Stage Lookup: Accelerating Path Lookup using Directory Shortcuts

Yanliang Zou†‡§, Tongliang Deng†, Jian Zhang†, Chen Chen†, Shu Yin*†
Email: {zouyl, dengtl, zhangjian, chenchen, yinshu}@shanghaitech.edu.cn
†School of Information Science and Technology, ShanghaiTech University, China
‡Shanghai Institute of Microsystem and Information Technology, Chinese Academy of Sciences, China
§University of Chinese Academy of Sciences, China

Abstract—The lookup procedure in Linux costs a significant portion of file accessing time as the virtual file system (VFS) traverses the file path components one after another. The lookup procedure becomes more time consuming when applications frequently access files, especially those with small sizes. We propose Stage Lookup, which dynamically caches popular directories to speed up lookup procedures and further reduce file accessing latency. The core of Stage Lookup is to cache popular dentries as shortcuts, so that path walks do not bother to traverse directory trees from the root. Furthermore, Stage Lookup enriches backward path walks as it treats the dentry tree in a VFS as an undirected map. We implement a Stage Lookup prototype and integrate it into Linux Kernel v3.14. Our extensive performance evaluation studies show that Stage Lookup offers up to 46.9% performance gain compared to ordinary path lookup schemes. Furthermore, Stage Lookup shows smaller performance overheads in rename and chmod operations compared to the original method of the kernel.

Index Terms—path lookup, VFS, kernel, directory

I. INTRODUCTION

Path lookup is an essential procedure in operating systems because many system calls must operate on file paths at first before they can manipulate files. iBench system shows that about 10-20% of entire system calls perform a path lookup [1]. The directory cache work by Tsai et al. presents that the path-based syscalls weight up to 60% of execution time in many file management commands such as find, tar, du, and git-diff. These operations are mainly composed of open and stat operations (e.g., open weights more than 2/3 of the find execution time, and stat dominates the git-diff execution time) [2]. We use LMbench [3] to evaluate latency of four basic file-related syscalls including stat, open/close, read, and write (see Fig. 1). The results show that the path lookup operation occupy the main latency of file managements.

Not only does the path lookup consume a large fraction of execution time, but it is of frequent occurrence in file operations. Smartphone applications such as File Explorer and NoxCleaner produce more than 10,000 path lookups per seconds [4]. We analyze operational statistics from the TaihuLight supercomputer and find that up to 52.4% operations require path lookups in a day [5], [6]. Further evaluations of a one-day statistic show that 13 million path lookup requests generate more than 89 million dentry searches, while only 235 distinct dentries are involved. In other words, only 14.6% of dentry searches are effective and a great number of dentries are repeatedly visited.

Improving the efficiency of path lookup brings a fast response of applications [7]. Studies show efforts to improve the path lookup latency at the file system level [8], [9]. For example, Ren et al. designs TableFS to translate the local file system as an object-store [10]. In the parallel file system and distributed file system, researchers prefer to reduce latency by optimizing metadata management [11], [12].

Other studies try to optimize the path lookup efficiency at the dentry level. Tsai et al. purposes a full-path directory cache mechanism to reduce the lookup latency by storing recently accessed dentries that are hashed by full and canonicalized paths [2]. However, this mechanism introduces an extensive overhead to maintain the path components hash table when a directory needs to be modified via commands such as rename or chmod. Other than caching the recently accessed dentries, Han et al. proposes a mechanism to cache frequently accessed prefixes to improve path lookup performance on smartphones [4]. But applications with large directory trees may suffer from low prefix hit rate and poor path lookup efficiency due to the limited cache size. Directory modifications including renames and change modes are also challenging on mobile devices.

Besides, the path lookup optimization needs to take file access patterns into account for two reasons:(1), files that are associated with the same application tend to be accessed together as they are commonly stored under the same directory [13], [14]; and (2), small files are accessed more frequently [1].

We propose a method called Stage-Lookup to accelerate path lookup procedures and reduce file lookup latency in
VFS. Stage-Lookup is designed for a common scenario that files are stored in a directory tree with more than one level (i.e., files are not stored directly under the root directory). By caching frequently accessed nodes in a directory, Stage-Lookup can reduce the number of path components for path-based syscalls, hence reduce the lookup latency. We call these nodes pivots and maintain a Pivot Pool in the memory to cache them based on dentries’ accessing frequency. Notice that the path between two specific nodes in a directory tree is unique. Every time a path-based request arrives, the system kernel chooses the optimal pivot of the given path in Pivot Pool and raises conventional operations starting from the chosen pivot. If the pivot is not included in the path, Stage-Lookup must roll up levels to one of its ancestor directories, which is also a path component of the given path. This operation can be skipped if the ancestor dentry is marked (details are discussed in Section III-B).

