doi:10.11887/j.cn.202402022 http://journal.nudt.edu.cn

Linux 虚拟文件系统层的路径检索加速

邹彦良[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)

摘 要:为解决 Linux 内核传统路径检索日益凸显的开销问题,提出 Staged Lookup 以加速路径检索,通过动态缓存热目录来减少文件访问的时延。Staged Lookup 的核心在于缓存频繁使用的目录项,从而避免从根节点重复遍历路径。不同于从根结点开始的检索操作,Staged Lookup 扩展了搜索策略,允许从最近缓存的目录项向后或向前进行路径检索。在 Linux 内核版本 3.14 和 5.4 上部署 Staged Lookup 的原型,并开展实际系统测试。实验数据显示,相比于传统的路径检索方式,Staged Lookup 能实现高达 46.9% 的性能提升。

关键词:路径检索;虚拟文件系统;内核;目录缓存;性能
中图分类号:TP31 **文献标志码**:A **开放科学(资源服务)标识码(OSID)**:
文章编号:1001-2486(2024)02-215-09

听语音 与作者互动 聊科研

Accelerating path lookup in virtual file system of Linux

ZOU Yanliang[1,2,3], *YIN Shu*[1*]

(1. School of Information Science and 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: To solve the increasingly prominent cost of traditional path lookup strategy in the Linux kernel, Staged Lookup was proposed to accelerate path lookup by dynamically caching hot directories to reduce file access latency. The essence of Staged Lookup lay in caching frequently used directory entries, thus avoiding the repetitive traversal of paths from the root node. Unlike the retrieval operations that start from the root node, Staged Lookup expanded the search strategy to allow for forward or backward path lookup from the most recently cached directory entry. A prototype of Staged Lookup was deployed on Linux kernel versions 3.14 and 5.4 and subjected to real system tests. Experimental data indicates that Staged Lookup can achieve performance improvements of up to 46.9% compared to traditional path lookup methods.

Keywords: path lookup; virtual file system; kernel; directory cache; performance

基于 Linux 虚拟文件系统(virtual file system, VFS)的基本框架,文件访问通常可以分为处于 VFS 层的内存级操作,以及由 VFS 下层如文件系统、驱动等所管理的设备 I/O 操作。长期以来,由于常用的存储设备相比于内存具有更高数量级的访问时延,如机械硬盘、固态硬盘等,设备 I/O 通常占据文件访问的主要时间开销[1-2]。随着闪存、非易失内存等持久化存储设备的快速发展,其访存速度与内存的差距逐步拉近[3-5],Suzuki 等研究了 2000—2014 年之间非易失内存(non-volatile memory, NVM)技术相关的论文[6],NVM 的访问时延和带宽普遍被认为与动态随机存取存储器(dynamic random access memory, DRAM)处于同一数量级,大量针对 NVM 的研究工作由于无法获取现成的 NVM 设备,甚至会采用 DRAM 来进行模拟。

在 NVM 技术的影响下,文件访问过程中,存储设备级的数据 I/O 时延将会被急剧减小[7],相对应地,内存级的 VFS 操作所占据的时间占比将会被放大[8-9],尤其对于小文件或小 I/O 的文件操作,这种影响更加明显。如何降低传统系统的软件开销成为文件系统性能优化的新挑战。作为文件操作最先发起的操作之一,降低路径检索的时间开销成为文件系统适配高速持久化存储设备

收稿日期:2021-12-10
基金项目:中国博士后科学基金资助项目(2015M572708);上海科技大学启动基金资助项目(2017F020300001)
第一作者:邹彦良(1992—),男,广东佛山人,博士研究生,E-mail:zouyl@shanghaitech.edu.cn
***通信作者**:殷树(1984—),男,江苏泰兴人,研究员,博士,博士生导师,E-mail:yinshu@shanghaitech.edu.cn

的开端。

根据文件系统基准测试级 iBench 的统计，近 20% 的系统调用需要进行路径检索[10]。研究表明，基于路径检索的系统调用在执行文件相关的应用指令（如 find、tar、du 以及 git-diff 等）的时间开销中占主要部分。以 find 指令为例，超过 60% 的执行时间花在 open 操作上[11]；而对于 git-diff 指令，stat 操作占据主要的时间开销，无论 open 还是 stat，路径检索都是其主要操作。图 1 展示了利用 LMBench 基准测试工具对 4 种基本的文件相关系统调用（stat、open/close、read 以及 write）进行的测试，t 为时延。需要指出的是，测试针对的是系统调用时延，并不涉及实际的数据 I/O，此处 read、write 操作是往虚拟文件/dev/zero 中写 1 B 的数据。测试结果表明，路径查找操作在上述调用中均占据了主要的系统调用时间开销。可以认为，在不考虑数据 I/O 的前提下，路径查找在文件访问过程中占据着极其重要的时间开销。而在数据 I/O 时延快速减小的当下，路径检索将占据越来越重要的时间开销地位。

