How to Get More Value From Your File System Directory Cache

Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter

Stony Brook University

{chitsai, yazhan, jdivakared, yjiao, zhtao, porter}@cs.stonybrook.edu

Abstract

Applications frequently request file system operations that traverse the file system directory tree, such as opening a file or reading a file's metadata. As a result, caching file system directory structure and metadata in memory is an important performance optimization for an OS kernel.

This paper identifies several design principles that can substantially improve hit rate and reduce hit cost transparently to applications and file systems. Specifically, our directory cache design can look up a directory in a constant number of hash table operations, separates finding paths from permission checking, memoizes the results of access control checks, uses signatures to accelerate lookup, and reduces miss rates through caching directory completeness. This design can meet a range of idiosyncratic requirements imposed by POSIX, Linux Security Modules, namespaces, and mount aliases. These optimizations are a significant net improvement for real-world applications, such as improving the throughput of the Dovecot IMAP server by up to 12% and the updatedb utility by up to 29%.

1. Introduction

Operating System kernels commonly cache file system data and metadata in a virtual file system (VFS) layer, which abstracts low-level file systems into a common API, such as POSIX. This caching layer has become a ubiquitous optimization to hide access latency for persistent storage technologies, such as a local disk. The directory cache is not exclusively a performance optimization; it also simplifies the implementation of `mount`-ing multiple file systems, consistent file handle behavior, and advanced security models, such as SELinux [24].

Directory caches are essential for good application performance. Many common system calls must operate on file paths, which require a directory cache lookup. For instance, between 10–20% of all system calls in the iBench system call traces do a path lookup [17]. Figure 1 lists the fraction of total execution time several common command-line applications spend executing path-based system calls (more details on these applications and the test machine in §6). We note that these system calls include work other than path lookup, and that these numbers include some instrumentation overhead; nonetheless, in all cases

SOSP'15, October 4–7, 2015, Monterey, CA.

ACM 978-1-4503-3834-9/15/10. . . $15.00.

http://dx.doi.org/10.1145/2815400.2815405

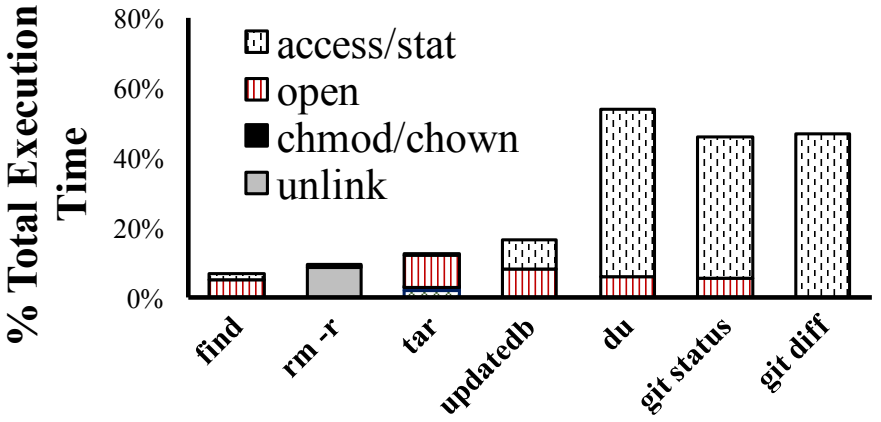


Figure 1: Fraction of execution time in several common utilities spent executing path-based system calls with a warm cache, as measured with ftrace.

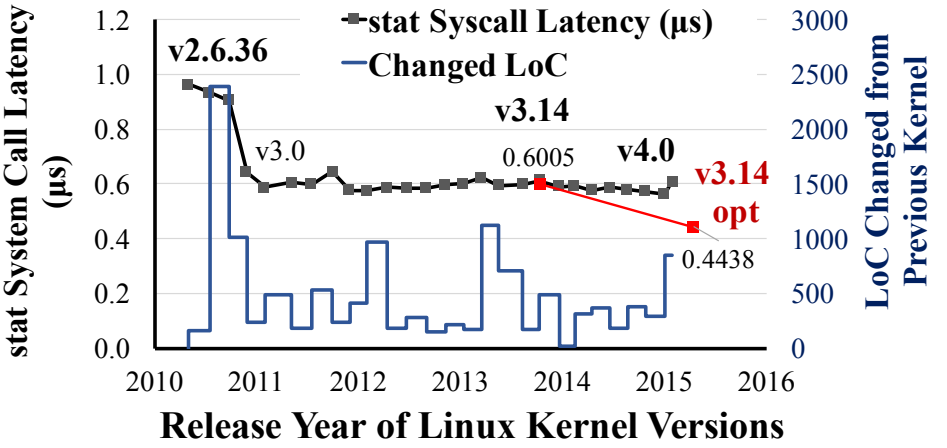


Figure 2: Latency of `stat` system call with a long path `XXX/YYY/ZZZ/AAA/BBB/CCC/DDD/FFF` on Linux over four years (lower is better), as well as the churn within the directory cache code (all insertions in `dcache.c`, `dcache.h`, `namei.c`, `namei.h` and `namespace.c`). Our optimized 3.14 kernel further reduces `stat` system call latency by 26%.

except `rm`, the system call times and counts are dominated by `stat` and `open`, for which path lookup is a significant component of execution time. For these applications, path-based system calls account for 6–54% of total execution time. This implies that lowering path lookup latency is one of the biggest opportunities for a kernel to improve these applications' execution time.

Unfortunately, even directory cache hits are costly—0.3–1.1 μs for a `stat` on our test Linux system, compared to only .04 μs for a `getppid` and 0.3 μs for a 4 KB `pread`. This issue is taken particularly seriously in the Linux kernel community, which has made substantial revisions and increasingly elaborate optimizations to reduce the hit cost of its directory cache, such as removing locks from the read path or replacing lock ordering with deadlock avoidance in a retry loop [11, 12]. Figure 2 plots directory cache hit latency against lines of directory cache code changed over several versions of Linux, using a path-to-inode lookup microbenchmark on the test system described in Section 6. These efforts have improved hit latency by 47% from 2011 to 2013, but have plateaued for the last three years.

The root of the problem is that the POSIX path permission semantics seemingly require work that is linear in the number of path components, and severely limit the kernel developer's implementation options. For instance, in order to open file `/X/Y/Z` one must have search permission to parent directories `/`, `/X`, and `/X/Y`, as well as permission to access file `Z`. The Linux implementation simply walks the directory tree top-down to check permissions. Unfortunately, when the critical path is dominated by walking a pointer-based data structure, including memory barriers on some architectures for multi-core consistency, modern CPUs end up stalling on hard-to-prefetch loads. Moreover, because so many Linux features are built around this behavior, such as Linux Security Modules (LSMs) [47], namespaces, and mount aliases, it is not clear that any data-structural enhancements are possible without breaking backward-compatibility with other Linux kernel features. A priori, it is not obvious that a faster lookup algorithm, such as a single hash table lookup, can meet these API specifications and kernel-internal requirements; to our knowledge, no one has tried previously.

This paper proposes a decomposition of the directory cache, which allows most lookup operations to execute with a single hash table lookup (§3), as well as optimizations to reduce the miss rate based on information that is *already in the cache*, but not used effectively (§5). Our design maintains compatibility (§4) through several essential insights, including how to separate the indexing of paths from checking parent permissions, and how to effectively and safely memoize the results of access control checks.

Our optimizations improve the performance of frequent lookup operations, but introduce several costs, described in §3 and measured in §6, which we believe are acceptable and a net improvement for applications. First, these optimizations slow down infrequent modifications to the directory hierarchy, such as `rename`, `chmod`, and `chown` of a directory. However, these slower operations account for less than .01% of the system calls in the iBench traces [17]. Second, the memory overheads of the dcache are increased. Third, lookup has a probability of error from signature collisions that can be adjusted to be negligible and within acceptable thresholds widely used by data deduplication systems [13, 34, 40, 50]. In the micro-benchmark of Figure 2, our directory cache optimizations improve lookup latency by 26% over unmodified Linux.

This paper demonstrates that these techniques improve performance for applications that use the directory cache heavily, and the harm is minimal to applications that do not benefit. These changes are encapsulated in the VFS—individual file systems do not have to change their code. This paper describes a prototype of these improvements implemented in Linux 3.14. Section 2 explains that the directory cache structure of Mac OS X, FreeBSD, and Solaris are sufficiently similar that these principles should generalize.

The contributions of this paper are as follows:

- A performance analysis of the costs of path lookup and the opportunities to improve cache hit latency.
- A directory cache design that improves path lookup latency with a combination of techniques, including:
  - Indexing the directory cache by full path, reducing average-case lookup from linear to constant in the number of path components.
  - A Prefix Check Cache (PCC) that separates permission checking from path caching. The PCC memoizes permission checks, and is compatible with LSMs [47].
  - Reducing the cost of checking for hash bucket collisions with path signatures.
- Identifying opportunities to leverage metadata the kernel already has to reduce miss rates, such as tracking whether a directory is completely in cache.
- Carefully addressing numerous, subtle edge cases that would frustrate rote application of these techniques, such as integration with symbolic links and Linux namespaces.
- A thorough evaluation of these optimizations. For instance, our optimizations improve throughput of the Dovecot IMAP server by up to 12% and latency of updatedb by up to 29%.

2. Background

This section first reviews the Unix directory semantics which a directory cache must support; and then explains how directory caches are implemented in modern OSes, including Linux, FreeBSD, Solaris, Mac OS X, and Windows.

2.1 Unix Directory Hierarchy Semantics

The most common operation a directory cache performs is a **lookup**, which maps a path string onto an in-memory inode structure. Lookup is called by all path-based system calls, including `open`, `stat`, and `unlink`. Lookup includes a check that the user has appropriate search permission from the process's root or current working directory to the file, which we call a **prefix check**.