The major challenge of this research is to efficiently find the best pivot and management of Pivot Pool. We apply an ascending-ordered list to maintain and search suitable pivots. Besides, Stage-Lookup performs string comparison only once for the given path instead of complex dentry searches. Moreover, Stage-Lookup can run permission checks quickly as every pivot stores its ancestors’ permission information, any lookup procedure will carry out a permission check on the pivot right away. Stage-Lookup also shows its advantages in handling directory modifications especially renames and change modes because these metadata operations only involve a limited number of pivots.

The main contributions of this paper include:

- We propose a path lookup optimization scheme in the VFS layer called Stage Lookup, which comprises techniques including:
  - Two-Stage path lookup: picking an optimal pivot as the start point, an then walking to the target from the pivot;
  - Pivot pool management: generating new pivots and updating pivots to ensure the two-stage path lookup procedure can always choose the optimal pivots;
  - Directory metadata modification: let pivots to inherit their ancestors’ directory metadata to restrain the modification procedure in the scope of pivots.
- We implement Stage Lookup on Linux kernel v3.14 and provide a comprehensive experimental study to fully evaluate the efficacy of Stage Lookup with microbenchmark and real-world workloads.

II. BACKGROUND

A. Original Lookup

When the Linux kernel tries to lookup a file with a given path, it parses the path into components and traverses them one by one. The lookup procedure accesses each component’s metadata for permission checks. And the kernel will access storage devices to get the required path components if they are not cached in the main memory.

Linux kernel builds a structure called dentry for each directory and caches them in the memory to avoid duplicated efforts of accessing file systems for certain directories [15]. The kernel maintains a hash table for dentries searches. Dentries with the same hashing value are aggregated into a bucket in the hash table. The lookup process parses the first component from the path and calculates its hash value to target the corresponding bucket. Then it scans dentries in the bucket and determines the target ones by checking their ancestors and comparing the name strings. If the subsequent component does not exist in the directory cache, the process will collect the component’s inode ID from the current directory and then access the inode to build a corresponding dentry.

B. Full-path Indexing

Full-path indexing is commonly studied to optimize the performance of indexing a file at the file system level [16], [17] and at the dentry level [2].

As a state-of-art study, Tsai’s propose a directory cache design that modifies the structure of dentries. It maintains a hash table to map a full-path to a specific dentry. When indexing a file, the kernel will calculate a hash value for the full path and look up its corresponding dentry in the hash table Tsai’s design also maintains a cache for the permission check. Every time a new dentry’s permission is checked, an entry containing the permission to this dentry will be inserted into the cache.

If the entry of a given path for indexing does not exist in the cache or out-of-date, the kernel has to switch back to the original path lookup procedure. In this case, the permission checks become noticeable overhead. This directory cache design works well when a directory has been accessed before (i.e., the directory is cached in the main memory) or users do not frequently change permissions and names of files. When a file changes its permission or name, not only the indexing efficiency is affected but it introduces more dentry traversals to update the relevant entries. A quick test indicates that Constant-latency operations (i.e., chmod and rename) in the original lookup may become linear in the size of Tsai’s directory cache, and take up to 100 times of latency to complete such an operation.

Besides, how to efficiently handle directory metadata modification is a common problem for the full-path indexing strategy. For example, BetrFS is local file system based on $B^r$-tree, a structure to realize full-path indexing [18]. BetrFS proposes a coupling optimization to particularly handle this problem [16]. BetrFS uses a mechanism called Tree Surgery to carve up nodes on the $B^r$-tree, in order to ensure consistency and efficiency when rename and chmod occur.

However, in the VFS layer, the performance of full-path indexing still suffers from handling metadata modification. This motivates us to propose a path lookup optimization scheme to optimize both the lookup latency and the directory metadata modifications including rename and chmod.
III. DESIGN OF STAGE LOOKUP

A. Overview

We can observe from the investigation on operational statistics of the TaihuLight supercomputer that different path walks commonly share some prefixes with each other. Path lookups will be much more efficient if we can cache the commonly accessed prefixes and start path walks from them instead of starting all the way back from the root ("/").