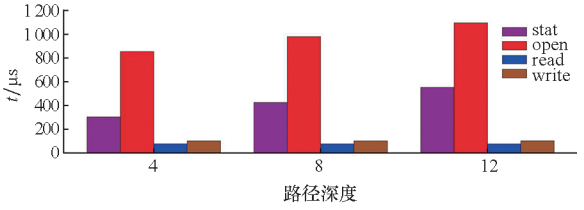


图 1 部分系统调用针对不同深度文件路径的时延

Fig. 1 Latency of some system calls with different depth of paths

除了高开销，路径检索在文件操作中出现的频率也很高。以智能手机的文件浏览器 File Explorer 和垃圾清理应用 NoxCleaner 为例，它们每秒能产生高达 10 000 条路径检索操作[12]。不仅是智能手机应用，大规模科学计算应用也会产生大量的路径检索。据国家超算无锡中心“神威·太湖之光”超级计算机的统计，每天有高达 52.4% 的操作需要进行路径检索[13-14]。这意味着“神威·太湖之光”超级计算机系统一天中有超过 1 300 万个路径检索请求，包含高达 8 900 万个路径分量的访问。进一步的分析发现，上述 8 900万个路径分量访问请求仅涉及 235 个不同的目录/文件，换言之，少数目录被大量频繁重复地访问。

如何提高路径访问效率，直接影响到了应用的响应时间，目前主要有两种类型的优化思路：第一种是在文件系统层减小路径分量检索的时延[15-16]。比如，TableFS 通过额外添加对象存储的方式，加速元数据以及小文件的访问操作[17]。在并行与分布式文件系统中，研究人员通过采用扁平式命名空间[18]、命名空间快照[19]以及动态管理[20]等优化元数据管理的方法，降低路径检索时延。第二种优化路径检索的思路是针对各级目录项采用缓存技术。Tsai 等提出一种基于全路径哈希的目录缓存机制，用哈希表来缓存目录项[11]，但维护哈希表的开销较大；Shen 等基于智能手机平台提出一种路径前缀缓存技术[12]，但该技术面临缓存命中率较低，且无法修改元数据等棘手问题。

在现有研究工作的基础上，本文发现路径检索的优化可以考虑另外两个因素：①相同应用访问的文件多位于同一个目录下[21]；②小文件的访问频次大多高于大文件[10,16,22]。基于上述发现，本文提出了名为 Staged Lookup 的分级缓存技术，以缩短文件检索的时延。Staged Lookup 通过缓存若干个被频繁访问的 dentry 目录项，采用多“起点”检索的方式，从而减少路径查找的时延。特别地，被缓存的频繁访问 dentry 结点被称为 pivot。在路径检索时，内核线程会在有限大小的 pivot 池中选取一个最优的 pivot 作为起点进行检索。

Staged Lookup 的主要挑战在于，如何选取最优的 pivot 以及如何管理 pivot 池。为此，Staged Lookup 采用了一个有序队列来组织 pivot 池并加速 pivot 的查找。除此，每个 pivot 都记录其祖先 dentry 的权限信息，Staged Lookup 可以快速进行权限验证。值得注意的是，当目录的元数据被修改时（例如重命名），Staged Lookup 需要对 pivot 池进行维护。但由于 pivot 数量有限，且维护操作可以采用异步的方式完成，实际产生的开销可以忽略。

本文主要的创新点包括：①提出了一种在 Linux 内核虚拟文件系统层的多层级路径检索优化策略 Staged Lookup，其利用缓存频繁访问的目录并以此替代根结点作为路径检索起点的方式，减少重复 dentry 的查找；②分别在 Linux v3.14 和 v5.4 两个内核版本上实现了 Staged Lookup 的原型系统，并利用真实服务器环境采集的路径检索负载开展测试；③利用一系列综合实验测试方法多角度评估 Staged Lookup 原型的有效性和优化程度。

1 相关技术

1.1 路径检索操作

Linux 的路径检索操作主要发生在 VFS 层，

而不是在具体底层文件系统上。当内核线程查找文件时,会切分路径的分量,即是一个个目录,然后逐个遍历并进行权限验证。为了避免频繁访问底层文件系统,Linux 内核在内存中维护了一个目录缓存,缓存中的每个单项都是一个 dentry 结构体,记录着对应目录的元数据信息。为了方便查找 dentry,内核维护了一个哈希表,把具有相同哈希值的 dentry 放在同一个哈希桶中。在查找某个目录项时,内核线程会计算它的哈希值,继而找到对应的哈希桶,逐个比较桶中 dentry 的父节点信息以及名字来确定目标 dentry。如果目录项不在目录缓存中,内核线程会向底层文件系统获取该目录的元数据,并建立对应的 dentry。

路径检索被频繁触发,而且涉及较多重复的目录查找,但 Linux 内核从 v3. x 到 v4. x 很少对路径检索的过程进行优化[11],图 2 进一步利用 LMBench 探究 v4. 0 到 v5. 6 版本内核的路径检索时延,结果显示,路径深度越深,时延越高,相同路径深度情况下,时延趋势基本保持平稳,直到 v4. 15 出现了阶跃,这是因为在 v4. 15 及以后的版本中,内核安装了内核页表隔离(kernel page table isolation, KPTI)补丁来应对两个重要漏洞:Meltdown[23]和 Spectre[24]。这个补丁为内核和用户态页表提供了隔离,防止用户态应用利用旁路漏洞来获取内核数据。但在带来安全性的同时,KPTI 的引入对进出内核造成了 4 到 5 倍的时延开销[25],这也影响到路径检索的时延。

