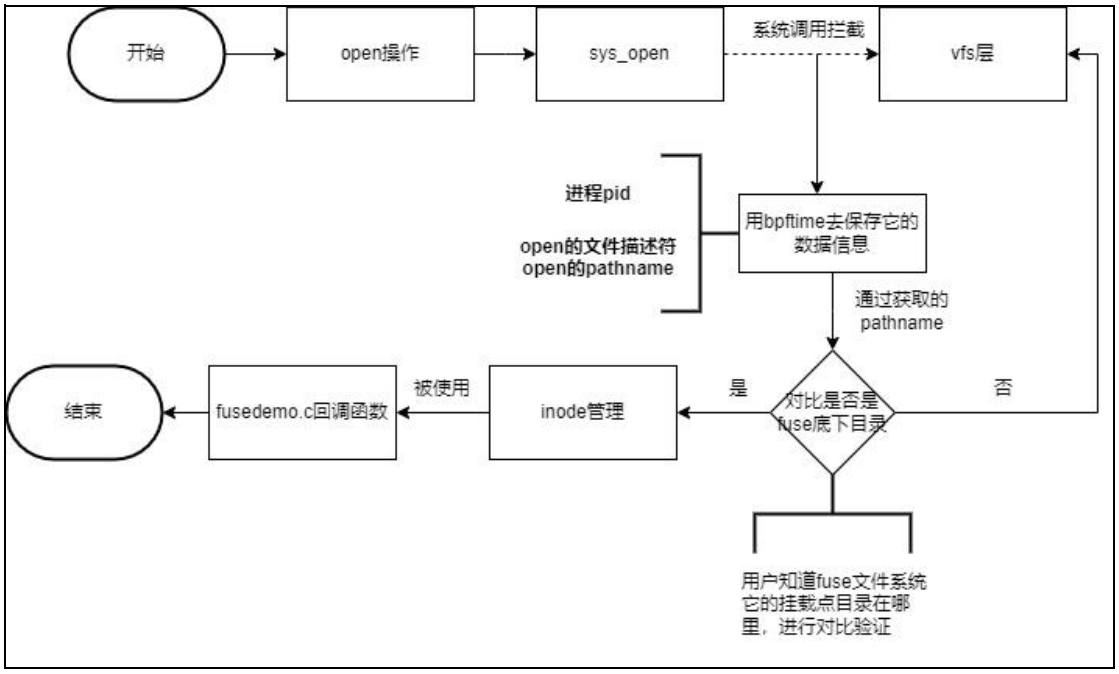


图 7：fuse 实现流程图

通过这些优化，我们借助 bpftime 工具截断系统调用，将大部分功能全部集成在用户态，改变了原有 FUSE 的执行流。如下所示为我们改变 FUSE 执行流后的框架图：