The design goal of Stage lookup is to reduce the path lookup latency by minimizing the number of dentries visited during the operation. Stage Lookup introduces a pivot structure, which stores the commonly accessed dentries so that most path lookups can start at an optimal pivot instead of the root (see Fig. 2 case “Path1”). In order to find the file foo, Stage Lookup picks the pivot c1 and starts the path walk from c1 to its descendant directory (d1), then reaches to the target file foo. Compared to the original path lookup that starts from the root directory (“/”), Stage Lookup only walks two path components. Furthermore, Stage Lookup supports backward pivot walks, meaning that path walks can roll up one level to the chosen pivot’s ancestor directory then walks down to another descendant directory (see Fig. 2 case “Path2”). Stage Lookup starts at the pivot d2 to look for the file bar, even though d2 is not a direct ancestor directory node to bar. Stage Lookup first rolls up to d2’s ancestor c2 and then walks down to d3 before it finds bar. In this case, Stage Lookup still walks fewer components than what the original lookup operation does.

![Fig. 2. A Stage Lookup path walk example. Grey nodes represent pivots.](image)

We then discuss the design details of Stage Lookup. Fig. 3 shows the architecture and workflow of Stage Lookup. Stage Lookup is comprised of two modules (Heat Counter and Pivot Manager) and two data structures (Candidate Set and Pivot Pool):

**Heat Counter** calculates the popularity of a directory node with a heat value. The heat value is self-incremented by one when the node is accessed. The higher the heat value of a node, the more frequently the node is accessed, and more than likely such node will be a pivot candidate. We explain the heat value updating strategy in Section III.C.1.

**Candidate Set** stores directory nodes with high heat values. These nodes are treated as pivot candidates for Pivot Manager that determines which candidates are pushed to Pivot Pool. Candidates will be evicted from Candidate Set if their heat values are surpassed by others, indicating the access frequency of other nodes is increasing.

**Pivot Pool** accommodates all the pivots for Stage Lookup. Each pivot contains descriptive information of a frequently accessed directory. Most pivots in Pivot Pool stay statically before Pivot Manager updates the pool. But pivots have to be updated immediately to retain consistency if directory metadata modification operations (i.e. rename and chmod) are performed.

**Pivot Manager** updates pivots in Pivot Pool by generating new pivots from Candidate Set and recycling old pivots in the next period. This manager is running in the background and is awakened periodically.

![Fig. 3. The architecture and workflow of Stage Lookup. The Stage Lookup workflow consists of four steps: (1) look for the optimal pivot in Pivot Pool with the given path (Stage One); (2) pick the optimal pivot as the path lookup starting point; (3) walk the path components in the DCache Hash table to search the target dentry; and (4) return the dentry and finish the path lookup. Stage Lookup manages pivots in three steps: (a) inserts frequently access directory nodes to Candidate Set as pivot candidates; (b) Pivot Manager determines pivots from Candidate Set; and (c) new pivots are pushed into Pivot Pool.](image)

Stage Lookup performs two stages for a path lookup operation:

**Stage One:** Find a pivot which is much closer to the target than the root (“/”), and pick the chosen pivot as the starting point of the path lookup. (Steps (1)&(2) in Fig. 3).

**Stage Two:** Walk the path components in the DCache Hash Table to search the target dentry; and (4) return the dentry and finish the path lookup operation. (Steps (3)&(4) in Fig. 3). The cache hash table is an intrinsic container to maintain dentries in the Linux kernel.

B. Finding An Optimal Pivots

Performing a path lookup from a directory node (a.k.a. pivot) that is much closer to the target than that from the root (“/”) is the major advantage of the Stage Lookup mechanism. How to quickly find the optimal pivot becomes the main challenge of Stage Lookup.

A simple way is to compare each pivot’s path with the given path and find the optimal one. Although this method makes it easy to manage pivots in Pivot Pool, the time complexity to find the best-fit pivots is too high to be acceptable. To solve this problem, we arrange the pivots as a list in ascending order of their paths. Each list entry stores the length of the shared prefix so as to reduce the repetitive string comparison of a path.