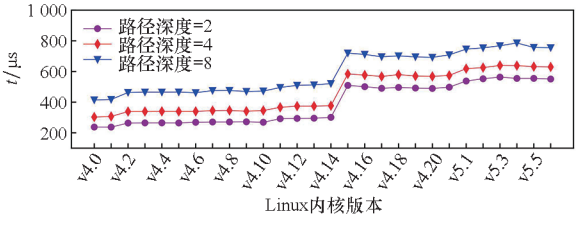


图 2　不同版本内核执行 stat 的时延

Fig. 2　Latency of stat under different versions of Linux kernel

1.2　全路径索引

全路径索引不像 Linux 原生的逐级查找,是一种利用全路径哈希来直接索引目标文件的方式。这种方式既可以设计在文件系统层[26-29],也可以在 VFS 层[11]。全路径索引需要考虑如何有效地处理目录名、权限等元数据的修改,因为必然会带来哈希键[30]、值的维护。

在 VFS 层进行路径检索时,内核线程会先根据全路径计算哈希值,然后在哈希表上快速找到对应的 dentry。但是一旦出现权限、名字被修改时,哈希表以及权限缓存中的所有相关条目都要被更新,这必然会对 rename、chmod 等操作的响应时间造成较大影响,时延开销会随着哈希表的大小而线性增长,达到几十倍甚至上百倍。

如何有效应对目录元数据的修改是全路径索引亟须解决的一个问题。文件系统层的全路径索引方式提出过一些解决办法,比如:BetrFS 用了两个 B^{ε} 树来加速元数据更新[31],后来提出 Tree Surgery 来做进一步优化[27]。但在 VFS 层,全路径索引仍然没有很好的处理办法。这驱使我们去重新思考在 VFS 层的路径检索优化,在减少检索时延的同时,更高效地处理元数据修改的问题。

2　Staged Lookup 的设计

根据国家超算无锡中心“神威 · 太湖之光”超级计算机上的监控系统 Beacon[22]的统计,检索的路径往往具有共同前缀,因为应用倾向于把文件放在同一个目录下。如果把最常被访问的前缀目录缓存起来,路径检索从这些目录开始而不是从根结点开始,可以避免很多冗余的目录遍历,从而加快路径检索的响应。本节主要介绍基于该思路 Staged Lookup 的设计方案。

Staged Lookup 的设计目标,是通过减少冗余 dentry 的访问,从而减少路径检索的时延。Staged Lookup 设计了一个名为 pivot 的数据结构,用于存储经常被访问的 dentry 信息。路径检索的时候会选择一个 pivot 作为起点而不是根结点("/")。图 3 展示了两个路径 Path1 和 Path2 的检索流程。当应用根据路径 Path1 检索文件 foo 时,Staged Lookup 选取 pivot c1 作为起点,经过 d1 找到 foo。在这个例子中,相比于内核中原生的流程(从根结点开始逐个遍历),Staged Lookup 把访问的目录数从原来的 5 个减少到了 2 个。

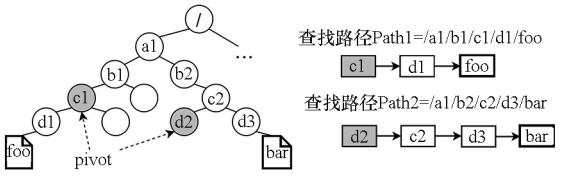


图 3　Staged Lookup 路径遍历举例

Fig. 3　Examples of Staged Lookup path traversal

不仅如此,Staged Lookup 还支持 pivot 的反向遍历,Staged Lookup 可以从某个 pivot 开始,向上遍历它的若干祖先结点。当应用通过 Path2 路径检索文件 bar 时,Staged Lookup 选取 pivot d2 作为起点,反向找到 c2,然后再向下遍历 d3 和 bar。

在这个例子中,Staged Lookup 把访问的目录数从原来的 5 个减少到了 3 个。这个回滚操作,可以通过在 pivot 中记录其祖先信息来进行加速(详见 2.2 小节)。

2.1 Staged Lookup 架构

图 4 展示了 Staged Lookup 的架构及工作流程,它包含两个模块(热度计数器、pivot 管理器)和两个数据集合(候选集、pivot 池)。

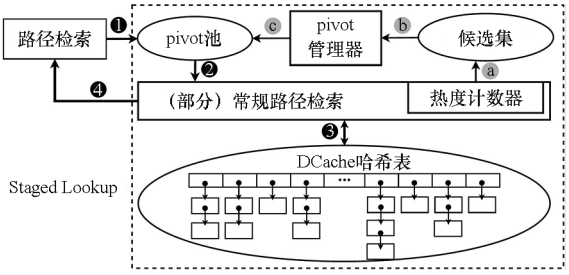


图 4　Staged Lookup 的架构以及工作流程

Fig. 4　Architecture and workflow of Staged Lookup

每个 dentry 中都内嵌一个热度值,热度计数器用来管理 dentry 的热度值,每个 dentry 在被访问的时候,热度值就会自增 1,高热度值代表 dentry 被频繁访问,2.3.1 小节会介绍如何更新维护热度值。而 pivot 管理器负责周期性地管理 pivot 池中的 pivot。