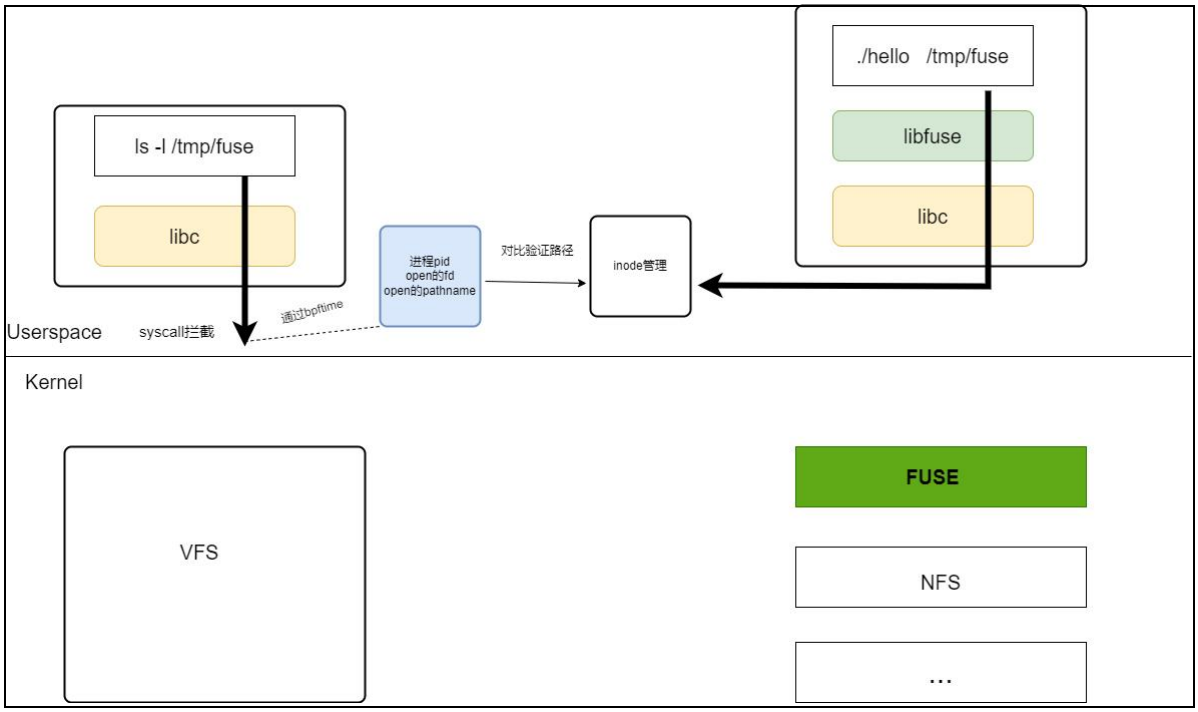


图 8：fuse 框架图

为了评估优化后的框架性能，我们使用精准的时间计数器来检测优化的效果。如下是优化性能检测的流程图：

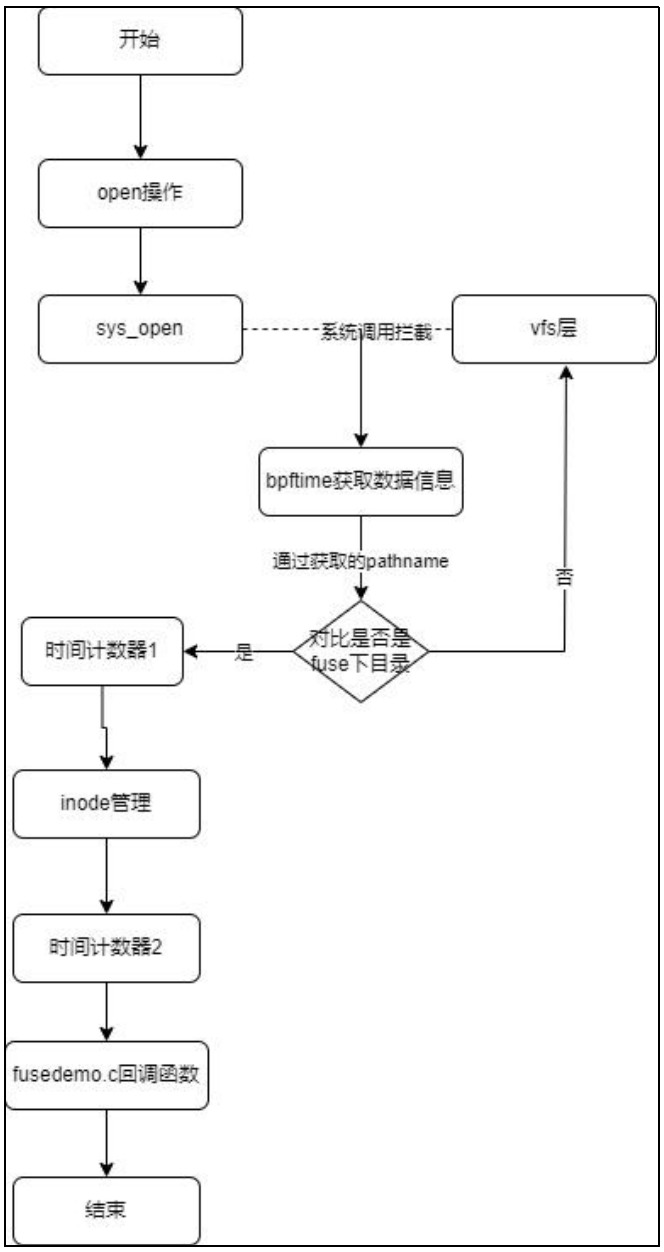


图 9：检测流程图

3.2 项目结构介绍

- **核心模块：**
  - ✧ **FUSE 文件系统代码：**实现 FUSE 文件系统的主要功能模块，包括文件操作、目录管理等。
  - ✧ **inode 管理模块：**重新设计和实现 inode 管理逻辑，优化 inode 的存储和访问方式，提高文件系统操作的效率。

- **拦截与优化模块：**
  - ✧ **bpftime 工具集成：**使用 bpftime 工具拦截系统调用，并根据预设条件引导系统调用到自定义模块。
  - ✧ **路径名判断逻辑：**通过路径名判断是否属于 FUSE 目录，决定是否截断系统调用。
- **测试与验证模块：**
  - ✧ **性能测试工具：**利用时间计数器和其他性能测试工具，检测优化后的系统性能。
  - ✧ **测试用例：**编写一系列测试用例，验证优化措施的有效性和稳定性。

通过以上模块的协同工作，我们期望能够显著提升 FUSE 文件系统的性能，达到优化的目标。

4 项目实现方式

4.1 bpftime 获取 pathname

将路径名（pathname）映射到 inode（索引节点）是文件系统的核心功能之一。我们实现这一过程的方式，是通过 bpftime 实现用户态 syscall 拦截，得到进程 pid，open 的文件描述符（fd），open 的 pathname。