For instance, in order for Alice to read `/home/alice/X`, she must have search permission on directories `/`, `/home`, and `/home/alice`, as well as read permission on file `X`. In the interest of frugality, the execute permission bit on directories encodes search permission. Search is distinct from read permission in that search only allows a user to query whether a file exists, but not enumerate the contents (except by brute force) [35]. SELinux [24] and other security-hardened Linux variants [1, 47], may determine search permission based on a number of factors beyond the execute bit, such as a process's role, or the extended attributes of a directory.

2.2 Linux Directory Cache

The Linux directory cache, or **dcache**, caches **dentry** (directory entry) structures, which map a path to an in-memory inode[1] for the file (or directory, device, etc). The inode stores metadata associated with the file, such as size, permissions, and ownership, as well as a pointer to a radix tree that indexes in-memory file contents [4]. Each dentry is tracked by at least four different structures:

- The hierarchical tree structure, where each parent has an unsorted, doubly-linked list of its children.
- A hash table, keyed by parent dentry virtual address and the file name.
- An alias list, tracking the hard links associated with a given inode.
- An LRU list, used to compress the cache as needed.

Linux integrates the prefix check with the lookup itself, searching paths and checking permissions one component at a time. Rather than using the tree structure directly, lookup searches for each component using the hash table. For larger directories, the hash table lookup will be faster than searching an unsorted list of children. The primary use for the hierarchical tree structure is to evict entries bottom-up, in order to uphold the implicit invariant that all parents of any dentry must also be in the cache. Although all dentries are stored in a hash table keyed by path, the permission check implementation looks up each path component in the hash table.

Linux stores **negative dentries**, which cache the fact that a file is known *not* to exist on disk. A common motivating example for negative dentries is searching for a file on multiple paths specified by an environment variable, such as `LD_LIBRARY_PATH`.

Current dcache optimizations. Much of the dcache optimization effort illustrated in Figure 2 has improved cache hit latency, primarily by reducing the cost of synchronization in the lookup function with read-copy update (RCU) [26, 27]. RCU eliminates the atomic instructions needed for a read lock and for reference counting individual dentries, pushing some additional work onto infrequent code that modifies the directory structure, such as `rename` and `unlink`.

The most recent Linux kernels also use optimistic synchronization when checking path permissions, using sequence locks (essentially version counters), to detect when the subtree might have changed concurrently with the traversal. If the optimistic fast path fails because

[1] Other Unix systems call the VFS-level representation of an inode a vnode.

of a concurrent modification, the kernel falls back on a slow path that uses hand-over-hand locking of parent and child dentries.

Because the Linux developer community has already invested considerable effort in optimizing its dcache, we use Linux as a case study in this paper. The optimizations in this paper are not Linux-specific, but in some cases build on optimizations that other kernels could adopt.

2.3 Other Operating Systems

FreeBSD, OS X, and Solaris. These Unix variants all have a directory cache that is structurally similar to Linux's [25, 28, 39]. Each system organizes its directory cache with a hash table, checks paths one component at a time, and stores negative dentries. Here we use FreeBSD as a representative example of the BSD family, and the most popular according to recent surveys [6]. The OS X kernel adopted its file system from FreeBSD, and has not substantially changed their behavior with respect to directory metadata caching [31].

Linux is distinctive in that the hit path avoids calling the low-level file system, whereas other Unix variants always call the low-level file system. A low-level file system may opt out of the default structures if it has a more specialized structure, say for large directories, or it may directly implement its own lookup function. Directly managing a file system's portion of the cache is problematic because mount points are not visible to the low-level file system. Several previous works have found this restriction onerous, especially for network file systems [14]. These Unix variants also do not use optimistic synchronization in their dcaches, but this is not fundamental.

The Solaris dcache, called the Direct Name Lookup Cache (DNLC), features sophisticated cache management heuristics, such as weighing relevance as well as temporal locality in replacement decisions [25] Solaris also has a simpler reference management system for cached paths than FreeBSD (and more similar to Linux) [16].

Windows. Essentially all OS API abstractions in Windows are represented with objects, managed by the Object Manager [36]. The Object Manager is the closest analog to the Unix directory cache, which tracks hierarchical paths and permissions. Unfortunately, public documentation on Windows internals is limited, especially so for internal data structures and caching policies for metadata not in active use, so a detailed comparison is difficult. Nonetheless, we can compare the impact of some high-level design choices.

First, Windows only supports one root file system format, and a very limited number of other file systems. Thus, there is less value in a general-purpose, in-memory organization for file system metadata, and Windows does not have vnodes, dentries, or other VFS-level generalizations. Instead, caching is primarily the responsibility of the file system, and on-disk and in-memory structure layouts may be the same.

Unlike Unix variants, when a Windows file system path is not cached in the Object Manager, the low-level file system is responsible for resolving the full path, rather than one component at a time. For this to work, Windows NT also propagates parent directory permissions to each child's on-disk metadata at creation or modification time [41]. This approach allows for direct lookup, but also creates a subtle manageability problem. Suppose Alice makes her home directory world readable: should this change be propagated to all sub-directories? To answer this question, Windows adopts an error-prone heuristic of not changing manually-modified child permissions. This paper shows how to keep the performance benefits of direct lookup in memory without the manageability issues of *storing* propagated hierarchical permissions on disk.

2.4 Opportunities for Improvement

Figure 3 shows the time spent in the principal components of a path lookup in Linux, for four paths of increasing lengths. The first-order impact on lookup time is the length of the path itself, which dictates how many times each component will be hashed, looked-up in the hash table, and execute a permission check on each directory's inode. These costs are linear in the number of path components.

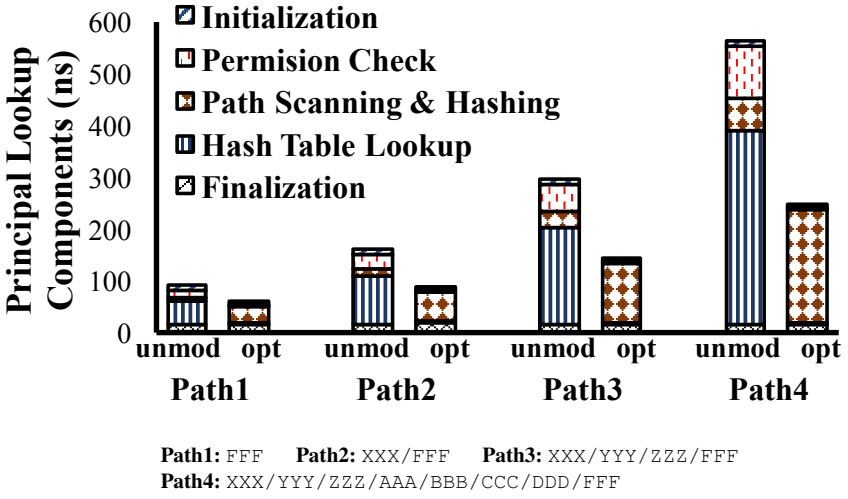


Figure 3: Principal sources of path lookup latency in the Linux 3.14 kernel. Lower is better.

The hit latency optimizations described in this paper make most of these operations constant time, except for hashing, which is still a function of the length of the path.

3. Minimizing Hit Latency

This section describes algorithmic improvements to the dcache hit path. In the case of a cache hit, one of the most expensive operations is checking whether a process's credentials permit the process to search the path to a dentry top-down (called a **prefix check**). This section shows how the hit latency can be significantly reduced by caching prefix check results. This section explains the optimization, how it is integrated into the existing Linux directory cache framework, how these cached results are kept coherent with other file system operations, and how we use path signatures to further accelerate lookup.

3.1 Caching Prefix Checks

Like many Unix variants, Linux stores cached path-to-inode mappings (dentries) in a hash table (§2.2). This hash table is keyed by a combination of the virtual address of the parent dentry and the next path component string, illustrated in Figure 4. Virtual addresses of kernel objects do not change over time and are identical across processes.

In practice, prefix checks have a high degree of spatial and temporal locality, and are highly suitable for caching, even if this means pushing some additional work onto infrequent modifications of the directory structure (e.g., `rename` of a directory). RCU already makes this trade-off (§2.2).

In order to cache prefix check results, we must first decouple *finding* a dentry from the prefix check. We added a second, system-wide hash table exclusively for finding a dentry, called the **direct lookup hash table (DLHT)**. The DLHT stores recently-accessed dentries hashed by the full, canonicalized path. A dentry always exists in the primary hash table as usual, and may exist in the DLHT. The DLHT is lazily populated, and entries can be removed for coherence with directory tree modifications (§3.2).

Each process caches the result of previous prefix checks in a **prefix check cache** (PCC), associated with the process's credentials (discussed further in §4.1), which can be shared among processes with identical permissions. The PCC is a hash table that caches dentry virtual addresses and a version number (sequence lock), used to detect stale entries (§3.2). When a prefix check passes, indicating that the credentials are allowed to access the dentry, an entry is added to the PCC; entries are replaced according to an LRU policy. A miss in the PCC can indicate a permission denied or that the permission check has not executed recently.