候选集则实时动态地维护着若干拥有高热度值的 dentry,作为下一周期的 pivot 候选。pivot 池管理着所有的 pivot,路径检索的时候,会在这里选取 pivot 作为起点。

Staged Lookup 在路径检索过程中可以分为两个阶段:阶段一为查找并选取一个最优的 pivot 作为路径检索起点,它一般是在所有 pivot 中离目标文件最近的;阶段二为逐个遍历目标路径中剩余的分量,如果需要回滚,则先回滚到某个祖先结点再往下遍历。

2.2 选取最优 pivot

Staged Lookup 最大的优点在于它能灵活地选取路径检索的起点,而不是固定地从根结点开始。如何快速地选取最优的 pivot(阶段一),决定了 Staged Lookup 的效率。pivot 池按照 pivot 的全路径字典序维护成一个有序队列,每个队列元素即是一个 pivot,同时记录当前 pivot 与前一个 pivot 所共享的前缀长度(overlap 值),通过这种机制可以减少大量的重复字符串比较。

图 5 展示了 Staged Lookup 选取最优 pivot 的例子,假设要根据路径/a1/b1/c2/d2/e3/f3/foo 检索文件 foo,Staged Lookup 先拿目标路径与 pivot1 的路径比较,c1 不匹配转而与 pivot2 的路径比较,因为 pivot1 与 pivot2 共享前两个路径分量,所以直接从 c2 开始比较。同理相继与 pivot3、pivot4 比较,发现 pivot4 与 pivot3 没有重合前缀,因此 pivot3(g3)就是要找的目标 pivot。

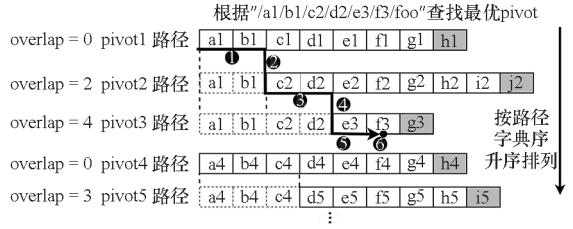


图 5　查找最优 pivot 过程

Fig. 5　Process of finding an optimal pivot

接着 Staged Lookup 进入阶段二,它会从 pivot3(g3)开始,往回滚一个结点到 f3,然后从 f3 再逐个遍历目标路径中剩余的分量。在这两个阶段中有一个优化是,在 pivot 中记录它的祖先结点信息,Staged Lookup 可以在阶段一查找到 pivot3 的过程中,直接获取 f3 的信息,省略掉选取 g3 后往回滚到 f3 的操作。这种优化下,Staged Lookup 选取的最优 pivot,一定是它或者它的祖先离目标文件最近的。

2.3 pivot 的管理

前面介绍了如何使用 pivot,本节将描述如何筛选和更新 pivot。

2.3.1 热度值的维护

pivot 是被频繁访问的目录结点,Staged Lookup 采用热度值这个属性去描述一个目录在最近时段内被访问的频次,为每个 dentry 都维护一个热度值。

当某个目录在一次路径检索中被访问时,其热度值就会自增 1。另外,每个 dentry 里记录了一个热度值的版本号,而内核也维护了一个周期性更新的全局版本号,当 dentry 中的版本号与全局版本号匹配时,热度值才有效,否则会认为它过时并重置热度值,然后更新 dentry 中的版本号。

比如,当内核访问目录 d1,Staged Lookup 过程会检查 d1 的版本号,如果与全局版本号一致,d1 的热度值加 1,否则 d1 的热度值会被重置为 1,再更新它的版本号。

全局版本号的更新,由 pivot 管理器在每次新周期到达时同步更新,周期时长是一个预设值,也是一个可以动态设置的值。

2.3.2 候选 pivot 的维护

在每个周期结束时,pivot 管理器会被唤醒并

进行 pivot 池更新。根据统计,一个本地文件系统会包含约 100 万个目录,显然遍历整个 DCache 找高热度的 dentry 是不可取的[12]。

Staged Lookup 引入了一个候选集来维护若干个被频繁访问的 dentry 作为候选 pivot,还设置了一个称为 least_popular_cand 的指针,指向候选集中“最少”被访问的 dentry,其示意图见图 6。在某个 dentry 被访问后,内核会把该 dentry 的热度值与 least_popular_cand 所指 dentry 的热度值进行比较,如果后者加上一个阈值之和小于前者,least_popular_cand 所指 dentry 将会被当前 dentry 取代。阈值的作用是防止短时间内出现几个目录项交替地换入换出。

