A Directory Index for Ext2

Daniel Phillips

18th September 2001

Abstract

In recent years, filesystem designers have tended to favor the use of BTrees and variants thereof to eliminate directory performance bottlenecks. The idea of adding BTree directory indexing to Linux's Ext2 filesystem has been much discussed but never implemented, more probably due to an aversion to complexity than any laziness on the part of developers. Recently, I conceived and implemented a new kind of indexing technique using hashed keys held in a uniform-depth tree. I gave this structure the name "HTree". Compared to a BTree, the HTree has a considerable simpler implementation but offers similar performance. Furthermore, it turned out to be possible to achieve not only backward compatibility with existing Ext2 filesystems, but forward compatibility with earlier versions of Linux as well.

1 Introduction

The motivation for the work reported in this paper is to provide Linux's native filesystem, Ext2, with directory indexing, one of the standard features of a modern filesystem that it has lacked to date. Without some form of directory indexing, there is a practical limit of a few thousand files per directory before quadratic performance characteristics become visible. To date, an applications such as a mail transfer agent that needs to manipulate large numbers of files has had to resort to some strategy such as structuring the set of files as a directory tree, where each directory contains no more files than Ext2 can handle efficiently. Needless to say, this is clumsy and inefficient.

The design of a directory indexing scheme to be retrofitted onto a filesystem in widespread production use presents some interesting and unique problems. First among these is the need to maintain backward compatibility with existing filesystems. If users are forced to reconstruct their partitions then much of the convenience is lost and they might as well consider putting in a little more work and adopting an entirely new filesystem. Perhaps more importantly, Ext2 has proved to be more reliable than any of new generation of filesystems available on Linux, in part due to its maturity, but also no doubt partly due to its simplicity. Ext2 has proved to be competitive with any of the new filesystems in manipulating the relatively small files and filesets (by today's standards) that are common on typical installations. Ext2's suite of filesystem utilities which includes e2fsck for performing filesystem checking and recovery, and debugfs for performing manual filesystem examination and repair, is particularly mature and capable. Finally, Ext2, being Linux's native filesystem, almost by definition is the filesystem that gives the broadest range of support for Linux's many features.

The simplicity of Ext2's design places a special burden on the designer of a directory indexing extension to strive for a similar degree of simplicity. The idea of adding BTree directory indexing to Linux's Ext2 filesystem has been much discussed but never implemented, more probably due to an aversion to complexity than any laziness on the part of developers. Unfortunately, a survey of implementations of directory indexing strategies in existing "big iron" filesystems shows that the directory indexing code by itself, is comparable in size to all of Ext2. So a simple port of such code would not achieve the desired results, and that would still leave open the question of how to make the new indexing facility backward compatible with existing Ext2 volumes.

In the event, with some luck, determination and expert advice, I was able to come up with a design that can be implemented in a few hundred lines of code and which offers not only superior performance with very large directories, but performance that is at least as good as traditional Ext2 with small directories, and not only backward compatibility with existing Ext2 volumes, but forward compatibility with pre-existing versions of Linux as well. In fact, the new design uses the same directory block format as Ext2 has traditionally used, and to earlier versions of Linux, an indexed directory appears to be a perfectly valid directory in every detail.

2 Background

The earliest versions of Linux used the Minix filesystem.[1] Recognizing its limitations, Remy Card designed and implemented the Extended Filesystem, BSD's UFS. That design was improved and became the Second Extended Filesystem, or Ext2.

Ext2's design is characterized by extreme simplicity, almost to a fault. For example, certain features that were planned for Ext2 were never implemented, such as fragment support modeled on UFS, and BTree directory indexing. Instead of adding features, Ext2 maintainers have tended to concentrate on making the existing features work better. Today Ext2 is well known for its high degree of stability, and ruggedness in the face of abuse. To some extent, the perhaps subconscious philosophy of minimalism can be credited for this. Nonetheless, as Linux evolves it encounters ever-rising expectations from its users, including those who have traditionally worked with "big iron" variants of Unix and tend to measure the worth of Linux by that yardstick. Pressure has increased to address those areas where Ext2 does not scale very well into enterprise-class applications.