在经过 pathname 对比验证是否是我们在 fuse 文件系统挂载下的目录,用 bpf 系统调用，给系统调用的返回值赋值为-1，达到截断系统调用的目的。从而将执行流切换到我们的 inode 管理部分，通过遍历，得到有 pathname 属性的 inode 指针，这样其他 Fuse 文件系统的回调函数使用的就是目前有进程响应的 inode。

如下图所示为提取到的 pathname 的数据信息：

图 10：提取的 pathname 数据信息

4.2 inode 管理

将 inode 的整体数据结构进行重新设计与优化，令其拥有更加高效的运行结构。将 inode 代码从原 FUSE 文件管理系统代码中全部抽离进行重构，将 inode 代码单独管理，实现单一职责原则，使得 FUSE 文件系统的整体代码耦合度更低，结构更加的合理。实现对 inode 的九大操作：创建 inode、删除 inode、读取 inode、写入 inode、链接 inode、截断文件、更改所有者、更改权限与查找 inode。
我们将根据 Fuse 每个回调函数的需求，设计 inode 功能。分为四部分：

第一部分：Fuse 是一个用户空间文件框架，通过回调函数来实现文件系统的各种操作。要将自定义的 inode 管理部分整合到 Fuse 文件系统中，通常使用 Fuse 提供的库函数来创建 Fuse 文件系统对象。这个对象将包含你的自定义回调函数以及其他必要的信息。根据我们自定义的 inode 管理部分的功能，实现对应的 Fuse 回调函数。例如，如果 inode 管理部分包含创建和删除 inode 的功能，我们需要实现 mkdir、rmdir、unlink、create 等回调函数。因为在 Fuse 文件系统中，一个命令的触发，一般会触发多个回调函数。这是在考虑测试文件系统的时候，一个命令会不会被触发，是牵扯到多个回调函数的。

第二部分：在创建 Fuse 文件系统对象时，将我们实现的回调函数注册到相应的回调函数指针中。这样当 Fuse 文件系统接收到对应的操作时，就会调用对应的回调函数来处理。