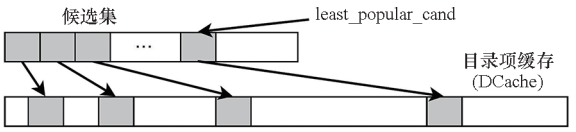


图 6　候选集示意图

Fig. 6　Schematic diagram of candidate set

举个例子,假设 least_popular_cand 指向一个名为 n1 的 dentry,阈值为 t_n,d1 是当前被访问的目录项。当内核访问完 d1 后,会拿“d1 的热度值”与“n1 的热度值 + t_n”比较,如果前者较大,则 n1 会被移除,d1 被插入候选集,同时 least_popular_cand 会指向 d1。

least_popular_cand 还会在候选集内部更新。每当候选集内某个 dentry 被访问并更新了热度值,它会被拿来跟 least_popular_cand 所指 dentry 的热度值比较,least_popular_cand 会指向热度值较小者。

2.4　处理数据一致性

Staged Lookup 的另一个重要问题就是元数据一致性,这个问题需要考虑两个场景:更新 pivot 池以及更新元数据。

2.4.1　更新 pivot 池

pivot 池的更新包括旧 pivot 的清除,以及生成新的 pivot 并插入。如果通过阻塞整个池来同步实现 pivot 的更新,会增加响应时延,Staged Lookup 设置了一个辅助池来协同完成 pivot 的更新。pivot 管理器在生成新 pivot 后会把它们插入到辅助池中,在这个过程中 pivot 池保持它的工作状态,插入完成后,pivot 管理器会交换两个池,辅助池变为新的 pivot 池,原 pivot 池会变为辅助池并进入垃圾回收状态。内核中内置的读 - 拷贝 - 更新[32](read copy update, RCU)机制被用来异步清理旧 pivot,当旧 pivot 没有被使用时,RCU 回调函数会进行旧 pivot 的释放。

2.4.2　更新元数据

元数据修改操作尤其是 rename 和 chmod 两种,会为 Staged Lookup 带来一致性问题,所有相关的 pivot 都要被处理。但相比于更新所有这些相关的 pivot,Staged Lookup 会直接删除清理,避免因阻塞更新带来的时间开销。因为 pivot 的数量非常有限,所以是常数级的复杂度。

以图 5 为例,假设用户要把/a1/b1 重命名为/a1/b2,所有相关的 pivot,包括 pivot1、pivot2 和 pivot3,都将会被标记为不可用,随后被异步地清理删除,但其他 pivot 不受影响。

而在周期更替期间,辅助池中新生成的 pivot 也可能会包含被修改的目录结点,Staged Lookup 会把整个辅助池都异步清理掉。

2.5　Staged Lookup 的开销

本节将分析 Staged Lookup 的各项主要开销。

2.5.1　检索的时间复杂度

路径检索时,内核原生方法会切分并解析每一个分量,分量会被扫描两次:一次用来计算哈希值,另一次用来验证 dentry 名字。相似地,全路径索引也有两次类似的路径扫描。

对于 Staged Lookup,参考图 5 的例子,有序队列以及 overlap 值的设置,阶段一只遍历一次路径,而阶段二与内核原生方法一样,因此 Staged Lookup 的基础时间复杂度要优于内核原生方法以及全路径索引。

2.5.2　移除 pivot 的时间开销

无论是 pivot 池周期性更新还是修改元数据,都需要清理 pivot,主要是通过标记无效 pivot 然后再清理。但考虑 pivot 正在被使用的情况时,阻塞等待会影响响应时间,持续监听又会浪费线程资源。幸运的是,内核自带的 RCU 机制提供了一种异步的方法来解决这个问题。在定义并关联一个回调函数后,内核每次产生 RCU 软中断后,后台系统线程都会检查并执行回调函数。把 pivot 的清理释放交给 RCU,当前进程可以无须阻塞等待,因此移除 pivot 的时间开销仅为查找相关 pivot 的时间。

2.5.3　内存占用

Staged Lookup 在设计中,为 dentry 结构体创建了 4 个新成员,在 32 位和 64 位系统中分别占用 18 B 和 30 B 的空间。但新成员不是简单地插入到 dentry 结构体中,因为会破坏 dentry 的 cacheline 对齐,而是占用了 d_iname 成员的空间,

它是一个字符数组,用来存储短文件名,但它本来就是个预留的空间,压缩 d_iname 并不会影响 dentry 的功能,因此 dentry 并没有因为新成员而增加空间开销。

在候选集中,每个成员都是一个 dentry,所以也没有额外的空间开销。主要的空间开销在于 pivot 池,在实验中,对于一个大小为 16 的 pivot 池来说,总共占据约 7 KB 的空间。

3 实验测试

本文将 Staged Lookup 与 Linux 内核原生的路径检索方式(以下称为 Original Lookup)以及全路径索引[20]的方式(以下称为 Directory Cache)进行比较。实验中基于 Linux 内核的 v3.14 和 v5.4 两个版本对 Staged Lookup 进行了部署,v3.14 是为了与 Directory Cache 进行比较,v5.4 是为了证明 Staged Lookup 在新版本内核的有效性。