Although there is a new crop of enterprise-class filesystems appearing in Linux this year - XFS and JFS, which were ported from SGI's Irix and IBM's OS/2 respectively - there exists a strong sentiment that Ext2 should retain the role of Linux's native filesystem, both for pragmatic reasons such as its feature set - by no accident well matched to Linux's requirements - and its stability. Perhaps there is also an element of pride, since if anything can be said to be the heart of Linux, it would be its filesystem, Ext2. Whatever the reason, motivation is strong to address the remaining weaknesses that separate Ext2 from a true enterprise-class filesystem. One of the most obvious is the lack of any kind of directory indexing: simple linear search is used, as it was in the earliest days of Unix.

For some common operations, a linear directory search imposes an $O(n^2)$ performance penalty where n is the number of files in a directory. With n in the thousands, the observed slowdown is very noticable. Typically, enterprise-class filesystems use some form of balanced tree indexing structure, which gives $O(Log(n))$ performance for individual directory operations, or $O(nLog(n))$ across all the files of a directory. Ext2 was expected eventually to adopt such a scheme and in fact some provision was made for it in internal structures, but none was ever implemented. That this was never done should be ascribed to an aversion to complexity rather than any laziness on the part of Ext2 maintainers. After all, Ext2 in its entirety consists of about 5,000 lines of code and an implementation of a standard BTree algorithm could easily double that.

At the Linux Storage Management Workshop last year in Miami, Ted Ts'o described to me some design ideas he had for a simplified BTree implementation. At the time, there still seemed to be unresolved design issues that could lead to significant complexity, so implementation work was not begun at that time. Some months later while describing Ted's ideas in a posting to a mailing list, a fundamental simplification occurred to me, namely that we could use logical addressing for the BTree instead of physical, thus avoiding any change to Ext's existing metadata structure. Thus inspired, I went on to discard the BTree approach, and invented a new type of indexing structure whose characteristics lie somewhere between those of a tree and a hash table. Since I was unable to find anything similar mentioned in the literature I took the liberty of giving this structure the name HTree, a hash-keyed uniform-depth index tree.

After a week of intense development, and with the assistance of my coworker Uli Luckas, I managed to produce a prototype implementation with enough functionality to perform initial benchmarks. The measured results, which Uli prepared for me in the form of a chart, were described in a post to the Linux Kernel mailing list that same day.[8] Aided by the lopsided relationship between $O(n^2)$ and $O(nLog(n))$complexity, I was able to show a spectacular 145-times improvemen[5]t against standard Ext2, for an test case which I had of course chosen carefully. Whatever my bias, the performance improvements in practice were real and measurable. Suddenly Linux's venerable Ext2 no longer seemed to be on the verge of extinction in the face of competition from a new crop of enterprise-class filesystems.

The original prototype achieved its performance gains using a degenerate index tree consisting of only a single block. In the following months I carried on further development, incorporating many suggestions from Andreas Dilger, Ted and others, to finalize the disk format and bring my prototype to the a stage where it could be tested in live production testing. At this time, I, together with members of the Ext3 development team (Stephen Tweedie, Andrew Morton and Ted Ts'o) am preparing to incorporate this feature into Ext3, which task should be completed by the time this paper is published.

3 Data Structures and Algorithms

HTree Data Structure

A flag bit in a directory's inode indicates whether a directory index is indexed or not. If this bit is set then the first directory block is to be interpreted as the root of an HTree index.

Given the design goal that HTree-indexed directories appear to preexisting versions of Linux as normal, unindexed directory files, the structure of an HTree every directory block is dictated by the traditional Ext2 directory block format. If this were not the case, then a volume with directories created with HTree indexes would appear to have garbage in directories if mounted by an older version of Linux. Fortunately, it is possible to place an empty directory entry record in a block which is actually an HTree index constructed so that the entire block appears to be free when interpreted as an Ext2 directory block, yet only the first 8 bytes are actually used. This leaves the remainder of the block free for HTree-specific structures.

The root of an HTree index is the first block of a directory file. The leaves of an HTree are normal Ext2 directory blocks, referenced by the root or indirectly through intermediate HTree index blocks. References within the directory file are by means of logical block offsets within the file. The possibility of using direct physical pointers was considered for reasons of efficiency, but abandoned due to the onerous requirement of incorporating special handling for such pointers in many parts of Ext2 outside the directory handling code. Luckily, it turned out that with Linux's recent change to logical indexing of file data, logical block pointers are no less efficient than physical ones.