Fig. 4 presents a Stage Lookup example of finding an optimal pivot to target file foo at...
Fig. 4. The procedure of finding the best pivot in Pivot Pool. Each item represents a component of the pivot’s path and the grey color marks the pivot. Overlap gives the number of components that a pivot’s path shares with the previous one. Step 1-6 shows an example: a) the process compares the given path with Pivot1 (Step 1); b) it meets a distinct component and finds that Pivot2 shares the same prefix by checking its overlap (Step 2); c) Step 3-5 act similar; d) When the process finds that Pivot4 does not share the same prefix with the given path, it stops (Step 6) and Pivot3 is the best pivot.

Stage Lookup firstly compares the given path with pivot1 (Pivot1.path) then turns to compare with pivot2 (Pivot2.path) from its third component because the path does not match with the third component of pivot1. Stage Lookup then jumps to pivot3 (Pivot3.path) after it finds out the path does not match with pivot2’s fifth component and finally compares the manipulation in pivot3 when it finds f3. Pivot3 will be picked as the optimal pivot even though it caches the directory node g3 which is one level lower than the requested one (f3). Notice that Stage Lookup may not find a pivot that perfectly matches the given path, but it can find a pivot which is closer enough to the path so that the path lookup procedure can be much faster than that from the root (“/”).

In order to find foo, Stage Lookup then processes Stage Two, which performs the path lookup from pivot3 (g3). Stage Lookup first rolls one level up to pivot3’s ancestor node (f3) and succeeds in finding foo.

In order to minimize the overhead of a directory rolling upwards, we keep the dentity pointer to each component of the chosen pivot. In this case, the lookup procedure can directly access the valid dentity, neglecting the distance between the chosen pivot (g3) and the valid component (f3).

Fig. 5 demonstrates the data structure of a pivot and its components. Pivot Pool is a list of struct pivot in ascending order of their path. The header of Pivot Pool stores a pointer that points to the first pivot. The struct pivot is comprised of pivot’s path, overlap value, and an fixed-sized array of struct component. The overlap stores the number of shared path components between the current pivot with the pivot that sits above it in the Pivot Pool list. The struct component includes a struct dentity pointer, depth, and offset. The fixed-size array may waste space for short paths, but it can diminish data loading operations from memory to CPU cache. We keep a list pointer for struct pivot to extend the extra components to adapt the pivot structure to the path’s arbitrary length.

C. Maintaining Pivots

After explaining the way to find optimal pivots, we then describe how to pick and update pivots in this section.

1) Heat Value Updating Strategy: As aforementioned, frequently accessed directory nodes will be considered as pivots. Stage Lookup uses Heat Counter to set a heat value for dentries, representing the access frequency of lookup targets.

Heat Counter increases the target’s heat value by one when executing a path lookup. We do not increase the heat values of other involved path components because it will eventually make the top nodes of the directory tree hold the highest value. Besides, a pivot stores dentries of all the involved nodes so that Stage Lookup can traverse to any ancestor nodes of the pivot. We do not adopt dynamic weight to different directory depth as it will introduce path lookup overhead. Because a process has to scan the path to collect the weight information before it executes a path lookup.

However, the heat values can not be valid all the time. For example, if a directory lies idle for a long time, it does not deserve a pivot although it used to be required frequently.

Thus, we set a validity period for all the heat values, which is identified by a version number. We further set a global version number managed by Pivot Manager. A dentity’s heat value is valid only if its version number matches the unique global one; otherwise, it will be reset when new access occurs. The global version number is updated by Pivot Manager only when a new period comes. The length of the period is static in our study, which could be dynamic in our future work. But it can be adjusted by a system call with any other value.

2) Candidates for pivots: At the end of each period, Pivot Manager will be waked up to update pivots. However, it is inefficient to traverse the whole DCache to find out a number of dentries with outstanding heat values. According to the statistic of a study [17], about one million directories are contained in a local file system.

Thus, We introduce Candidate Set to gather a number of popular dentries as candidates of pivots. We set a pointer called least_popular_cand to mark the item whose heat value is “least” frequently updated in Candidate Set (see Fig6). During a lookup operation, the process compares the target’s heat value with the sum of a threshold and least_popular_cand’s heat value. If the former wins, the target’s dentity will replace what least_popular_cand
points to as a candidate and inherit the pointer. The threshold is to prevent some popular dentries from being in and out frequently in a short time.

![Diagram](image)

**Fig. 6.** Candidate set collects the most popular dentries in the current period. And *least_popular_cand* points to the item that may be replaced by a new member.

