Hash Trees (htree) in ext4 File System

Technical Documentation

July 19, 2025

Abstract

This document provides a comprehensive analysis of Hash Trees (htree) implementation in the ext4 file system. Hash trees represent a significant optimization for directory indexing, addressing the O(n) linear search limitation of traditional directory structures. This paper examines the architectural design, implementation details, algorithmic complexity, and performance characteristics of htree in ext4, demonstrating how hash-based indexing achieves O(log n) directory operations while maintaining backward compatibility.

Contents

1 Introduction

The ext4 file system implements Hash Trees (htree) as a directory indexing mechanism to overcome the performance limitations of linear directory searches. Traditional directory implementations store entries sequentially, resulting in O(n) lookup time complexity. Hash trees provide a hierarchical hash-based indexing structure that reduces directory operation complexity to O(log n), making ext4 suitable for directories containing thousands of entries.

1.1 Problem Statement

Important Note

Large directories in traditional file systems suffer from:

- **Linear search complexity O(n)** for file lookups
- Poor cache locality for large directory scans
- Scalability issues with increasing directory size
- Performance degradation in directory-intensive workloads

1.2 Solution Overview

Key Concept

Hash trees address these limitations through:

- **Hash-based partitioning** of directory entries
- **Hierarchical tree structure** for logarithmic access
- Efficient space utilization with configurable branching factors
- Backward compatibility with linear directory format

2 Hash Tree Architecture

2.1 Structural Components

Definition

The hash tree consists of two primary node types:
Internal Nodes: Contain hash-based routing information and pointers to child nodes. Each internal node maintains:

- Array of child pointers (typically 4-way branching)
- Level information for hash bit extraction
- Node type identifier

Leaf Nodes: Store actual directory entries with configurable capacity limits:

- Array of directory entries (filename, inode pairs)
- Entry count tracking
- Overflow handling mechanisms

2.2 Hash Function Design

Example

The hash tree employs a deterministic hash function mapping filenames to 32-bit hash values:

$$H(filename) = \text{hash\_function}(filename) \bmod 2^{32} \tag{1}$$

Hash bit extraction for level-based routing:

$$\text{child\_index} = \frac{H(filename) \gg (\text{level} \times \text{bits\_per\_level})}{2^{\text{bits\_per\_level}}} \tag{2}$$

Where:

- `bits_per_level` $= 2$ (4-way branching)
- `level` $=$ current tree depth
- `child_index` $\in [0, 3]$ for 4-way branching

3 Implementation Details

3.1 Data Structures

Listing 1: Core Data Structures

```
struct DirectoryEntry {
    std::string filename;
    uint64_t inode;
};

struct Node {
    bool is_leaf;
    int level;
};

struct InternalNode : Node {
    std::vector<std::unique_ptr<Node>> children;
};

struct LeafNode : Node {
    std::vector<DirectoryEntry> entries;
};
```

3.2 Configuration Parameters

primaryblue!20 **Parameter**	**Value**	**Description**
lightgray MAX_LEAF_ENTRIES	3-8	Maximum entries per leaf node
HASH_BITS	32	Hash value bit width
lightgray BITS_PER_LEVEL	2	Bits extracted per tree level
CHILDREN_PER_NODE	4	Branching factor ($2^{\text{BITS\_PER\_LEVEL}}$)

Table 1: Hash Tree Configuration Parameters

3.3 Tree Operations

3.3.1 Insertion Algorithm

Algorithm 1 Hash Tree Insertion

Require: filename, inode

```
hash ← hash_filename(filename)
node ← find_leaf(hash)
if entry exists in leaf then
    update existing entry
else
    add new entry to leaf
    if leaf size > MAX_LEAF_ENTRIES then
        split_leaf(leaf)
    end if
end if
```