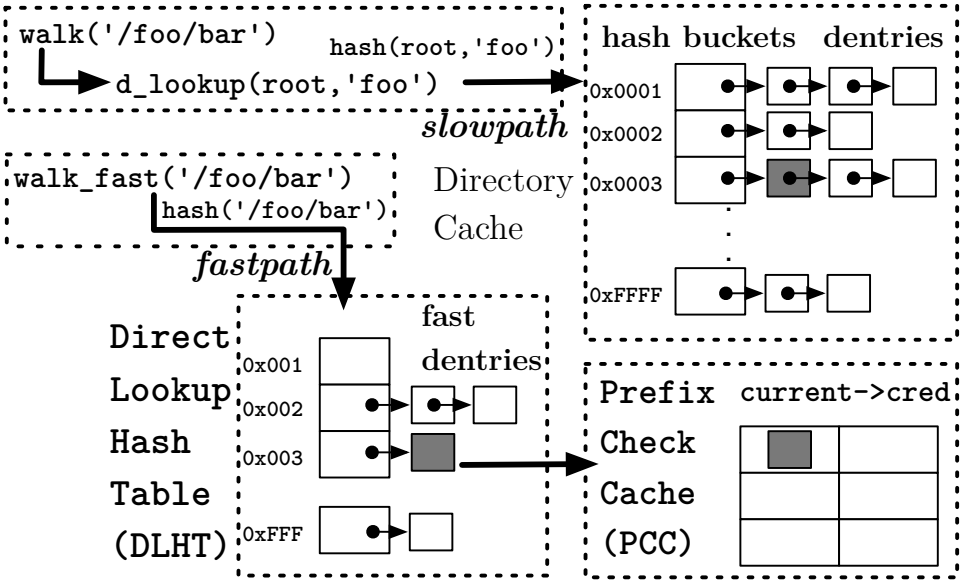


Figure 4: Optimized Linux Directory Cache Structure. dentries are chained in hash buckets. To index the hash bucket for a target dentry, the original lookup routine `d_lookup` uses a hashing function with key as a combination of the pointer to parent directory and file name (*slowpath*). Our *fastpath* hashes the full canonical path of target file to look up the dentry in the Direct Lookup Hash Table, and checks the per-credential Prefix Check Cache.

Thus, given any path, the kernel has a *fastpath* that directly looks up the path in the DLHT. If the fastpath hits in the DLHT, the dentry is then looked up in the process's PCC. If a PCC entry is found and the version counter matches the cached counter, the cached prefix check result is used. If the fastpath lookup misses in the DLHT or PCC, or the version counter in the PCC entry is older than the dentry, the code falls back on the original Linux lookup algorithm (the *slowpath*), using the primary hashtable exclusively and traversing one component at a time.

In the case of a relative path, such as `foo/bar` under directory `/home/alice`, we effectively concatenate the relative path and the path of the current working directory. To implement relative paths, Linux already stores a pointer to the `dentry` of the current working directory in each process descriptor (`task_struct`). Rather than `memcpy` the strings, we store the intermediate state of the hash function in each `dentry` so that hashing can resume from any prefix.

The current design includes two very infrequent edge cases. First, a dentry could be freed and reallocated with stale PCC entries. We detect this case by initializing newly allocated dentries with a monotonically increasing version number, allowing PCC entries to detect staleness across reallocation. Freeing a dentry removes it from the DLHT. Second, a version number can wrap around after every 2^{32} initializations of new dentries or renames, chmods, or chowns of non-empty directories; our design currently handles wrap-around by invalidating all active PCCs.

Figure 5 illustrates the modifications to the Linux dentry structure. The `fast_dentry` stores the signature, flags, a sequence count, a mount point, lists for managing deep directory entries (§5.2), and a list (`hash_chain`) for adding the fast_dentry to a DLHT bucket. The PCC is added to the kernel credential structure (`struct cred`), and stores a tunable number of tuples of dentry pointers and sequence numbers; the system is evaluated with a PCC of 64 KB. Because the highest and lowest bits in each dentry pointer are identical, the PCC only stores the unique pointer bits (8–39 in x86_64 Linux) to save space.

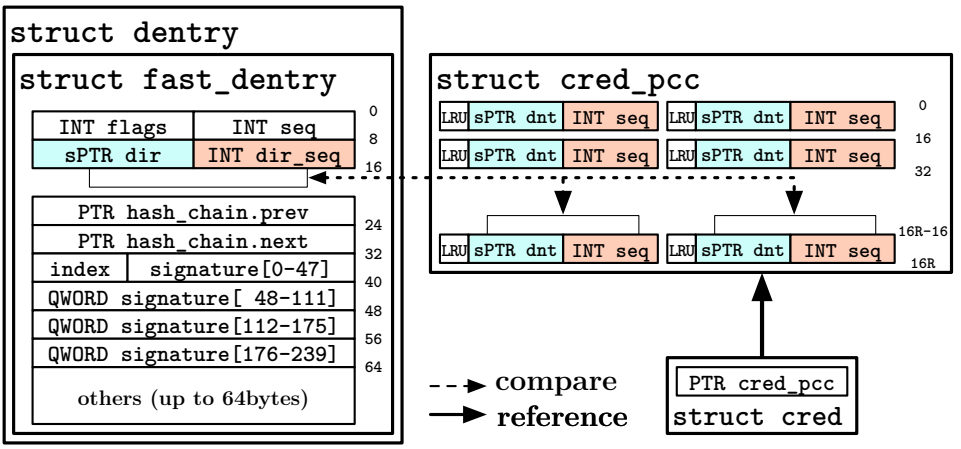


Figure 5: Data structures added for fast directory cache lookup. To support *fastpath* lookup, we add a 88-byte fast dentry structure to the original dentry and a variable-sized PCC structure into `cred`.

3.2 Coherence with Permission and Path Changes

When permissions on a directory or the directory structure are changed, such as with `chmod` or `rename`, any cached prefix checks that include this directory must be invalidated. Our design ensures the safety of concurrent lookups and changes by invalidating relevant PCC and DLHT entries before a change to the hierarchy, preventing stale slowpath lookups from being re-cached, and leveraging VFS-level synchronization to ensure correct slowpath behavior.

First, we ensure that a fastpath lookup cannot complete with stale data after a change to the directory structure. Before a mutation, such as a `rename` or `chmod`, the operation must recursively walk all children in the dcache and increment the `fast_dentry` version counter (`seq`). The `fast_dentry` version counter is used by each process's PCC to detect changes to cached prefix checks on a lookup; incrementing this version counter invalidates all PCC entries for that dentry without directly modifying each PCC. Changes to the directory structure (e.g., `mount` and `rename`) also remove dentries under the old and new path from the direct lookup hash table (DLHT). PCC and DLHT entries are lazily repopulated on the slowpath.

Second, we ensure that the results of a stale slowpath lookup cannot be re-added to the DLHT or PCC by using an atomic, global sequence counter (`invalidation`). The sequence counter is read before and after a slowpath traversal; results are added to the DLHT and PCC only if the counter has not changed, implying no concurrent shootdowns.

Third, we use VFS-level synchronization to ensure that slowpaths synchronize correctly with the mutation. As an example, `rename` acquires both a global `rename_lock` sequence lock, along with per-dentry locks on the old and new parent directory. When the `rename_lock` is held for writing, all lookups on the slowpath (i.e., the current Linux code) must lock each dentry in a hand-over-hand fashion from the root (or current working directory, for relative paths) to the target child. The locks on target dentries obstruct the hand-over-hand traversal until the rename completes. The `invalidation` counter prevents caching the results of slowpath lookups that already passed this point before the dentry locks were acquired. Our implementation follows the VFS's existing locking discipline to avoid deadlocks; it adds version counters that detect inconsistencies and falls back on the slowpath. Thus, relevant PCC and DLHT entries are invalidated before the rename begins, blocking the fastpath; slowpath traversals will block until the rename is complete and the per-dentry locks are released; and a sequence counter ensures that only slowpath traversals that observe the new paths can repopulate the DLHT and PCC.

These recursive traversals shift directory permission and structure changes from constant time to linear in the size of the sub-tree. As one example, to `rename` or `chmod` a directory that has 10,000 descendants with at most depth of 4 takes roughly 330 microseconds to complete. In the original Linux kernel, `rename` and `chmod` are nearly constant-time operations, and only take 4.5 and 1.1 microseconds. A few applications, such as `aptitude` or `rsync`, rely on `rename` to atomically replace a directory, but this is a small fraction of their total work and orders of magnitude less frequent than lookups, making this a good trade-off overall.

Directory References. Unix semantics allow one to `cd` into a directory, and continue working in that directory after a subsequent permission change would otherwise prohibit further accesses. For instance, suppose a process is in working directory `/foo/bar` and `foo`'s permissions change such that the process would not be able to enter `bar` in the future. The process should be able continue to open files under `bar` as long as the process does not leave the directory or exit. Similar semantics apply to open directory handles. In our design, such a permission change would ultimately result in a blocked PCC entry, and a fastpath lookup would violate the expected behavior. Our design maintains compatibility by checking if the open reference is still permitted in the PCC. If the PCC has a more recent entry that would prevent re-opening this handle, the lookup is forced to take the the slowpath, and this stale result is not added to the PCC.

3.3 Accelerating Lookups with Signatures

Our optimized lookup uses 240-bit signatures to minimize the cost of key comparison. Linux looks up dentries in a hash table with chaining. When the hash table key is a relatively short path component, the cost of simply comparing the keys is acceptable. However, a full path on Linux can be up to 4,096 characters, and comparing even modest-length strings can erode the algorithmic benefits of direct lookup. We avoid this cost by creating a signature of the path, which minimizes the cost of key comparisons.

Using signatures introduces a risk of collisions, which could cause the system to map a path onto the wrong dentry. We first explain how signature collisions could cause problems in our design, followed by the required collision resistance properties, and, finally, how we selected the signature size to make this risk vanishingly small.

Signature collisions. When a user looks up a path, our design first calculates a signature of the canonicalized path, looks up the hash in the global DLHT, and, if there is a hit in the DLHT, looks up the dentry and sequence number in the per-credential PCC.

A user can open the wrong file if the dentry for another file with the same signature is already in the DLHT, and that dentry is in the PCC. For example, if Alice has opened file `/home/alice/foo` with signature X, and then opens file `/home/alice/bar` that also has signature X, her second open will actually create a handle to file `foo`. This creates the concern that a user might corrupt her own files through no fault of her own. This risk can be configured to be vanishingly small based on the signature size (discussed below).

Any incorrect lookup result must be a file that the process (or another process with the same credentials) has permission to access. For a fastpath lookup to return anything, a matching dentry pointer must be in the task's PCC, which is private to tasks with the same credentials. Thus, a collision will not cause Alice to accidentally open completely irrelevant files that belong to Bob, which she could not otherwise access.