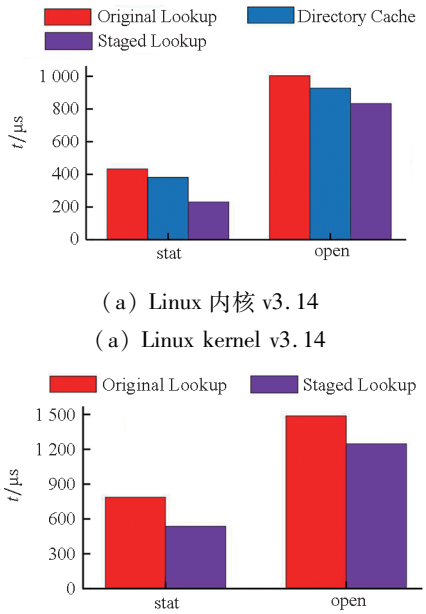
实验在一个单节点工作站上进行,该工作站配备了一个 4 核 3.3 GHz 的 Intel Xeon 处理器,8 GB RAM,以及一个 1 TB 7200 RPM 的硬盘。工作站采用 64 位 Ubuntu 14.04 Server 系统,挂载 ext4 文件系统。Staged Lookup 的更新周期设置为 2 s,pivot 池和候选集的大小为 16。

3.1 检索性能基准测试

LMBench v3.0 被选用作为 benchmark 来测试 stat 和 open 操作的时延。

图 7(a)展示了在 3.14 版本内核环境下,stat 或 open 一个深度为 8 的路径时,各对比项的性能表现。与 Original Lookup 相比,Staged Lookup 的 stat 和 open 操作分别减少了 46.9% 和 17% 的时间开销;与 Directory Cache 相比,分别减少了 39.4% 和 10.2% 的开销。

图 7(b)展示了 Original Lookup 与 Staged Lookup 在 5.6 版本内核的环境下的性能表现,尽管 KPTI 补丁对响应时延有影响,相比于 Original Lookup,Staged Lookup 依然有明显的优势,Directory Cache 没有参与对比,因为它无法兼容这个版本的内核,除非对它的开源代码进行大量的适配性修改。



(a) Linux 内核 v3.14

(a) Linux kernel v3.14

(b) Linux 内核 v5.4

(b) Linux kernel v5.4

图 7 路径检索时延 LMBench 测试

Fig. 7 Latency test of path lookup using LMBench

图 8 测试了 Staged Lookup 阶段一中 pivot 数量对性能的影响,展示了 Staged Lookup 在阶段一中比较了不同数量的 pivot 后,响应时延的结果。横坐标 N 表示在阶段一中比较了各个数量的 pivot 后,阶段二仅需再遍历 N 个剩余分量完成检索。比如横坐标为 6 且 pivot# = 16,表示阶段一在比较了所有 16 个 pivot 后,在阶段二中还需遍历剩余的 6 个路径分量。对比 Original Lookup,在最好的情况下(图 8 中横坐标为 0 且 pivot# = 1 的组)Staged Lookup 减少了 46.9% 的时间开销,然而在最坏情况(图 8 中横坐标为 6 且 pivot# = 16)下,Staged Lookup 增加了将近 10% 的时间开销,因为在阶段一中查找了较多无效的 pivot,同时还要遍历较多的剩余目录分量,这种情况一般出现在应用大量访问某些路径后,突然出现随机访问,这是 Staged Lookup 目前所不能很好应对的。

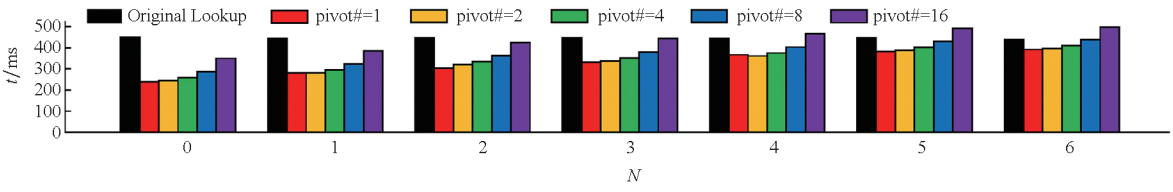


图 8 Staged Lookup 阶段一的 pivot 数量对性能的影响

Fig. 8 The impact of the number of pivot in staged one gains of Staged Lookup on performance

3.2　真实应用负载的检索性能

基准测试之后，Staged Lookup 会接受真实环境下的负载数据测试，进而进行性能评估，数据采集“神威·太湖之光”存储集群的客户端节点，利用“神威·太湖之光”的 I/O 监控系统 Beacon[22]，采集了 4 d 里真实应用对文件的访问行为，由于生产环境中不能随意修改服务器内核，实验是在测试机上建立了与存储集群上相同的目录树，让 Staged Lookup 重新执行相同的文件访问行为。

图 9 展示了 Original Lookup、Staged Lookup 和 Directory Cache 3 个对比项在测试中的时间开销，每个结果都是在相同的 6 次测试后取的平均值，每次测试都会清理 inode 缓存、目录项缓存以及页表缓存。以第二天的数据结果为例，相比于 Original Lookup 和 Directory Cache，Staged Lookup 分别节省了 29.6% 和 21.3% 的时间开销，Staged Lookup 优胜于 Original Lookup 显然是因为它减少了冗余的路径分量遍历；而 Directory Cache 采用的全路径索引方式虽然查找较快，但它的缓存机制仅对再次访问的文件才有收益。相反，基于 pivot 的职能设计，Staged Lookup 对第一次访问的文件同样有收益。根据对真实应用的观察，应用在运行过程中访问的文件，一般存放于相同的目录下，pivot 能发挥较大的作用，因此 Staged Lookup 比其他两个参照项，都具有较好的性能表现。