第三部分：编译和加载文件系统，将代码编译成二进制可执行文件，并使用 Fuse 提供的工具加载文件系统到系统中。一般来说，我们需要提供一个挂载点（mount popint）来挂载文件系统。

第四部分：运行一些命令，测试文件系统，确保它能够正确地处理各种文件操作。因为 Fuse 被编译后是一个守护进程，调试的时候，需要使用写入日志文件的方式进行调试，查看输出数据信息，是否正确。

如下所示为，我们在文件系统中主要实现的几个回调函数调用的函数过程：

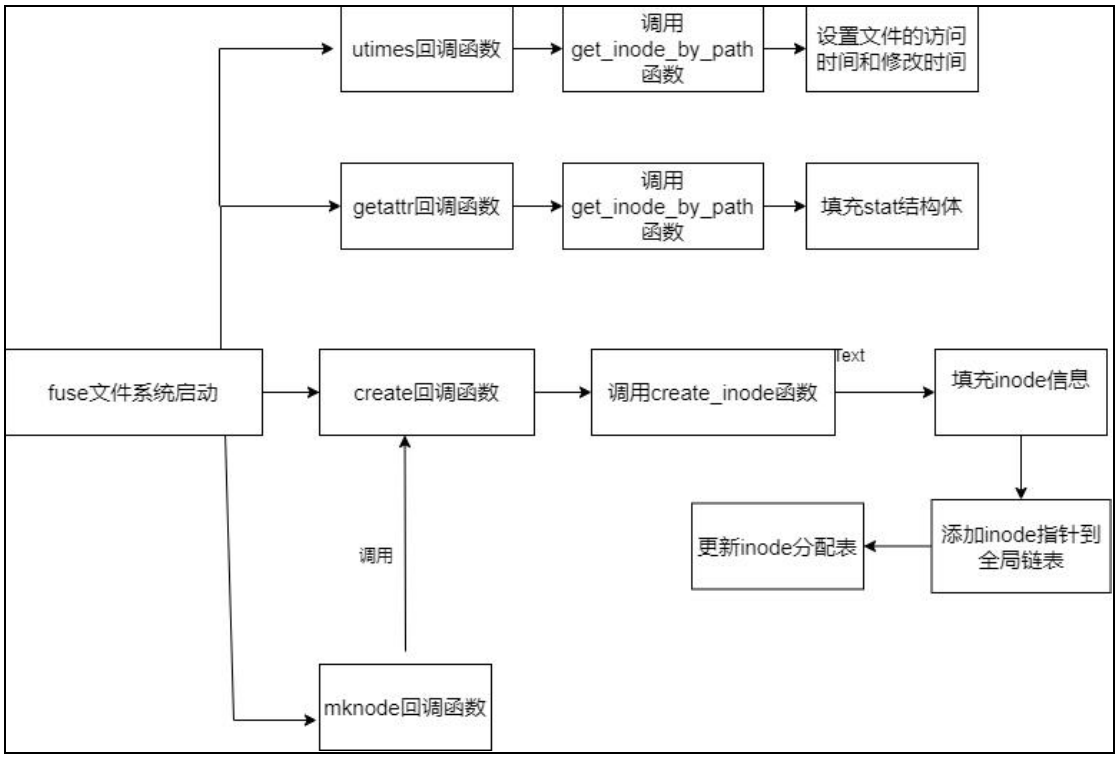


图 11：fuse 实现的回调函数调用情况

5 项目测试

5.1 项目测试方法介绍

为了全面评估本项目对于 FUSE 文件管理系统的优化效果，我们设计了一套测试框架，以此对我们的文件系统进行全方面的评估。

5.1.1 跨越内核性能检测

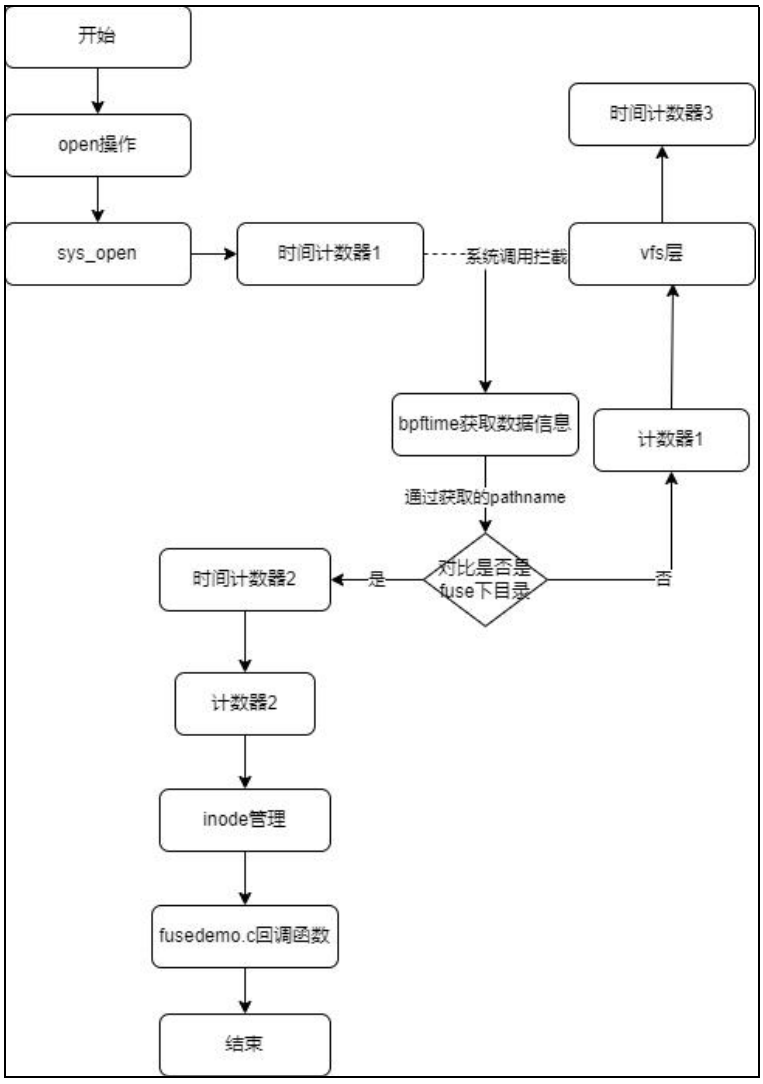


图 12：性能检测流程图

5.1.2 inode 优化性能测试