An HTree uses hashes of names as keys, rather than the names themselves. Each hash key references not an individual directory entry, but a range of entries that are stored within a single leaf block. An HTree first level index block contains an array of index entries, each consisting of a hash keys and a logical pointer to the indexed block. Each hash key is the lower bound of a all the hash values in a leaf block and the entries are arranged in ascending order or hash key. Both hash keys and logical pointers are 32-bit quantities. The lowest bit of a hash key is used to flag the possibility of a hash collision, leaving 31 bits available for the hash itself.

The HTree root index block also contains an array of index entries in the same format as a first level index block. The pointers refer to index blocks rather than leaf blocks and the hash keys give the lower bounds for the hash keys of the referenced index block. The HTree root index also contains a short header, providing information such as the depth of the HTree and information oriented towards checking the HTree index integrity and such other functions as specifying which of several possible hash functions was used to create the directory.

Each index entry requires 8 bytes, so allowing for a 32 byte header, a single 4K index block can index up to 508 leaf blocks. Assuming that each leaf block can hold about 200 entries and each leaf block is 75% full, a single index block can index about 75,000 names. A second index level is required only for very large directories, which can accommodate somewhat more than 30 million entries. A third level would increase capacity to over 11 billion entries. Such a large number of directory entries is unlikely to be needed in the near future, so the current implementation provides a maximum of two levels in the index trees.

Hash Probe

The first step in any indexed directory operation is to read the index root, the first block of the directory file. Then a number of tests are performed against the header of the index root block in an attempt to eliminate any possibility of a corrupted index causing a program fault in the operating system kernel. It is intended that the detection of any inconsistency would cause the directory operation to revert to a linear search. In this way the user is given the best possible chance to access data on a corrupted volume. This mechanism also allows for certain changes to be made to the index structure in the future; for example, more than two levels might be allowed in the tree, while still allowing earlier versions of the directory index code to access the directory. Should such an inconsistent index be detected an error flag is set in the filesystem superblock so that appropriate warnings can be issued and the directory index can be automatically rebuilt by the fsck utility.

Next the hash value of the target name is computed, and from that a determination is made of which leaf block to search. The desired leaf block is the one whose hash range includes the hash of the target entry name. Since index entries are of fixed size and maintained in sorted order, a binary search is used here. The format of an index entry is the same whether it references a leaf block or an interior index node, so this step is simply repeated if the index tree has more than one level.

As the hash probe descends through index entries an access chain is created, for use by the lookup and creation operations described below. Finally, the target block is read.

Entry Lookup

Once a target leaf block has been obtained lookup proceeds exactly as without an index, i.e., by linearly searching through the entries in the block. If the target name is not found then there is still a possibility that the target could be found in the following leaf block due to a hash collision. In this case, the parent index is examined to see if the hash value of the successor leaf block hash has its low bit set, indicating that a hash collision does exit. If set, then the hash value of the target string is compared for equality to the successor block's hash value, less the collision bit. If it is the same then the successor leaf block is read, the access chain updated, and the search is repeated.

If the leaf block happens to be referenced by the final index entry of an index block then the successor hash value is obtained from the parent index block, which has already been read. If the possibility of a collision with the target exists then the successor index block will be read as a prelude to reading the successor leaf block.

Although it sounds messy, the resolution of hash collisions described here is accomplished in just a few lines of code. Furthermore, it is very efficient to determine whether a hash collision with the target string could exist. Therefore, the common case where no collision exists can be checked for without any significant overhead. With a hash range of 31 bits collisions will occur very rarely even in large directories, and collisions that lie on either side of a block boundary will be rarer yet.

In summary, once the hash probe step has identified a leaf block to search, the target name will always be found in that block if it exists, except in the unlikely event its hash collides with that of an entry in a successor block. Typically, then, the number of blocks that need to be accessed to perform a lookup is two - the index and the leaf - or three, for extremely large directories. Because the index tree consists of a very small number of blocks, even for large directories, it is a practical certainty that they will all be retained in cache across multiple operations on the same directory.

Entry Creation

Except for the leaf splitting operation - described separately below - creation of entries is simpler than lookup. The target leaf block in which the entry will be created is located as for a lookup operation (and with exactly the same code) then, if there is sufficient space, the entry is created in that block as in unindexed Ext2. If the target block has insufficient space then the block is first split, as described below, and the entry is created in either the original block or the new block, according to its hash value.