Our design correctly handles the case where two users access different files with the same signature, because misses in the PCC will cause both users to fall back on the slowpath. Suppose Bob has opened `foo`, which collides with Alice's `bar`. When Alice opens `bar`, its signature will match in the DLHT, but will miss in the PCC. This causes Alice's lookup to take the slowpath to re-execute the prefix check, ultimately opening the correct file and adding this dentry to her PCC. Thus, if Bob were adversarial, he cannot cause Alice to open the wrong file by changing dcache-internal state.

We choose a random key at boot time for our signature hash function, mitigating the risk of deterministic errors or offline collision generation, as one might use to attack an

application that opens a file based on user input, such as web server. Thus, the same path will not generate the same signature across reboots or instances of the same kernel.

Despite all of these measures, this risk may still be unacceptable for applications running as root, which can open any file, especially those that accept input from an untrusted user. For example, suppose a malicious user has identified a path with the same signature as the password database. This user might pass this path to a setuid-root utility and trick the setuid utility into overwriting the password database. This risk could be eliminated by disallowing signature-based lookup acceleration for privileged binaries or security-sensitive path names, although this is not implemented in our prototype.

Collision Resistance Requirements. The security of our design hinges on an adversary only being able to find collisions through brute force. Our design can use either a 2-universal hash function or a pseudorandom function family (PRF) to generate path signatures. In terms of collision resistance, the difference between a 2-universal hash and a PRF is that the adversary can potentially learn the secret key by observing the outputs of the 2-universal function, but cannot learn the key from the outputs of a PRF. Because our dcache design does not reveal the signatures to the user, only whether two paths have a signature collision, a hash function from either family is sufficient.

One caveat is that, with a 2-universal hash function, one must be careful that timing and other side channels do not leak the signature. For example, one cannot use bits from the signature to also index the hash table, as one might learn bits of the signature from measuring time to walk the chain on a given hash bucket. In the case of our selected function, one can safely use the lower bits from the 256-bit hash output, as lower bits are not influenced by the values in higher bits in our particular algorithm; we thus use a 16 bit hash table index and a 240-bit signature. In contrast, when the signature is generated with a PRF, concerns about learning the signature from side channels are obviated.

Our design uses the 2-universal multilinear hash function [21]. We did several experiments using PRFs based on the AES-NI hardware, and could not find a function that was fast enough to improve over baseline Linux. Using current 128-bit AES hardware, we could improve performance at 4 or more path components, but creating a 256-bit PRF required a more elaborate construction that is too expensive. A more cautious implementation might favor a PRF to avoid any risk of overlooked side channels, especially if a fast, 256-bit PRF becomes available in future generations of hardware.

Probability of a signature collision. We selected a 240-bit signature, which is comparable to signature sizes used in data deduplication systems, ranging from 128–256 bits. Deduplication designs commonly select a signature size that introduces a risk of collisions substantially less than the risk of undetected ECC RAM errors [13, 34, 40, 50].

We assume an adversary that is searching for collisions by brute force. This adversary must lookup paths on the system, such as by opening local files or querying paths on a web server. Because our hash function is keyed with a random value and the output is hidden from the user, the adversary cannot search for collisions except on the target system. Thus, the adversary is limited by the rate of lookups on the system, as well as the capacity of the target system to hold multiple signatures in cache for comparison.

We calculate the expected time at which the risk of a collision becomes non-negligible (i.e., higher than 2^{-128}) and model the risk of collision as follows. First, $|H(X)| = 2^{240}$ is the number of possible signatures. We limit the cache to $n = 2^{35}$ entries (i.e., assuming 10TB of dcache space in RAM and 320 bytes per entry), with an LRU replacement policy. We calculate the number of queries (q) after which the risk of a collision is higher than $P = 2^{-128}$ as follows:

$$q \simeq ln(1-p) * \frac{|H(x)|}{-n} \simeq ln(1-2^{-128}) * \frac{2^{240}}{-2^{35}} \simeq 2^{77}$$

At a very generous lookup rate of 100 billion per second (current cores can do roughly 3 million per second), the expected time at which the probability of a brute-force collision goes above 2^{-128} is 48 thousand years.

4. Generalizing the Fast Path

Thus far, we have explained our fast path optimization using the relatively straightforward case of canonical path names. This section explains how these optimizations integrate with Linux's advanced security modules, as well as how we address a number of edge cases in Unix path semantics, such as mount options, mount aliases, and symbolic links.

4.1 Generalizing Credentials

Linux includes an extensible security module framework (LSMs [47]), upon which SELinux [24], AppArmor [1], and others are built. An LSM can override the implementation of search permission checks, checking customized attributes of the directory hierarchy or process. Thus, our dcache optimizations must still work correctly even when an LSM overrides the default access control rules.

Our approach leverages the `cred` structure in Linux, which is designed to store the credentials of a process (`task_struct`), and has several useful properties. First, a `cred` struct is comprehensive, including all variables that influence default permissions, and including an opaque `security` pointer for an LSM to store metadata. Second, a `cred` is copy-on-write (COW), so when a process changes its credentials, such as by executing a `setuid` binary or changing roles in SELinux, the `cred` is copied. We manually checked that AppArmor and SELinux respect the COW conventions for changes to private metadata. Moreover, a `cred` can be shared by processes in common cases, such as a shell script forking children with the same credentials. Thus, the `cred` structure meets most of our needs, with a few changes, which we explain below.

We store cached prefix checks (§3.1) in each `cred` structure, coupling prefix check results with immutable credentials. New `cred` structures are initialized with an empty PCC. When more processes share the PCC, they can further reduce the number of slowpath lookups.

One challenge is that Linux often allocates new `cred` structures *even when credentials do not change*. The underlying issue is that COW behavior is not implemented in the page tables, but rather by convention in code that *might* modify the `cred`. In many cases, such as in `exec`, it is simpler to just allocate another `cred` in advance, rather than determine whether the credentials will be changed. This liberal allocation of new `cred`s creates a problem for reusing prefix cache entries across child processes with the same credentials. To mitigate this problem, we wait until a new `cred` is applied to a process (`commit_creds()`). If the contents of the `cred` did not change, the old `cred` and PCC is reused and shared.

Our `cred` approach memoizes complex and potentially arbitrary permission evaluation functions of different LSMs.

4.2 Non-Canonical Paths and Symbolic Links

Our optimized hash table is keyed by full path. However, a user may specify variations of a path, such as `/X/Y/./Z` for `/X/Y/Z`. Simple variations are easily canonicalized in the course of hashing.

A more complex case is where, if `/X/L` is a symbolic link, the path `/X/L/../Y` could map to a path other than `/X/Y`. Similarly, if the user doesn't have permission to search `/X/Z`, a lookup of `/X/Z/../Y` should fail even if user has permission to search `/X/Y`. In order to maintain bug-for-bug compatibility with Linux, our prototype issues an additional fastpath lookup at each dot-dot to check permissions. Maintaining Unix semantics introduces overhead for non-canonical paths.

We see significantly higher performance by using Plan 9's *lexical* path semantics [33]. Plan 9 minimized network file system lookups by pre-processing paths such as `/X/L/../Y` to `/X/Y`. We note that Plan 9 does component-at-a-time lookup, but does not have a directory cache.

Symbolic Links. We resolve symbolic links on our lookup fastpath by creating dentry aliases for symbolic links. For instance, if the path `/X/L` is an alias to `/X/Y`, our kernel

will create dentries that redirect `/X/L/Z` to `/X/Y/Z`. In other words, symbolic links are treated as a special directory type, and can create children, caching the translation.

Symbolic link dentries store the 240-bit signatures that represent the target path. The PCC is separately checked for the target dentry. If a symbolic link changes, we must invalidate all descendant aliases, similar the invalidation for a directory `rename`. This redirection seamlessly handles the cases where permission changes happen on the translated path, or the referenced dentries are removed to reclaim space.

4.3 Mount Points

Our fastpath handles several subtle edge cases introduced by mount points.

Mount options. Mount options, such as read-only or nosuid, can influence file access permission checks. The Linux dcache generally notices mount points as part of the hierarchical file system walk, and checks for permission-relevant mount flags inline. Once this top-down walk is eliminated, we need to be able to identify the current mount point for any given dentry. We currently add a pointer to each dentry, although more space efficient options are possible.

Mount Aliases. Some pseudo file systems, such as `proc`, `dev`, and `sysfs`, can have the same *instance* mounted at multiple places. This feature is used by `chroot` environments and to move these file systems during boot. A `bind` mount can also create a mount alias.

In our system, a dentry only stores one signature and can only be in the direct lookup hash table by one path at a time. Our current design simply picks the most recent to optimize—favoring locality. If a slowpath walk notices that the matching dentry (by path) has a different signature, is under an aliased mount, and is already in the DLHT, the slowpath will replace the signature, increment the dentry version count, and update the pointer to the dentry's mount point. The version count increment is needed in case the aliased paths have different prefix check results. This approach ensures correctness in all cases, and good performance on the most recently used path for any mount-aliased dentry.

Mount Namespaces. Mount namespaces in Linux allow processes to create private mount points, including `chroot` environments, that are only visible to the process and its descendants. When a process creates a new mount namespace, it also allocates a new, namespace-private direct lookup hash table. The slowpath always incorporates any mount redirection, and any new signature-to-dentry mappings will be correct within the namespace. Thus, the same path (and signature) inside a namespace will map to a different dentry than outside of the namespace. Similarly, the prefix check cache (PCC) will always be private within the namespace.

As with mount aliases, we only allow a dentry to exist on one direct lookup hash table at a time. This favors locality, and makes the invalidation task tractable when a renamed directory is shared across many namespaces. The invalidation code used for directory tree modifications simply evicts each child dentry from whatever DLHT it is currently stored in.

Network File Systems. Our prototype does not support direct lookup on network file systems, such as NFS versions 2 and 3 [37]. In order to implement close-to-open consistency on a stateless protocol, the client must revalidate all path components at the server—effectively forcing a cache miss and nullifying any benefit to the hit path. We expect these optimizations could benefit a stateful protocol with callbacks on directory modification, such as AFS [19] or NFS 4.1 [38].