我们在 inode 管理模块前后各设置一个时间计数器，用来统计优化后的 inode 管理模块与未经优化的 inode 所需的时间。

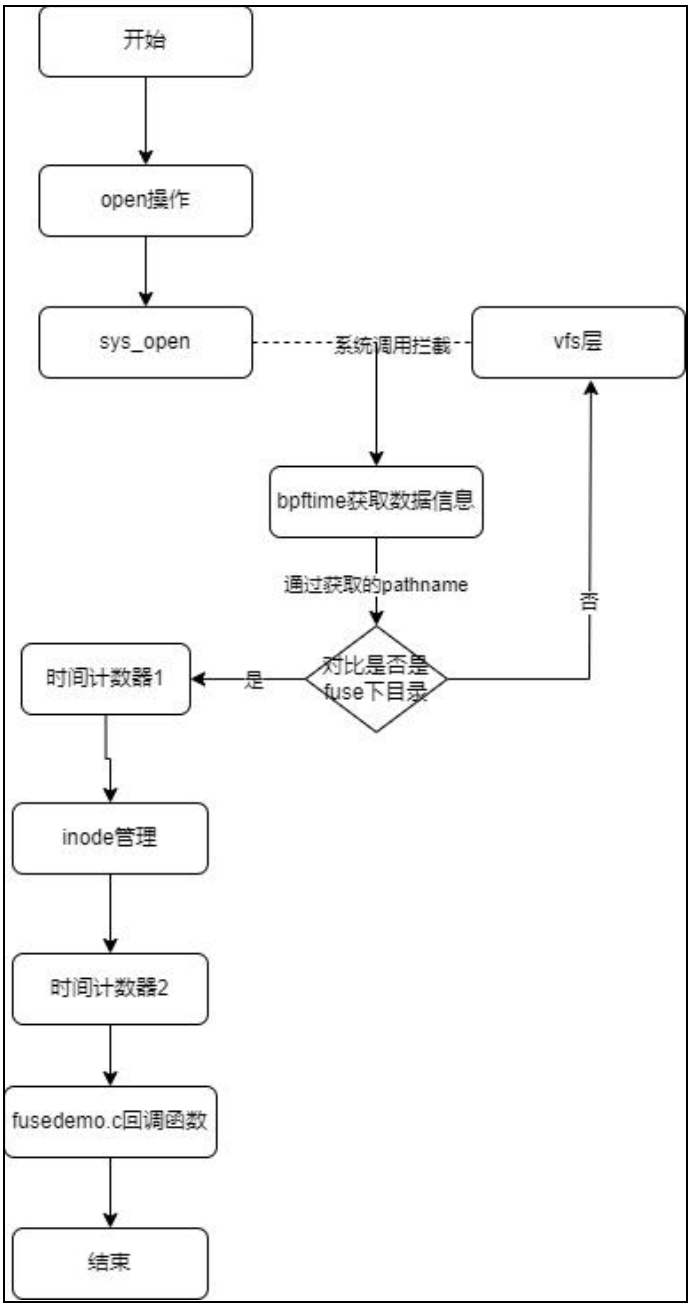


图 13：inode 性能优化检测流程图

6 总结与展望

6.1 项目总结

本报告旨在分析并提出对现有 FUSE（Filesystem in Userspace）文件系统的性能优化方案。FUSE 允许创建用户空间文件系统，这样的设计虽然提供了灵活性，但也引入了额外的开销，特别是在用户态和内核态之间的交互频繁时。本

研究通过技术改进，目标是减少这些开销，提高文件系统的效率和响应速度。

为此，我们借助 bpftime 工具，尽量减少 FUSE 文件系统在用户态和内核态的不必要切换，并且重写 inode 管理模块。以下是我们在项目初赛阶段所完成的工作：

确定选题并进行题目分析，讨论方案，搭建开发环境：在项目初期，我们明确了研究方向，分析了题目的可行性和挑战，最终确定了优化 FUSE 文件系统的目标，并成功搭建了开发所需的环境。

构思并确定内核 bypass 机制的方案，bpftime+inode 管理功能：通过深入讨论和研究，我们决定采用 bpftime 工具来实现内核 bypass 机制，并重写 inode 管理模块，以此提高系统性能。

进行架构设计，完成方案初稿：在详细讨论的基础上，我们设计了整个系统的架构，并撰写了初步设计方案，为后续开发工作提供了指导。

集体论讨方案问题并提出解决方案：在开发过程中，我们定期召开会议，针对出现的问题进行讨论并提出相应的解决方案，确保项目按计划推进。

实现 bpftime 提取 pathname，并验证路径和系统调用截断：我们已经基本完成了 bpftime 工具对路径名的提取和验证工作，并实现了系统调用的截断，这为减少用户态和内核态之间的切换奠定了基础。

实现 inode 管理，包括创建和查找 inode：我们成功地重写了 inode 管理模块，实现了 inode 的创建和查找功能，这一模块是整个优化方案的核心部分之一。

重新设计并实现 FUSE 文件系统的调用 inode 管理功能：在此基础上，我们对 FUSE 文件系统进行了重新设计，使其能够调用自定义的 inode 管理功能。