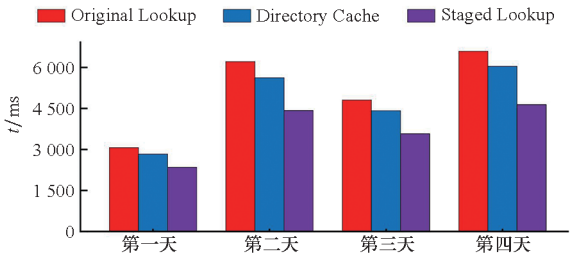
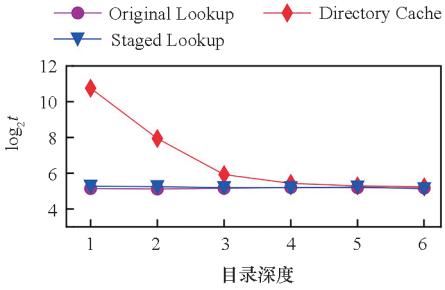


图 9　真实应用行为的路径检索时间开销

Fig. 9　The time cost of path lookup for real application behavior

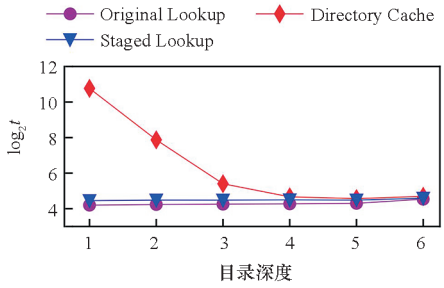
3.3　元数据修改的性能

除了性能优化外，元数据修改操作的性能也需要被评估。为此，实验中建立了一棵 6 层高的目录树，包含有超过 10 000 个结点，前 4 层每个目录下包含 10 个子目录，第 5 层的目录下有若干个文件。在每轮测试之前，所有的缓存都会被清理，然后用 stat 操作对目录树上所有的结点进行访问，以创造热启动的环境。针对不同深度的目录，实验测试了重命名 rename 以及修改权限 chmod 操作。图 10 为重命名和修改权限的性能比较，其每个结果数据都是 10 轮测试后取的平均值，t 的单位为 μs。



（a）rename 耗时

（a）rename time



（b）chmod 耗时

（b）chmod time

图 10　重命名和修改权限的性能比较

Fig. 10　Performance comparison of rename and chmod system call

相比于 Original Lookup，理论上 Staged Lookup 与 Directory Cache 因为引入了额外的机制，会在元数据修改中产生额外开销。图 10 结果表明，Directory Cache 方法引入了 50～100 倍的时间开销，这是因为修改目录元数据后，全路径索引对应的哈希表项都需要取遍历更新，哈希表越大，时间开销就越大，尤其在大访问量的服务器上，哈希表过大会产生难以接受的响应延迟。而 Staged Lookup 的时间开销几乎忽略不计，基本与 Original Lookup 持平。这是因为在修改目录元数据时，Staged Lookup 只需要处理有限数量的 pivot，而且只是标记受影响的 pivot，实际的清理工作由内核异步执行。

4　结论

本文提出了一种名为 Staged Lookup 的 VFS 路径检索优化方法，测试结果表明该方法能有效减少检索文件的时延。不同于内核原生的路径检索，Staged Lookup 避免从目录树根节点（"/"）开始检索，而是从提前缓存被频繁访问的

若干个 dentry 结点中选取一个作为路径检索起点,然后再逐个结点遍历最后找到目标文件。Staged Lookup 已经分别成功部署在 Linux v3.14 和 v5.4 两个内核版本上,相比内核原生的路径检索方式和全路径索引方式,Staged Lookup 相应分别减少了高达 46.9% 和 39.4% 的 stat 操作时间开销,open 操作也分别减少了 17% 和 10.2% 的时延。除此之外,Staged Lookup 在应对基于真实应用负载的测试中,展现出 29.6% 的性能优势。而在维护元数据一致性所产生开销中,如 rename、chmod 等操作,Staged Lookup 由于采用小规模缓存以及异步维护的方式,时间开销几乎与内核原生路径检索策略处于同一水平。

参考文献(References)

[1] WANG S C, LU Z Y, CAO Q, et al. BCW: buffer-controlled writes to HDDs for SSD-HDD hybrid storage server[C]// Proceedings of the 18th USENIX Conference on File and Storage Technologies, 2020.

[2] WANG S C, LU Z Y, CAO Q, et al. Exploration and exploitation for buffer-controlled HDD-writes for SSD-HDD hybrid storage server[J]. ACM Transactions on Storage, 18(1): 6.

[3] LIU S, KANNIWADI S, SCHWARZL M, et al. Side-channel attacks on optane persistent memory[C]// Proceedings of the 32nd USENIX Security Symposium, 2023.

[4] KIM M, KIM B S, LEE E, et al. A case study of a DRAM-NVM hybrid memory allocator for key-value stores[J]. IEEE Computer Architecture Letters, 2022, 21(2): 81 – 84.