4.4 Summary

This section demonstrates how our directory cache optimizations can support the wide range of features Linux has built upon the directory cache, including namespaces, enhanced security modules, and symbolic links. Our prototype focuses on Linux, which has arguably the most features intertwined with its directory cache, but we believe these optimizations would work in other systems, with modest porting effort.

Our design has the following requirements, which we expect would be met by any POSIX-compliant directory cache. First, POSIX permission semantics require directory access checks on the path from the current root or working directory to the file (i.e., prefix checking); our implementation inherits Linux's invariant that any cached directory's parents are in the cache, but any design that can implement prefix checking should suffice. Second, we require that, if a directory's permissions change, there is a programmatic way to find all descendants in the cache (§3.2). Our implementation integrates with optimistic synchronization in the Linux dcache for good performance and consistency, but this design could integrate with any reasonable synchronization scheme, such as FreeBSD's reader/writer locks. Finally, we leverage the fact that Linux has an immutable credentials structure (§4.1); adapting to mutable or less consolidated credentials would require extra effort.

5. Improving the Hit Rate

The previous sections explain how changes to the structure of the dcache can lower the average hit latency, through algorithmic improvements. This section identifies several simple changes that can improve the hit rate. In the case of a dcache miss, the low-level file system is called to service the system call. At best, the on-disk metadata format is still in the page cache, but must be translated to a generic format; at worst, the request blocks on disk I/O. Although not every application heavily exercises these cases with unusually low hit rates, the evaluation shows several widely-used applications that substantially benefit from these optimizations.

5.1 Caching Directory Completeness

Although the Linux dcache tracks the hierarchical structure of directories, it has no notion of whether a directory's contents are completely or partially in the cache. Suppose Alice creates a new directory `X` on a local file system; if her next system call attempts to create file `X/Y`, the dcache will miss on this lookup and ask the low-level file system if `X/Y` exists. This overhead can be avoided if the VFS tracks that all directory contents are in the cache.

A second example is `readdir`, which lists the files in a directory, along with their inode number and their types, such as a regular file, character device, directory, or symbolic link. In the current VFS `readdir` operation, the low-level file system is always called, *even if the entire directory is in cache*. For directories too large to list in the user-supplied buffer, `readdir` may be called multiple times, storing an offset into the directory. To construct this listing, the low-level file system must reparse and translate the on-disk format, and may need to read the metadata block from disk into the buffer cache. As a result, `readdir` is generally an expensive file system operation, especially for large directories.

We observe that repeatedly listing a directory is a common behavior in file systems. For example, a user or a shell script may repeatedly run the `ls` command in a directory. Some applications coordinate state through directory contents, requiring frequent and repeated directory listings. For example, `maildir` is a popular email back-end storage format [2], yielding better performance scalability than the older `mbox` format. Maildir stores each inbox or subfolder as a directory, and each individual message is a file within the directory. File names encode attributes including flags and read/unread status. If a message changes state, such as through deletion or being marked as read, the IMAP server server will rename or unlink the file, and reread the directory to sync up the mail list. Similarly, a mail delivery agent (MDA), running as a separate process, may concurrently write new messages into the directory, requiring the IMAP server to monitor the directory for changes and periodically re-read the directory's contents.

Our Linux variant caches `readdir` results returned by the low-level file system in the directory cache. If all of a directory's children are in the cache, the dentry is marked with a new `DIR_COMPLETE` flag. This flag is set upon creation of a new directory (`mkdir`), or when a series of `readdir` system calls completes without an `lseek()` on the directory handle or a concurrent eviction of any children to reclaim space. We note that concurrent

file creations or deletions interleaved with a series of `readdir`s will still be in the cache and yield correct listing results. After setting the `DIR_COMPLETE` flag, subsequent `readdir` requests will be serviced directly from the dentry's child list. Once a directory enters the complete state, it leaves this state only if a child dentry is removed from the cache to reclaim space.

One caveat to this approach is that readdir returns part of the information that would normally appear in an inode, but not enough to create a complete inode. For these files or subdirectories, we add dentries without an inode as children of the directory. These dentries must be separated from negative dentries when they are looked up, and be linked with a proper inode. This approach allows `readdir` results to be used for subsequent `lookup`s, cleanly integrates with existing dcache mechanisms, and gets the most possible use from every disk I/O without inducing I/O that was not required.

We note that Solaris includes a similar complete directory caching mode [25], but it is not integrated with readdir or calls other than lookup, is a separate cache (so the same dentries can be stored twice, and both hash tables must be checked before missing), and the comments indicate that it only has performance value for large directories. Our results demonstrate that, when properly integrated into the directory cache, tracking complete directories has more value than previously thought.

File Creation. Directory completeness caching can also avoid compulsory misses on new file creation. Although negative dentry caching works well for repeated queries for specific files that do not exist, negative dentries are less effective when an application requests *different* files that do not exist. A common example of unpredictable lookups comes from secure temporary file creation utilities [8]. In our prototype, a miss under a directory with the `DIR_COMPLETED` flag is treated as if a negative dentry were found, eliding this compulsory miss. In our current implementation, this flag will only be set in a directory that has been read or newly created, but other heuristics to detect frequent misses for negative dentries and to load the directory may also be useful.

5.2 Aggressive Negative Caching

Negative dentries cache the fact that a path does not exist on disk. This subsection identifies several opportunities for more aggressive use of negative dentries, some of which work in tandem with direct lookup.

Renaming and Deletion. When a file is renamed or unlinked, the old path can be converted to a negative dentry. Although Linux does convert a cached, but unused dentry to a negative dentry on unlink, this is not the case for `rename` and `unlink` of a file that is still in use. We extend these routines to keep negative dentries after a file is removed, in the case that the path is reused later, as happens with creation of lock files or Emacs's backup ("tilde") files.

Pseudo File Systems. Pseudo file systems, like `proc`, `sys`, and `dev`, do not create negative dentries for searched, nonexistent paths. This is a simplification based on the observation that disk I/O will never be involved in a miss. Because our fastpath is still considerably faster than a miss, negative dentries can be beneficial even for in-memory file systems, accelerating lookup of frequently-searched files that do not exist.

Deep Negative Dentries. Finally, we extended the direct lookup fastpath (§3) with the ability to create "deep" negative dentries. Consider the case where a user tries to open `/X/Y/Z/A`, and `/X/Y/Z` does not exist. In the slowpath, the lookup will fail when it hits the first missing component, and it is sufficient to only cache a negative dentry for `Z`. Repeated lookups for this path will never hit on the fastpath, however, because there is no entry for the full path.

In order for this case to use the fastpath, we allow negative dentries to create negative children, as well as deep children. In other words, we allow negative dentry `/X/Y/Z` to create children `A` and `A/B`, which can service repeated requests for a non-existent path. If

a file is created for a path that is cached as negative, and the file is not a directory, any negative children are evicted from the cache.

We also create deep negative dentries under regular files to capture lookup failures that return `ENOTDIR` instead of `ENOENT`. This type of lookup failure happens when a filename is use as if it were a directory, and a path underneath is searched. For example, if `/X/Y/Z` is a regular file, and a user searches for `/X/Y/Z/A`, the Linux kernel will return `ENOTDIR` and never create a negative dentry. We optimize this case with a deep, `ENOTDIR` dentry.

6. Evaluation

This section evaluates our directory cache optimizations, and seeks to answer the following questions:

1. How much does each optimization—the lookup fastpath, whole directory caching, and more aggressive negative dentries—improve application performance?
2. How difficult are the changes to adopt, especially for individual file systems?

The evaluation includes both micro-benchmarks to measure the latency of file system related system calls in best-case and worst-case scenarios, and a selection of real-world applications to show potential performance boost by our solution in practice.

All experiment results are collected on a Supermicro Super Server with a 12-core 2.40 GHz Intel Core Xeon CPU, 64GB RAM, and a 2 TB, 7200 RPM ATA disk, formatted as a journaled `ext4` file system, configured with a 4096-byte block size. The OS is Ubuntu 14.04 server, Linux kernel 3.14. All measurements are a mean of at least 6 runs (for the longer-running experiments); most measurements are hundreds or thousands of runs, as needed to ensure a consistent average. Tables and graphs indicate 95% confidence intervals with "+/-" columns or error bars.

6.1 File Lookup Optimizations

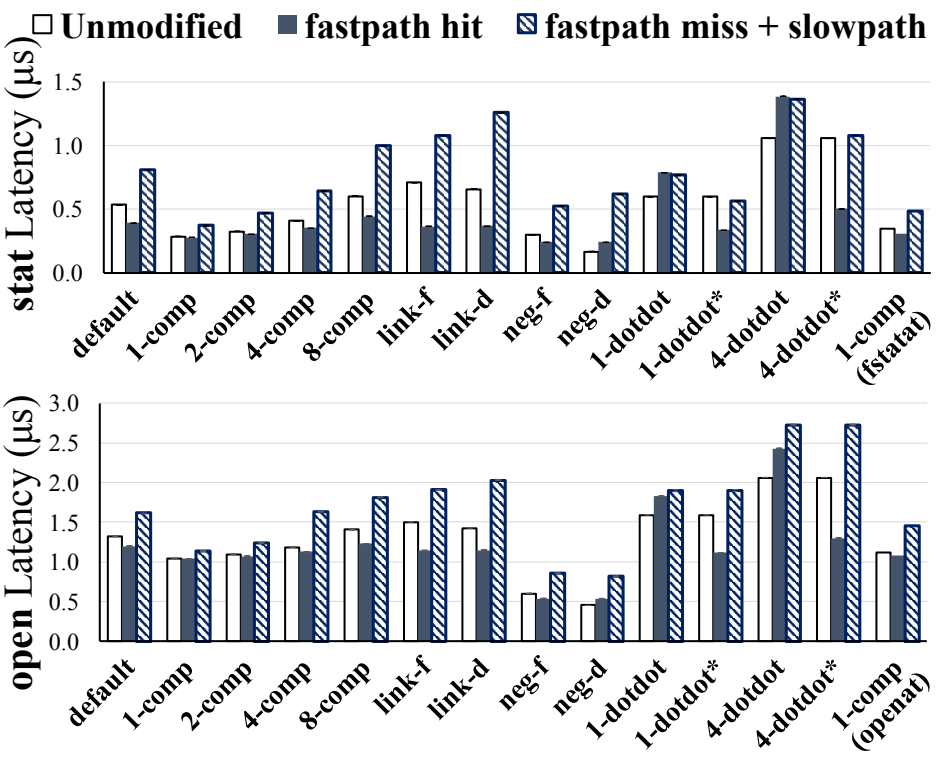
Micro-benchmarks. We use an extended LMBench 2.5 UNIX microbenchmark suite [29] to evaluate latency of path lookup at the system call level. Figure 6 shows the latency to `stat` and `open` sample paths with various characteristics, including varying lengths, symbolic links, parent (dot dot) directories, and files that are not found.