在终端测试 FUSE 文件系统的用户注册函数：通过对如 touch 等命令的测试，我们验证了自定义的 inode 管理功能是否被正确触发，确保了系统的可靠性。

尽管我们在初赛阶段完成了上述多个目标，但仍有两个重要工作未能完成：

对 inode 管理进行优化：这是整个项目的核心部分，但由于时间限制，我们未能完成此项优化工作。

对文件系统性能进行检测并完成可视化：我们计划通过性能检测和数据可视

化来评估优化效果，但这一工作也未能在初赛阶段完成。

总体而言，本次研究在 FUSE 文件系统的性能优化上取得了显著进展，虽然尚有部分工作未完成，但我们为后续的深入研究和优化打下了坚实的基础。通过进一步的努力和优化，相信我们能够实现目标，显著提升 FUSE 文件系统的性能和效率。

6.1 项目创新点

在本项目中，我们通过以下几个创新点提升了 FUSE 文件系统的性能：

内核 bypass 机制的引入：传统的 FUSE 文件系统在用户态和内核态之间频繁切换，导致性能瓶颈。我们引入了内核 bypass 机制，通过 bpftime 工具减少不必要的态切换，从而降低开销，提升响应速度。这一创新点显著减少了系统调用开销，使文件系统的效率得到了提升。

重写 inode 管理模块：为了进一步优化 FUSE 文件系统，我们重写了 inode 管理模块。通过自定义 inode 的创建和查找机制，我们不仅增强了系统的灵活性，还提高了 inode 操作的效率。这一改进使得文件系统在处理大量文件和目录时，能够更加高效地进行管理和访问。

集成 bpftime 与 inode 管理功能：将 bpftime 的路径名提取和验证功能与自定义 inode 管理功能相结合，形成了一个高效的文件系统架构。这一整合方案不仅简化了系统设计，还提供了更高效的路径解析和 inode 管理，提升了整体系统性能。

测试与验证机制：通过在终端对 FUSE 文件系统的用户注册函数进行测试（例如 touch 命令），验证了自定义 inode 管理功能的有效性和稳定性。这一创新性的测试方法确保了系统在实际应用中的可靠性和性能。

这些创新点不仅提升了 FUSE 文件系统的性能，还为未来的优化和扩展提供了宝贵的经验和参考。通过这些技术改进，我们展示了在用户态文件系统中进行性能优化的有效方法。

6.3 项目优化方向

在本项目的研究过程中，我们识别出了一些关键领域，可以进一步优化 FUSE

文件系统的性能。以下是未来优化的几个方向：

深入优化 inode 管理模块：尽管我们重写了 inode 管理模块，但尚未进行全面优化。未来可以通过引入更高效的数据结构和算法，进一步提升 inode 操作的速度和效率。此外，改进 inode 的缓存机制，减少重复查询和操作，也是一个重要方向。

完善内核 bypass 机制：当前的内核 bypass 机制已经减少了一些开销，但仍有优化空间。可以通过更加智能的系统调用拦截策略，进一步减少用户态和内核态之间的切换次数。研究动态分析和预测技术，提前识别和优化高频系统调用路径，也将有助于提升性能。

增强 bpftime 工具的功能：bpftime 工具在路径名提取和验证方面起到了重要作用，但其功能还可以进一步扩展。例如，增加对更多文件操作的支持，优化路径验证算法，以及改进与其他系统组件的集成效率。

性能监测和可视化：建立全面的性能监测和可视化系统，实时跟踪和分析 FUSE 文件系统的运行状态。这不仅有助于发现性能瓶颈和优化机会，还能提供直观的数据支持，指导进一步的优化工作。

并行化和多线程支持：针对多核处理器的优化，探索并行化和多线程技术，提高文件系统在高并发环境下的性能。通过优化锁机制和减少竞争，提高系统的扩展性和响应速度。

用户态和内核态通信优化：研究更高效的通信机制，减少数据传输的延迟和开销。例如，探索使用共享内存或直接内存访问（DMA）等技术，优化用户态和内核态之间的数据交换。

通过这些方向的优化，我们可以进一步提升 FUSE 文件系统的性能和效率，满足更高性能和更复杂应用场景的需求。