Besides the impact of a new member, the *least_popular_cand* can also be updated by candidates internally. Everytime a candidate’s heat value is updated, it will compare its value with the *least_popular_cand’s*, and the loser becomes the new *least_popular_cand*. Thus, *least_popular_cand* does not strictly mean the candidate with the least heat value in the set. For example, if a dentry is never accessed after being a candidate, it can prevent itself from being the *least_popular_cand* because no heat value updating means no comparison with the *least_popular_cand*. Maintaining *least_popular_cand* offers an effective rather than the best strategy to update the set, since traversing the whole set to find out the real least-accessed candidate is costly. We believe that every candidate in the set will be accessed frequently.

When a new period comes, Pivot Manager will generate new pivots with dentries serving by Candidate Set. For each dentry, it will traverse all its ancestors to get the whole path.

**D. Handling Metadata Modification**

Besides improving the performance of path lookup, Stage Lookup can efficiently handle the metadata modification of directory, such as rename and chmod.

Modifying directory’s metadata, such as chmod, rename, move and etc., is always a potential problem for full-path indexing methods. Because these methods use the whole path string as a key to attain hashing value, which means they have to depend on extra mechanisms to guarantee permission check and directory operations. It may introduce considerable overheads.

In Stage Lookup, modifying directory’s metadata only affects Stage One while Stage Two follows the conventional lookup procedure. Considering that a common component may be included in multiple pivots, we can directly remove and free those related pivots from Pivot Pool before executing the modification. Since the size of Pivot Pool is limited, we can say that our overheads to these problems are in constant time complexity.

**E. Concurrency**

Besides the modification of metadata, another important issue for Stage Lookup is the concurrency problem, which can be considered respectively in two situations: updating Pivot Pool and modifying metadata.

The first one occurs when updating Pivot Pool periodically. We have to consider concurrency among the working processes. However, it is not an efficient way to block the whole Pivot Pool to insert new items and free out-of-date items.

Thus, we set up another pool as a collaborator. The two pools act as a working pool and a waiting pool by turns. When a new period comes, Pivot Manager generates new pivots in the waiting pool, say Pool B. The working one, say Pool A, keeps unaffected until Pivot Manager finishes generating. Then the Pool A will be totally replaced by the fresh Pool B if no references fall on it. We utilize RCU (Read Copy Update) to achieve efficient monitoring of Pivot Pool’s references, which is a technique to handle the concurrency in the kernel. With RCU, new requirements for pivots will be guided to the Pool B if it is ready, while the current working processes can still stay at the Pool A. When all cores of the machine complete at least one context switch respectively, it can be confirmed that no processes are accessing the Pool A. Because in the kernel, the path lookup procedure is wrapped by rcu_read_lock() and rcu_read_unlock() which prevent the process from context switch before finishing. After the two pools achieve such an exchange, Pivot Manager will clean up the old Pool A.

Pivot Manager will upgrade the global version number after generating pivots, which means a new round comes for all the dentries to join Candidate Set. Then Pivot Manager will kick those overdue members out of Candidate Set after exchanging the two Pivot Pool.

For the second situation, modifying a directory’s metadata (rename/chmod) causes a consistency problem since a pivot is related to its two neighbors in the ascending-order list. Supposed Path $P_a$ describes the target being modified, and $p_v$ is the first pivot contains this target in the working pool (Pool A). We set a flag on $p_v$ to mark that $p_v$ and all the following pivots are temporarily invalid for new accesses. The other pivots lying before $p_v$ is unaffected. Then we remove those pivots related to $P_a$, and reanimate the rest pivots after that. However, Pivot Manager may be generating new pivots covering $P_a$ in the alternate pool, say Pool B. In order to reduce the latency of rename/chmod, we mark Pool B as totally invalid since it is not in used. So that the Pool B will be cleaned but no pool exchanging occurs in the coming period, which means that Pool A will be still valid in the next period.

**F. Cost Analysis**

Stage Lookup brings convenience to the path-lookup procedure. Here we analyze the costs of different aspects of Stage Lookup.

**Time Complexity of Finding Pivot:** When executing a path lookup, the original method will parse the string into components. Each component string will be scan twice: one is to calculate the hash value; the other is to verify the dentry’s
name. Similarly, full-path indexing takes two scans on the whole path for the same reasons.