The primary trend we observe is that, as paths have more components, the relative gain for our optimization increases. For a single component file, `stat` gains 3% and open is equivalent to baseline Linux. For longer paths, the gain increases up to 26% and 12%, respectively.

To evaluate the worst case, we include a set of bars, labeled "fastpath miss + slowpath", which exercise the fast path code, but the kernel is configured to always miss in the PCC. This simulates the full cost of executing the optimized fastpath unsuccessfully, and then walking the O(n) slowpath in the cache. This case does not miss all the way to the low-level file system. The overhead typically ranges from 12–93%, except for path neg-d. In the case of neg-d, the first component is missing, and a component-at-a-time walk would stop sooner than a direct lookup. In general, the neg-d case would be mitigated by deep negative dentries. In practice, these overheads would only be observed for compulsory misses in the dcache, or by an application that exhibits an extreme lack of locality.

We next compare the costs of default Linux parent ("dot dot") semantics to Plan 9's lexical semantics. Enforcing Linux semantics for a path with parent references causes our optimizations to perform roughly 31% worse than unmodified Linux, as this requires an extra lookup per parent. Lexical path semantics, on the other hand, allow our optimization to continue using a single lookup, improving performance by 43–52%. Lexical path semantics have an independent benefit, and could reduce the number of components to walk in a lookup in unmodifed Linux. Although this difference is large, our test applications do not heavily use parent directory pointers, and are not sensitive to this difference.

Caching the resolution of a symbolic link improves performance for paths link-f and link-d by 44% and 48%, respectively. This improvement is insensitive to where in the path



(* represents Plan 9's lexical parent semantics)

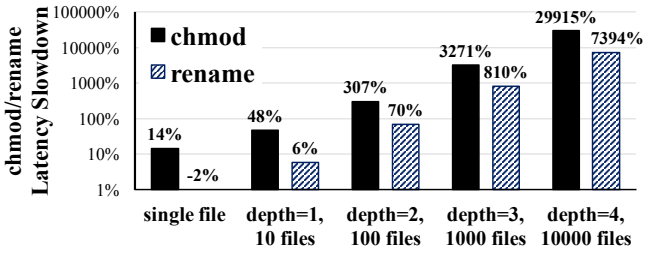
default: `/usr/include/gcc-x86_64-linux-gnu/sys/types.h`
1-comp: `FFF` **2-comp:** `XXX/FFF` **4-comp:** `XXX/YYY/ZZZ/FFF`
8-comp: `XXX/YYY/ZZZ/AAA/BBB/CCC/DDD/FFF`
link-f: `XXX/YYY/ZZZ/LLL -> FFF`
link-d: `LLL/YYY/ZZZ/FFF -> XXX/YYY/ZZZ/FFF`
neg-f: `XXX/YYY/ZZZ/NNN (NNN not found)`
neg-d: `NNN/XXX/YYY/FFF (NNN not found)`
1-dotdot: `XXX/../FFF` **4-dotdot:** `XXX/YYY/../../AAA/BBB/../../FFF`

Figure 6: System call `stat` and `open` latency for micro-benchmark (`lat_syscall` in `LMBench`), based on different path patterns. We include a synthetic evaluation of always missing on the fastpath and falling back to the slowpath, and a comparison with Plan 9's lexical parent semantics, where appropriate. Lower is better.

the link occurs, as both link-f and link-d walk the same number of components (link-d maps "LLL" onto "XXX").

For files that do not exist (negative dentries), we see comparable improvements to paths that are present. The one exception is long paths that don't exist under a directory early in the path. We believe this case is rare, as applications generally walk a directory tree top-down, rather than jumping several levels into a non-existent directory. In this situation (path neg-d), baseline Linux would stop processing the path faster than our optimization can hash the entire path, even with caching deep negative dentries. Nonetheless, deep negative dentries are an important optimization: without them, stat of path neg-d would be 113% worse and open would be 43% worse than unmodified Linux, versus 38% and 16% slower with deep negative dentries.

Linux also includes `*at()` system call variants, which operate under a working directory—typically using only a single component. Commensurate with the results above, `fstatat()` benefits from our optimizations by 12% for a single path component, and `openat()` is 4% faster than unmodified Linux. Some applications use multiple-component names in conjunction with an `*at` call; in these cases, the benefit of our optimization is proportional to the path length.



	µs	+/-	µs	+/-	µs	+/-	µs	+/-	µs	+/-
chmod	1.86	.00	1.60	.00	4.37	.00	36.38	.01	323	.02
rename	3.73	.00	4.68	.00	7.51	.00	40.14	.01	330	.05

Figure 7: `chmod` / `rename` latency in directories of various depths and sizes. Lower is better.

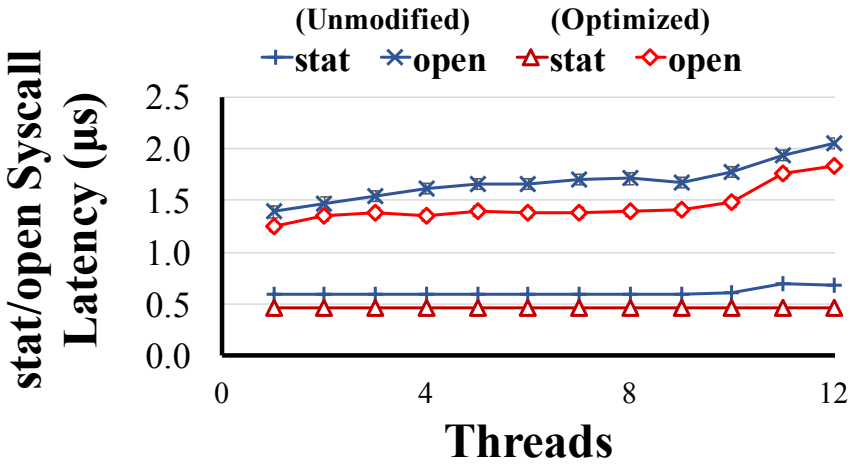


Figure 8: Latency of stat/open (of the same path), as more threads execute in parallel. Lower is better.

To evaluate the overhead of updating directory permissions and changing the directory structure, we measure `chmod` and `rename` latency. In our solution, the main factor influencing these overheads are the number of children in the cache (directory children out-of-cache do not affect performance). Figure 7 presents performance of `chmod` and `rename` on directories with different depths and directory sizes. In general, the cost of a rename or chmod increases dramatically with the number of children, whereas baseline Linux and ext4 make these constant-time operations. Even with 10,000 children all in cache, the worst-case latency is around 330 μs. As a point of reference, the Linux 3.19 source tree includes 51,562 files and directories. Initial feedback from several Linux file system maintainers indicate that this trade would be acceptable to improve lookup performance [15].

Space Overhead. Our prototype increases the size of a dentry from 192 bytes to 280 bytes. Our design also introduces a per-credential PCC of size 64 KB, and a second, global hash table (the DLHT), which includes 2^{16} buckets. Because Linux does not place any hard limits on dcache size, except extreme under memory pressure, it is hard to normalize execution time to account for the space cost. On a typical system, the dcache is tens to hundreds of MB; increasing this by 50% is likely within an acceptable fraction of total system memory. Alternatively, if one were to bound the total dcache size, this induces a trade-off between faster hits and fewer hits. We leave exploration of these trade-offs for future work.

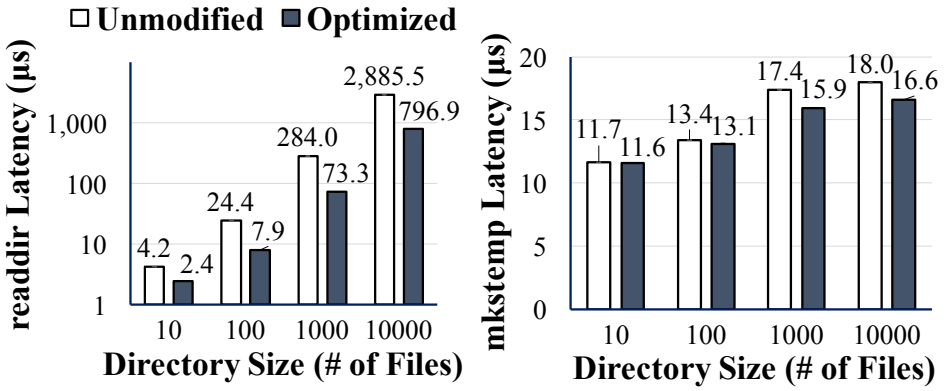


Figure 9: Latency in logscale for `readdir` function calls, and latency in microsecond for `mkstemp` function calls, on directories with different sizes. Lower is better.