3.3.2 Lookup Algorithm

Algorithm 2 Hash Tree Lookup

Require: filename

```
hash ← hash_filename(filename)
leaf ← find_leaf(hash)
for each entry in leaf do
    if entry.filename == filename then
        return entry.inode
    end if
end for
return NOT_FOUND
```

3.3.3 Leaf Finding Traversal

Listing 2: Tree Traversal Implementation

```cpp
LeafNode* find_leaf(const std::string& filename) {
    uint32_t hash = hash_filename(filename);
    Node* current = root.get();

    while (!current->is_leaf) {
        InternalNode* internal = static_cast<InternalNode*>(current);
        int index = get_hash_bits(hash, current->level);

        if (!internal->children[index]) {
            internal->children[index] =
                std::make_unique<LeafNode>(current->level + 1);
        }
        current = internal->children[index].get();
    }

    return static_cast<LeafNode*>(current);
}
```

4 Performance Analysis

4.1 Time Complexity

Operation	Hash Tree	Linear Directory
Lookup	O(log n)	O(n)
Insertion	O(log n)	O(1) append, O(n) search
Deletion	O(log n)	O(n)
Enumeration	O(n)	O(n)

Table 2: Time Complexity Comparison

4.2 Space Complexity

Hash trees introduce overhead through:

- Internal node storage: $O(\text{tree\_height} \times \text{branching\_factor})$
- Hash computation and storage
- Pointer overhead for tree navigation

Total space complexity: $O(n + \text{tree\_height} \times \text{branching\_factor})$

4.3 Performance Characteristics

Advantages

- **Logarithmic lookup time** for large directories
- **Excellent scalability** with directory size
- Cache-friendly access patterns
- Balanced tree structure through hash distribution

Limitations

- **No lexicographic ordering** maintained
- Hash collision handling complexity
- Additional memory overhead
- Tree rebalancing during splits

5 Hash Distribution Analysis

Important Note

The effectiveness of hash trees depends on uniform hash distribution. Poor hash functions can lead to:

- **Unbalanced tree structures**
- Hash collision clustering
- Degraded performance approaching O(n)

5.1 Hash Quality Metrics

Example

Hash function quality evaluation:

$$\text{Distribution\_Quality} = \frac{\min(\text{bucket\_sizes})}{\max(\text{bucket\_sizes})} \tag{3}$$

Target: Ideal hash distribution achieves Distribution_Quality ≈ 1.0.

6 Implementation Considerations

6.1 Backward Compatibility

ext4 maintains compatibility with non-htree directories through:

- Feature flag detection
- Graceful fallback to linear search
- Transparent conversion mechanisms

6.2 Concurrency Control

Hash tree operations require synchronization for:

- Concurrent read/write access
- Tree structure modifications
- Leaf node splitting operations

6.3 Error Handling

Robust error handling addresses:

- Hash collision resolution
- Memory allocation failures
- Tree corruption detection
- Recovery mechanisms

7 Practical Applications

7.1 Use Cases

Key Concept

Hash trees excel in scenarios with:

- **Large directory sizes** (>1000 entries)
- Frequent file lookup operations
- Random access patterns
- Directory-intensive applications

7.2 Performance Benchmarks

Example

Typical performance improvements:

- **10x-100x faster lookups** for directories >10,000 entries
- Reduced I/O operations for large directory scans
- Improved cache utilization
- Better scalability under load

8 Conclusion

Summary

Hash trees represent a fundamental advancement in file system directory indexing, transforming O(n) linear operations into **O(log n) hierarchical lookups**. The ext4 implementation demonstrates the practical benefits of hash-based indexing while maintaining backward compatibility and robust error handling.

Key Achievements

- **Logarithmic time complexity** for directory operations
- **Excellent scalability** for large directories
- Efficient space utilization
- Production-ready implementation in ext4

The hash tree architecture provides a solid foundation for modern file system performance requirements, enabling efficient handling of directories containing millions of entries while maintaining the simplicity and reliability expected from production file systems.