Differently, stage lookup will not complete the dual scans. Let \( p = p_1+p_2 \) be a given path where \( p_1 \) represents the prefix skipped in Stage One, and \( p_2 \) be the rest path used in Stage Two. Overlap(see Section 3) helps Stage One to take only one scan to work out \( p_1 \). And Stage Two is similar to the original lookup procedure, which will traverse the rest path twice.

**Overheads of Removing Pivots:** We discussed how to maintain Pivot Pool when metadata modification occurs in Section III-E. It is to make partial pivots invalid and remove those related pivots. However, freeing a data structure takes time which may lead to an unacceptable latency for the whole operation. Fortunately, RCU mechanism serves an asynchronous way to solve this problem. By inserting the removal into a callback function which will be detected and executed by the RCU soft interrupt asynchronously, the current process can directly move on without waiting. Thus, metadata modification can endure little overheads.

**Space Overheads:** To achieve the candidates and the heat value management, we create four new members for `struct dentry`, which worth 30 bytes in a 64-bit system or 18 bytes in a 32-bit system. Instead of simply inserting new members into `struct dentry`, we occupy the partial space of `d_iname` which is a `char` array to keep `struct dentry` aligned on 64 byte cachelines. Reducing the space of `d_iname` will not affect the function of `struct dentry`. So that we introduce new members in `struct dentry` without breaking its alignment on 64 bytes cachelines. As a comparison, Tsai’s full-path indexing method \(^2\) adds an extra 88-bytes overheads for each dentry structure to satisfy their requirement, which breaks the alignment on cachelines.

In Candidate Set, the members are dentries linked to each other as a list through two internal pointers which is included in `struct dentry` as new members. Thus, candidates take no extended space in our design.

Different from a candidate, a pivot is a new data structure in the memory. Memory footprint of a pivot depends on the number of `struct component` (see Fig.5). Our implementation takes about 7KB for a 16-sized Pivot Pool in the 64-bit system, suppose each pivot hold eight `struct component`.

## IV. Evaluation

In order to evaluate the performance of Stage Lookup, we compare Stage Lookup with the other two strategies, say the kernel’s Original Lookup and Tsai’s directory cache \(^2\). Since Tsai’s strategy is developed on Linux kernel v3.14, we implement Stage Lookup on the same version of the kernel.

All our tests are executed on a server with 4-cored 3.3 GHz Intel Core Xeon CPU, 8GB RAM, and a 1TB 7200 RPM disk formatted as an ext4 file system. The operating system is Ubuntu 14.04 Server equipped with a 64-bit Linux kernel v3.14. Rather than overwriting the original system calls, we create new system calls such as `stage_stat`, `stage_open`, `stage_rename` and etc. in the kernel.

### A. Lookup performance

We use the v3.0 LMBench microbenchmark to measure the latency of `stat` and `open`.

We explore the performance of Stage Lookup when Stage Two walks different lengths. Further, we show the impact of the pivot’s quantity during the searching. In Fig.7, the nine groups of bars give the latency of Stage Lookup when Stage Two has to walk 0-8 components. The first bar in each group represents the latency of Original Lookup. And the rest bars show the performances of Stage Lookup when Pivot Pool includes different numbers(1,2,4,8,16) of pivots. The first group represents that Stage One can skip the whole given path, while the last group means that Stage One completely becomes overheads since all workloads are paths with exactly eight components. Stage Lookup performs up to 46.9% compared to Original Lookup.

Fig.7 also shows the characteristic of Stage Lookup’s overheads in. On the one hand, it is reasonable that the more components for Stage Two, the worse Stage Lookup behaves. Because it is similar to Original Lookup and affected by the path length (see Fig.1). On the other hand, redundant pivots we insert are totally useless for Stage One but slow down the process, while only the last pivot is effective. However, path lookup in the real-world can more probably benefit from pivots since they are those popular nodes on the directory tree.

### B. Real-world Simulation

Then we turn to evaluate the performance of the three lookup strategies in a real-world environment. In a high-performance computing(HPC) cluster, I/O forwarding nodes receive requests from hundreds of computing nodes. And the parallel file system, Lustre, for example, acts as an underlying file system by installing a client on the I/O forwarding node. So that VFS-level path lookup on an I/O forwarding node is similar to a local machine in some way. We simulate an I/O forwarding node on our server and run real-world workloads on it.

We replay the `open` operations occurring in 4 days based on what Beacon \(^{19}\) tracked on TaihuLight Supercomputer. We generate a corresponding directory tree with random-sized(0-1MB) files as leaves. To compress the time span, we replace the large intervals between operations by four seconds, while Stage Lookup’s update period is set as two seconds.