Applications	Path Stats		Unmodified kernel				Optimized kernel		
	l	#	s	+/-	hit%	neg%	s	+/-	Gain
`find -name`	39	1	.055	.000	100.0	.18	.044	.000	19.2 %
`tar xzf linux.tar.gz`	22	3	4.039	.024	84.2	.06	4.038	.010	.05 %
`rm -r linux src`	24	3	.607	.008	100.0	.01	.621	.020	-2.32 %
`make linux src`	29	4	868.079	.647	91.2	17.84	868.726	.892	-.07 %
`make -j12 linux src`	29	4	102.958	.597	92.9	20.03	103.308	.288	-.34 %
`du -s linux src`	10	1	.070	.000	100.0	.01	.061	.012	12.65 %
`updatedb -U usr`	3	1	.011	.000	99.9	.00	.008	.000	29.12 %
`git status linux src`	16	4	.176	.000	100.0	.05	.168	.000	4.26 %
`git diff linux src`	16	4	.066	.000	100.0	1.49	.060	.000	9.89 %

Table 1: Execution time and path statistics of real-world applications bounded by directory cache lookup latency. Warm cache case. Hit rate and negative dentry rate are also included. The average path length in bytes (l) and components (#) are presented in the first two columns. Lower is better.

Scalability. Figure 8 shows the latency of a `stat/open` on the same path as more threads execute on the system. The read side of a lookup is already linearly scalable on Linux, and our optimizations do not disrupt this trend—only improve the latency. The `rename` system call introduces significant contention, and is less scalable in baseline Linux. For instance, a single-file, single-core rename takes 13µs on our test system running unmodified Linux; at 12 cores and different paths, the average latency jumps to 131µs for our optimized kernel, these numbers are 18 and 118µs, respectively, indicating that our optimizations do not make this situation worse for renaming a file. As measured in Figure 7, our optimizations do add overhead to renaming a large directory, which would likely exacerbate this situation.

6.2 Caching Directory Completeness

Figure 9 shows the latency of a `readdir` microbenchmark with varying directory sizes. The ability to cache `readdir` results improves performance by 46–74%. Caching helps more as directories get larger. OpenSolaris comments indicate that this idea was only beneficial in UFS for directories with at least 1,024 entries [2]. Our result indicates that there is benefit even for directories with as few as 10 children.

Figure 9 also shows the latency of creating a secure, randomly-named file in directories of varying size. We measure from 1–8% improvement for the `mkstemp` library. Although most applications' execution times are not dominated by secure file creation, it is a common task for many applications, and of low marginal cost.

[2] see line 119 of `fs/ufs/ufs_dir.c` in OpenSolaris, latest version of frozen branch onnv-gate.

App	Unmodified kernel				Optimized kernel		
	s	+/-	hit%	neg%	s	+/-	Gain
`find`	1.39	.01	38	1	1.35	.02	3.1 %
`tar`	4.00	.10	85	0	3.98	.04	.5 %
`rm -r`	1.81	.05	83	1	1.84	.06	-1.4 %
`make`	885.33	.31	100	45	883.88	2.03	.2 %
`make -j12`	114.51	.60	100	47	114.54	.89	.2 %
`du -s`	1.49	.01	6	0	1.46	.02	2.3 %
`updatedb`	.73	.01	34	0	.74	.01	-2.1 %
`git status`	4.13	.03	62	2	4.11	.03	.7 %
`git diff`	.84	.01	61	0	.86	.01	-14.0 %

Table 2: Execution time, hit rate, and negative dentry rate for real-world applications bounded by directory cache lookup latency with a cold cache. Lower is better.

6.3 Applications

Command-Line Applications. The improvement applications see from faster lookup is, of course, proportional to the fraction of runtime spent issuing path-based system calls as well as the amount of time listing directories. We measure the performance of a range of commonly-used applications. In most cases, these applications benefit substantially from these optimizations; in the worst case, the performance harm is minimal. The applications we use for benchmarking include:

- `find`: search for a file name in the Linux source directory.
- `tar xzf`: decompress and unpack the Linux source.
- `rm -r`: remove the Linux source tree.
- `make` and `make -j12`: compile the Linux kernel.
- `du -s`: Recursively list directory size in Linux source.
- `updatedb`: rebuild database of canonical paths for commonly searched file names in /usr from a clean `debootstrap`.
- `git status` and `git diff`: display status and unstaged changes in a cloned Linux kernel git repository.

For each application we test, we evaluate the performance in both cases of a warm cache (Table 1) and a a cold cache (Table 2). To warm the cache, we run the experiment once and drop the first run. For the warm cache tests, we also provide statistics on the path characteristics of each application.

Perhaps unsurprisingly, metadata-intensive workloads benefit the most from our optimizations, such as `find` and `updatedb`, as high as 29% faster. Note that `find`, `updatedb`, and `du` use the `*at()` APIs exclusively, and all paths are single-component; these gains are attributable to both improvements to lookup and directory completeness caching.

We note that the performance of directory-search workloads is sensitive to the size of PCC; when we run `updatedb` on a directory tree that is twice as large as the PCC, the gain drops from 29% to 16.5%. This is because an increased fraction of the first lookup in a newly-visited directory will have to take the slowpath. Our prototype has a statically-set PCC size and we evaluate with a PCC sufficiently large to cache most of the relevant directories in the warm cache experiments. We expect that a production system would dynamically resize the PCC up to a maximum working set; we leave investigating an appropriate policy to decide when to grow the PCC versus evict entries for future work.

The application that is improved primarily by our hit optimization is `git`, which shows a gain of 4–9.9%. Cases that are dominated by other computations, such as a Linux compile, show minimal ($\leq 2.3\%$ slowdown). In the cold cache cases, all gains or losses are roughly within experimental noise, indicating that these optimizations are unlikely to do harm to applications running on a cold system. In general, these results affirm that common Linux applications will not be harmed by the trade-offs underlying our optimizations, and can benefit substantially.

Table 1 also indicates statistics about these workloads on unmodified Linux. In general, each path component tends to be roughly 8 characters, and *at-based application gener-

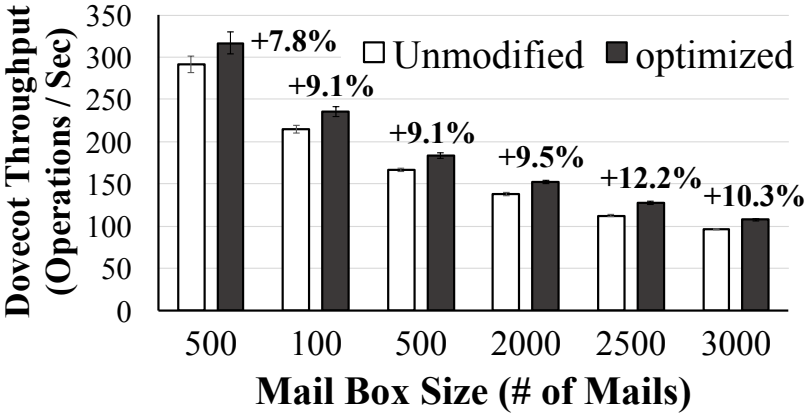


Figure 10: Throughput for marking and unmarking mail on the Dovecot IMAP server. Higher is better.

# of files	Unmodified kernel		Optimized kernel		
	Req/s	+/-	Req/s	+/-	Gain
10	27,638.23	43.31	31,491.98	55.24	12.24 %
10^2	7,423.86	11.81	7,934.07	12.76	6.43 %
10^3	1,017.02	0.58	1,081.02	0.38	5.92 %
10^4	99.03	0.11	110.14	0.10	10.09 %

Table 3: Throughput of downloading generated directory listing pages from an `Apache` server. Higher is better.

ally lookup single-component paths, whereas other applications typically walk 3–4 components. Our statistics also indicate that, with a warm cache, these applications should see 84–100% hit rate in the cache, so optimizing the hit path is essential to performance. Finally, make is the only application with a significant fraction of negative dentries (roughly 20%), which is to be expected, since it is creating new binary files.

Server Applications. An example of software that frequently uses `readdir` is an IMAP mail server using the MailDir storage format. We exercise the Dovecot IMAP server by creating 10 mailboxes for a client. We use a client script to randomly select messages in different mailboxes and mark them as read, flagged, or unflagged. Internally, marking a mail causes a file to be renamed, and the directory to be re-read. To eliminate network latency, we run network tests on the localhost; in a practical deployment, network latency may mask these improvements to the client, but the load of the server will still be reduced.

Figure 10 shows the throughput for the Dovecot mail server on both kernels; improvements range from 7.8–12.2%. Commensurate with the `readdir` microbenchmark, larger directories generally see larger improvement, plateauing at a 10% gain. We similarly exercise the Apache web server's ability to generate a file listing using the Apache benchmark (Table 3). These pages are not cached by Apache, but generated dynamically for each request. This workload also demonstrates improvements in throughput from 6–12%. Overall, these results show that the readdir caching strategy can reduce server load or improve server throughput for directory-bound workloads.

6.4 Code Changes

In order estimate the difficulty of adoption, Table 4 lists the lines of code changed in our Linux prototype. The vast majority of the changes required (about 1,000 LoC) are hooks localized to the dcache itself (`dcache.c` and `namei.c`); most of these optimizations are in a separate set of files totaling about 2,400 LoC. Also, the low-level file systems we tested did not require *any* changes to use our modified directory caches. The main impact on other subsystems was actually to the LSMs, which required some changes to manage PCCs correctly. Thus, the burden of adoption for other kernel subsystems is very minor.

Source files	Original (LoC)	Patched/Added (LoC)	
New source files & headers		2,358	
`fs/namei.c`	3,048	425	13.9 %
`fs/dcache.c`	1,997	142	7.1 %
Other VFS sources	3,859	98	2.5 %
Other VFS headers	2,431	218	9.0 %
Security Modules (SELinux, etc)	20,479	76	0.0 %

Table 4: *Lines-of-code* (LoC) in Linux changed, as measured by `sloccount` [46].