[5] RAI S, TALAWAR B. Analysis of power-performance trade-offs in DRAM-NVM based hybrid main memory[C]// Proceedings of International Conference on Applied Computational Intelligence and Analytics, 2023.

[6] SUZUKI K, SWANSON S. A survey of trends in non-volatile memory technologies: 2000 – 2014[C]//Proceedings of the IEEE International Memory Workshop (IMW), 2015.

[7] ZHONG S, YE C H, HU G Z, et al. MadFS: per-file virtualization for userspace persistent memory filesystems[C]// Proceedings of the 21st USENIX Conference on File and Storage Technologies, 2023.

[8] LI R B, REN X, XU Z et al. ctFS: replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory[C]// Proceedings of the 20th USENIX Conference on File and Storage Technologies, 2022.

[9] CHEN Y M, LU Y Y, ZHU B H, et al. Scalable persistent memory file system with kernel-userspace collaboration[C]// Proceedings of the 19th USENIX Conference on File and Storage Technologies, 2021.

[10] HARTER T, DRAGGA C, VAUGHN M, et al. A file is not a file: understanding the I/O behavior of apple desktop applications[J]. ACM Transactions on Computer Systems, 30(3): 10.

[11] TSAI C C, ZHAN Y, REDDY J, et al. How to get more value from your file system directory cache[C]// Proceedings of the 25th Symposium on Operating Systems Principles, 2015.

[12] SHEN Z Y, HAN L, CHEN R H, et al. An efficient directory entry lookup cache with prefix-awareness for mobile devices[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2020, 39(12): 4575 – 4586.

[13] JI X, YANG B, ZHANG T Y, et al. Automatic, application-aware I/O forwarding resource allocation[C]//Proceedings of the 17th USENIX Conference on File and Storage Technologies, 2019.

[14] YANG B, JI X, MA X S, et al. End-to-end I/O monitoring on a leading supercomputer[C]//Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation, 2019.

[15] LI S Y, LU Y Y, SHU J W, et al. Locofs: a loosely-coupled metadata service for distributed file systems[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2017.

[16] ZHANG S L, ROY R, RUMANCIK L, et al. The composite-file file system: decoupling one-to-one mapping of files and metadata for better performance[J]. ACM Transactions on Storage, 16(1): 5.

[17] REN K, GIBSON G. TABLEFS: enhancing metadata efficiency in the local file system[C]//Proceedings of the 2013 USENIX Annual Technical Conference, 2013.

[18] TANG H J, BYNA S, DONG B, et al. SoMeta: scalable object-centric metadata management for high performance computing[C]//Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER), 2017.

[19] ZHENG Q, REN K, GIBSON G, et al. DeltaFS: exascale file systems scale better without dedicated servers[C]// Proceedings of the 10th Parallel Data Storage Workshop, 2015.

[20] 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, 2006.

[21] JIANG S, DING X N, XU Y H, et al. A prefetching scheme exploiting both data layout and access history on disk[J]. ACM Transactions on Storage, 9(3): 10.

[22] ROSELLI D, LORCH J R, ANDERSON T E. A comparison of file system workloads[C]//Proceedings of the Annual Conference on USENIX Annual Technical Conference, 2000.

[23] LIPP M, SCHWARZ M, GRUSS D, et al. Meltdown: reading kernel memory from user space[C]//Proceedings of the 27th USENIX Security Symposium, 2018.

[24] KOCHER P, HORN J, FOGH A, et al. Spectre attacks: exploiting speculative execution[C]//Proceedings of the IEEE Symposium on Security and Privacy, 2019.

[25] REN X, RODRIGUES K, CHEN L Y, et al. An analysis of

performance evolution of Linux′s core operations [C]// Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019.

[26] LENSING P H, CORTES T, BRINKMANN A. Direct lookup and hash-based metadata placement for local file systems[C]// Proceedings of the 6th International Systems and Storage Conference, 2013.

[27] ZHAN Y, CONWAY A, JIAO Y Z, et al. The full path to full-path indexing [C]//Proceedings of the 16th USENIX Conference on File and Storage Technologies, 2018.

[28] ZHANG R Y, LIU D, CHEN X Z, et al. ELOFS: an extensible low-overhead flash file system for resource-scarce embedded devices [J]. IEEE Transactions on Computers, 2022, 71(9): 2327 - 2340.

[29] ZHANG R Y, LIU D, CHEN X Z, et al. LOFFS: a low-overhead file system for large flash memory on embedded devices[C]//Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC), 2020.

[30] ZHENG X Q, MA J, LIU Y B, et al. BDCuckoo: an efficient cuckoo hash for block device [C]// Proceedings of IFIP International Conference on Network and Parallel Computing, 2022.

[31] JANNEN W, YUAN J, ZHAN Y, et al. BetrFS: a right-optimized write-optimized file system[C]//Proceedings of the 13th USENIX Conference on File and Storage Technologies, 2015.

[32] KOKOLOGIANNAKIS M, SAGONAS K. Stateless model checking of the Linux kernel′s hierarchical read-copy-update (tree RCU) [C]//Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, 2017.

(编辑: 王颖娟, 罗茹馨)