Fig.8 shows the comparison of Original Lookup, Stage Lookup, and Tsai’s directory cache method when simulating the I/O forward node. To eliminate the impact of system cache, we clean the inode cache, dentry cache, and page cache before running every single measurement. And each result is the average of six repetitions. Stage Lookup leads the performance in the comparison by showing up to 29.6% compared to Original Lookup and 21.3% to Tsai’s design. Computing nodes in HPC are always running applications whose files are in a directory, which means that Stage Lookup can always skip various lengths of prefixes. Tsai’s directory cache has to initialize each dentry that it meets first. Further, the dual scans of a given path in Tsai’s strategy(see Section III-F) slows down
Fig. 7. Performance of Stage Lookup for two cases: 1) different numbers of components for Stage Two, and 2) different numbers of pivots within Pivot Pool. All the workloads are paths with 8 components. For example, in the second group of bars, we let Original Lookup and Stage Lookup execute `stat` for the same path with 8 components. And Stage Lookup can skip the first six components through Stage One.

Fig. 8. Latency of replaying real-world workloads which are open operations on an I/O forwarding node of TaihuLight Super Computer.

Fig. 9. Performance of `rename` and `chmod` system calls

its performance when comparing with Stage Lookup. Thus, Stage Lookup shows the leading position in this test.

C. Modifying Metadata

Schemes for modifying a directory/file’s metadata are different among the three lookup strategies. For example, when renaming a directory, Original Lookup of the kernel directly modifies the target’s dentry and inode, bring little overheads to the other directories. However, before modifying, Tsai’s directory cache has to visit all the target’s children in the DCache to increase their version values and remove all related entries in its customized cache. Stage Lookup needs to set a flag to invalid a part of Pivot Pool; then “remove” those related pivots from the pool which is asynchronously done by the RCU soft interrupt.

We generate a six-level directory tree owning 10,000+ nodes. Each directory in the first four levels contains 10 children, and the directories of the fifth level include a single 4KB file as a leaf for each. Before every measurement, we free all the page cache, dentry cache, and inode cache. Next, we warm up these system caches and customized cache by executing `stat` for all nodes of the whole six-level tree.

Fig. 9 gives the results of our comparison among the three methods. We work out our results by averaging 10 measurements’ results each of which we sum up the latency of single `rename` or `chmod` operation on a file. Since Stage Lookup holds a limited-size Pivot Pool in ascending order, the process can quickly locate and modify related pivots. And the asynchronous removal of pivots saves latency for a modification. Thus, Stage Lookup introduces little overheads compared with Original Lookup. Tsai’s directory cache, the full-path indexing strategy, is intimately affected by the number of directories, which is related to the target in the dentry cache. Thus, the larger its hash table is, the slower it executes `rename/chmod`. It may cause up to 48.8x latency for `rename` and 94.9x for `chmod` when modifying the directory in Level 1 containing 10,000+ files and directories.

V. CONCLUSION

This paper presents a method called Stage Lookup in the VFS level that reduces the latency of looking up a file. Rather than starting at the root(“/”) directory, we take one of the chosen popular directories/files named pivots as the start point which is closed to the target. Then the process will go forward or backward from the pivot to meet the target. We introduce a heat value to tell how popular a directory/file is, with which the time-window updating strategy can select a number of popular directories/files as pivots periodically. Different from some full-path indexing studies, we can quickly handle metadata modification, such as `rename` and `chmod`. We implement the Stage Lookup on the Linux kernel v3.14 and compare it with the Original lookup and a full-path indexing study. It decreases up to 46.9% path lookup latency compared with the original lookup in the vanilla kernel; and 39.4% compared with a state-of-art study.

REFERENCES

[1] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “A file is not a file: Understanding the i/o behavior of apple desktop applications,” in TOCS, 2011.
[2] C.-C. Tsai, Y. Zhan, J. Reddy, Y. Jiao, T. Zhang, and D. Porter, “How to get more value from your file system directory cache,” 10 2015, pp. 441–456.
[3] L. McVoy and C. Staelin, “Lmbench: Portable tools for performance analysis,” 01 1996, pp. 279–294.
[4] L. Han, B. Xiao, X. Dong, Z. Shen, and Z. Shao, “Ds-cache: A refined directory entry lookup cache with prefix-awareness for mobile devices,” in 2019 Design, Automation Test in Europe Conference Exhibition (DATE), March 2019, pp. 1052–1057.