6.5 Discussion and Future Work

If one were willing to sacrifice complete backward compatibility to maximize lookup performance, the primary opportunity for improvement may actually be in designing a simpler *interface* for path-based calls. As the evaluation above shows, there are several Linux/POSIX features that are unduly expensive to support in this design. For instance, implementing Plan 9-style lexical path semantics can significantly improve look up for paths with a "dot dot". Similarly, working directory semantics require a slowpath traversal. Arguably, these are points where a particular implementation choice has "leaked" into the interface specification, and these idiosyncracies constrain the choice of supporting data structure. We recommend an interface that is as simple and stateless as possible; this recommendation is in line with other recommendations for scalability [10].

Linux statically selects the number of buckets in the hash table (262,144 by default). If this number is not selected well, or the demand changes over time, space will be wasted or bucket chains will get longer, harming lookup performance. On our test systems, 58% of buckets were empty, 34% had one item, 7% had 2 items, and 1% had 3–10 dentries, indicating an opportunity to improve lookup time and space usage. A number of high-performance hash tables have been developed in recent years that impose a constant bound on the search time as well as on wasted space [18, 23, 32, 43].

7. Related Work

Most related work on improving directory cache effectiveness targets two orthogonal problems: reducing the miss latency and prefetching entries. Most similar to our optimization to memoize prefix check results, SQL Server caches the result of recent access control checks for objects [30].

Reducing Miss Latency. One related strategy to reduce miss latency is to pass all components to be looked up at once to the low-level file system, essentially creating a prefetching hint. Several network file systems have observed that component-at-a-time lookup generates one round-trip message per component, and that a more efficient strategy would pass all components under a mount point in one message to the server for lookup [14, 45]. A similar argument applies to local file systems, where a metadata index may be more efficiently fetched from disk by knowing the complete lookup target [22, 36]. As a result, this division of labor is adopted by Windows NT and Solaris [25, 36]. One caveat is that, when not taken as a prefetching "hint", this can push substantial VFS functionality into each low-level file system, such as handling redirection at mount points, symbolic links, and permission checking. Chen et al. note that pushing permission checks from the VFS layer down to individual file systems is a substantial source of difficult-to-prevent kernel bugs in Linux [9]. In contrast, this project caches the result of previous prefix checks over paths already in memory to reduce hit latency, rather than using the full path as a prefetching hint.

Another ubiquitous latency-reduction strategy is persistently storing metadata in a hash table. In order to reduce network traffic, several distributed file systems [5, 49, 51], clustered environments [20, 48], and cloud-based applications [44] have used metadata hashing to deterministically map metadata to a node, eliminating the need for a directory service. The Direct Lookup File System (DLFS) [22] essentially organizes the entire disk into a hash table, keyed by path within the file system, in order to look up a file with only

one I/O. Organizing a disk as a hash table introduces some challenges, such as converting a directory rename into a deep recursive copy of data and metadata. DLFS solves the prefix check problem by representing parent permissions as a closed form expression; this approach essentially hard-codes traditional Unix discretionary access control, and cannot easily extend to Linux Security Modules [47]. An important insight of our work is that full path hashing in memory, but not on disk, can realize similar performance gains, but without these usability problems, such as deep directory copies [22] on a rename or error-prone heuristics to update child directory permissions [41].

VFS Cache Prefetching. Several file systems optimize the case where a `readdir` is followed by `stat` to access metadata of subdirectories, such as with the `ls -l` command [3, 22, 42]. When a directory read is requested, these low-level file systems speculatively read the file inodes, which are often in relatively close disk sectors, into a private memory cache, from which subsequent lookups or stat requests are serviced. Similarly, the NFS version 2 protocol includes a `READDIRPLUS` operation, which requests both the directory contents and attributes of all children in one message round trip [7]. These file systems must implement their own heuristics to manage this cache. Prefetching is orthogonal to our work, which more effectively caches what has already been requested from the low-level file system.

8. Conclusion

This paper presents a directory cache design that efficiently maps file paths to in-memory data structures in an OS kernel. Our design decomposes the directory cache into separate caches for permission checks and path indices, enabling single-step path lookup, as well as facilitating new optimizations based on signatures and caching symbolic link resolution. For applications that frequently interact with the file system directory tree, these optimizations can improve performance by up to 29%. Our optimizations maintain compatibility with a range of applications and kernel extensions, making them suitable for practical deployment.

Acknowledgments

We thank the anonymous reviewers, Michael Bender, Rob Johnson, David Wagner, and our shepherd Michael Kaminsky for insightful comments on this work. Imran Brown and William Jannen contributed to the prototype and evaluation. This research is supported in part by NSF grants CNS-1149229, CNS-1161541, CNS-1228839, CNS-1405641, and CNS-1408695.

References

[1] AppArmor. `http://wiki.apparmor.net/`.

[2] Daniel J. Bernstein. Using maildir format. `http://cr.yp.to/proto/maildir.html`, 1995.

[3] Tim Bisson, Yuvraj Patel, and Shankar Pasupathy. Designing a fast file system crawler with incremental differencing. *ACM SIGOPS Operating Systems Review*, Dec 2012.

[4] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 3rd edition, 2005.

[5] Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Lan Xue. Efficient metadata management in large distributed storage systems. In *IEEE Conference on Mass Storage Systems and Technologies (MSST)*, Washington, DC, USA, 2003.

[6] The *bsdstats project. `www.bsdstats.org`.

[7] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, June 1995.

[8] CERT Secure Coding. FIO21-C. Do not create temporary files in shared directories.

[9] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Asia-Pacific Workshop on Systems*, pages 5:1–5:5, 2011.

[10] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 1–17, 2013.

[11] Jonathan Corbet. JLS: Increasing VFS scalability. *LWN*, November 2009. `http://lwn.net/Articles/360199/`.

[12] Jonathan Corbet. Dcache scalability and RCU-walk. *Linux Weekly News*, 2010.

[13] Biplob Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proceedings of the USENIX Annual Technical Conference*, pages 16–16, 2010.

[14] Dan Duchamp. Optimistic lookup of whole NFS paths in a single operation. In *Proceedings of the USENIX Summer Technical Conference*, 1994.

[15] Rik Farrow. Linux FAST'15 summary. *;login: Magazine*, 40(3):90–95, June 2015.

[16] Andriy Gapon. Complexity of FreeBSD VFS using ZFS as an example. Part 1. `https://clusterhq.com/blog/complexity-freebsd-vfs-using-zfs-example-part-1-2/`, 2014.

[17] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 71–83, 2011.

[18] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the International Conference on Distributed Computing (DISC)*, pages 350–364, 2008.

[19] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[20] Jharrod LaFon, Satyajayant Misra, and Jon Bringhurst. On distributed file tree walk of parallel file systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[21] Daniel Lemire and Owen Kaser. Strongly universal string hashing is fast. *The Computer Journal*, page bxt070, 2013.

[22] Paul Hermann Lensing, Toni Cortes, and André Brinkmann. Direct lookup and hash-based metadata placement for local file systems. In *ACM International Systems and Storage Conference (SYSTOR)*, 2013.

[23] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 27:1–27:14, 2014.

[24] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the USENIX Annual Technical Conference*, 2001.

[25] Richard McDougall and Jim Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, Second Edition*. Sun Microsystems Press, 2008.

[26] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy Update Techniques in Operating System Kernels*. PhD thesis, 2004.

[27] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. Scaling dcache with RCU. *Linux Journal*.

[28] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2005.

[29] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 23–23, 1996.

[30] Microsoft. Description of the "access check cache bucket count" and "access check cache quota" options that are available in the sp_configure stored procedure. `https://support.microsoft.com/en-us/kb/955644`.

[31] Danilov Nikita. Design and implementation of xnu port of lustre client file system. Technical report, 2005.

[32] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, pages 122–144.

[33] Rob Pike. Lexical File Names in Plan 9, or, Getting Dot-dot Right. In *Proceedings of the USENIX Annual Technical Conference*, pages 7–7, 2000.

[34] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 89–101, 2002.

[35] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Communication ACM*, July 1974.

[36] M. Russinovich and D. Solomon. *Windows Internals*. Microsoft Press, 2009.

[37] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation or the sun network filesystem. In *Proceedings of the USENIX Annual Technical Conference*, 1985.

[38] S. Shepler, Storspeed Inc., M. Eisler, D. Noveck, and NetApp. Network file system (NFS) version 4 minor version 1 protocol. RFC 5661, Jan 2010.

[39] Amit Singh. *Mac OS X Internals—A Systems Approach*. Addison-Wesley, 2006.

[40] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. idedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 24–24, 2012.

[41] Michael M. Swift, Peter Brundrett, Cliff Van Dyke, Praerit Garg, Anne Hopkins, Shannon Chan, Mario Goertzel, and Gregory Jensenworth. Improving the granularity of access control in Windows NT. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2001.

[42] Douglas Thain and Christopher Moretti. Efficient access to many samall files in a filesystem for grid computing. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, Washington, DC, USA, 2007. IEEE Computer Society.

[43] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the USENIX Annual Technical Conference*, pages 11–11, 2011.

[44] Yixue Wang and Haitao Lv. Efficient metadata management in cloud computing. In *ICCSN*, pages 514–519, 2011.

[45] Brent Welch. A comparison of three distributed file system architectures: Vnode, sprite, and plan 9. *Computer System*, March 1994.

[46] David Wheeler. Sloccount, 2009.

[47] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. K. Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security Symposium*, 2002.

[48] Jing Xing, Jin Xiong, Ninghui Sun, and Jie Ma. Adaptive and scalable metadata management to support a trillion files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 2009.

[49] Quan Zhang, Dan Feng, and Fang Wang. Metadata performance optimization in distributed file system. In *ICIS*, Washington, DC, USA, 2012.

[50] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 18:1–18:14, 2008.

[51] Yifeng Zhu, Hong Jiang, Jun Wang, and Feng Xian. HBA: Distributed metadata management for large cluster-based storage systems. *IEEE Trans. Parallel Distrib. Syst.*, pages 750–763, 2008.