[5] B. Yang, X. Ji, X. Ma, X. Wang, T. Zhang, X. Zhu, N. El-Sayed, H. Lan, Y. Yang, J. Zhai, W. Liu, and W. Xue, “End-to-end i/o monitoring on a leading supercomputer,” in 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). Boston, MA: USENIX Association, Feb. 2019, pp. 379–394. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/yang

[6] X. Ji, B. Yang, T. Zhang, X. Ma, X. Zhu, X. Wang, N. El-Sayed, J. Zhai, W. Liu, and W. Xue, “Automatic, application-aware i/o forwarding resource allocation,” in 17th USENIX Conference on File and Storage Technologies (FAST 19). Boston, MA: USENIX Association, Feb. 2019, pp. 265–279. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/ji

[7] D. Jeong, Y. Lee, and J.-S. Kim, “Boosting quasi-asynchronous i/o for better responsiveness in mobile devices,” in 13th USENIX Conference on File and Storage Technologies (FAST 15). Santa Clara, CA: USENIX Association, Feb. 2015, pp. 191–202. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/jeong

[8] S. Zhang, H. Catanese, and A. A.-I. Wang, “The composite-file file system: Decoupling the one-to-one mapping of files and metadata for better performance,” in 14th USENIX Conference on File and Storage Technologies (FAST 16). Santa Clara, CA: USENIX Association, Feb. 2016, pp. 15–22. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/zhang-shuanglong

[9] A. Mittal, Y. Hu, and T. Li, “Locafs: a loosely-coupled metadata service for distributed file systems,” 11 2017, pp. 1–12.

[10] K. Ren and G. Gibson, “TABLEFS: Enhancing metadata efficiency in the local file system,” in Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13). San Jose, CA: USENIX, 2013, pp. 145–156. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/ren

[11] H. Tang, S. Byna, B. Dong, J. Liu, and Q. Koziol, “Someta: Scalable object-centric metadata management for high performance computing,” in 2017 IEEE International Conference on Cluster Computing (CLUSTER). Sep. 2017, pp. 359–369.

[12] Q. Zheng, K. Ren, G. Gibson, B. Settlemyer, and G. Grider, “Deltasf: exascale file systems scale better without dedicated servers,” 11 2015, pp. 1–6.

[13] X. Ding, S. Jiang, F. Chen, K. Davis, and X. F. Zhang, “Diskseen: Exploiting disk layout and access history to enhance i/o prefetch.” 01 2007, pp. 261–274.

[14] S. Jiang, X. Ding, Y. Xu, and K. Davis, “A prefetching scheme exploiting both data layout and access history on disk.” ACM Trans. Storage, vol. 9, no. 3, Aug. 2013. [Online]. Available: https://doi.org/10.1145/2508010

[15] D. Bovet and M. Cesati, Understanding The Linux Kernel. Oreilly & Associates Inc, 2005.

[16] Y. Zhan, A. Conway, Y. Jiao, E. Knorr, M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, D. E. Porter, and J. Yuan, “The full path to full-path indexing,” in 16th USENIX Conference on File and Storage Technologies (FAST 18). Oakland, CA: USENIX Association, 2018, pp. 123–138. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/zhan

[17] P. H. Lensing, T. Cortes, and A. Brinkmann, “Direct lookup and hash-based metadata placement for local file systems,” in Proceedings of the 6th International Systems and Storage Conference, ser. SYSTOR 13. New York, NY, USA: ACM, 2013, pp. 5:1–5:11. [Online]. Available: http://doi.acm.org/10.1145/2485732.2485741

[18] W. Jannen, J. Yuan, Y. Zhan, A. Akshinlata, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reidly, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter, “Betrfs: A right-optimized write-optimized file system,” in 13th USENIX Conference on File and Storage Technologies (FAST 15). Santa Clara, CA: USENIX Association, 2015, pp. 301–315. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/jannen

[19] B. Yang, X. Ji, X. Ma, X. Wang, T. Zhang, X. Zhu, N. El-Sayed, H. Lan, Y. Yang, J. Zhai, W. Liu, and W. Xue, “End-to-end i/o monitoring on a leading supercomputer,” in 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). Boston, MA: USENIX Association, Feb. 2019, pp. 379–394. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/yang