Entry Deletion

Deletion of a directory entry is accomplished in exactly the same way as with no index: the entry is located via a lookup operation, and marked as free space, merging it with the preceding directory entry record if possible.

It is possible that, after a large number of deletions and creations in a directory, a significant amount of free space could accumulate in the blocks which are indexed by only a narrow range of hash values. In such a case, it is would be desirable to coalesce some adjacent blocks. So far, no form of directory entry coalescing has been implemented in Ext2 and this has caused few problems, if any. However, it is unknown at this time how prone the HTree algorithm is to fragmentation so it may turn out to be necessary to implement coalescing sooner rather than later, as opposed to relying on the fsck utility.

Splitting Leaves and Index Blocks

Splitting a leaf block is the most complex step in the HTree algorithm, accounting for somewhat more than half the lines of code in the implementation. Nonetheless, there is little here that is subtle or difficult.

Splitting a leaf block requires moving approximately half the contents of the original to a newly created block such that the original entries are partitioned into two ranges of hash values. The two ranges are distinct, except that the highest hash value of the lower range may be the same as the lowest hash value of the upper range. This exception is made necessary by the possibility of hash collisions.

Since entries are stored unsorted in leaf nodes, the first step of the partitioning is a sort. First, all the entries in the block are scanned and a hash value is computed for each one. The entry locations and hash values are stored in a map, and this map is sorted rather than the entries themselves. The sort (a combsort) executes in $O(nLog(n))$ time where n is the number of entries. It has been suggested to me that the partitioning could be done in $O(n)$ time, but this is true only if we are willing to pick an *a priori* pivot value for the partition. In any event, the splitting occurs relatively rarely - for 4K blocks, less than once per hundred creates - and the sort is efficient.

In the current implementation, leaf blocks are always split at the halfway point in terms of number of entries, which is not entirely optimal. At the expense of somewhat more complexity the split point could be chosen in terms of total data size and could take account of the knowledge of which of the two blocks a new entry will be inserted into. Currently, the lowest hash value in the upper range is always chosen as the lower bound of that range. An improved strategy would use the hash value which is exactly between the lower bounds of the two adjoining hash ranges whenever possible, dividing up the hash space more evenly. These two improvements would allow the theoretical 75% average fullness of leaf blocks to be more closely approached.

After choosing the split point the entries of the upper hash range are copied to the new block and the remaining entries are compacted in the original block. This is done with the aid of the sort map described above. Some complexity in the step arises from the desire to carry out this operation within the space of the two blocks involved plus a small amount of stack space, so no working storage needs to be allocated.

Having split the entries, a new index entry consisting of a pointer to and lower hash bound of the new leaf block is inserted into the parent index. This may require that the parent index block be split or, if the index block is the root, a new level is added to the tree. Since only two levels of index are supported the recursion does not go further than this in the current implementation.

The lowest bit of the hash value of the new leaf block is used to flag the relatively rare case where the split point has been chosen between two entries with the same hash value. This bit forms part of the new leafs hash value and is carried naturally through any recursive splitting of index blocks that is required. So, as far as entry creation is concerned, just two lines of code are required to handled the messy-sounding problem of hash collisions in entry creation. (A few more lines are required to handle collisions on lookup.) As a side note, I did manage to conceive and implement a much more complex and hard to verify solution to this hash collision problem, which attempted to avoid splitting apart entries with equal hash value. Then I realized that the rarity of the event meant that the simple approach could be used with no significant impact on performance. In general, it is impossible to guarantee that colliding entries will never have to be split between blocks, since we may be so unlucky as to have a large number of strings hash to the same value.

Splitting an index block is trivial compared to splitting a leaf. Half the index entries are copied to a newly allocated index block, the count fields of block blocks are updated, and an index entry is created for the new block in the same way as for a new leaf block. Adding a new tree level is also trivial: the entries contained in the root index are copied to a new block and replaced by an index entry for the new block. In the current implementation, should the root of a two level tree be found to be full then the index is deemed to be full and the create operation will fail. At this point the directory would contain several tens of millions of entries or the index would have become badly fragmented. Though neither possibility is considered likely, both can be addressed by generalizing the implementation to N levels, and the second could be corrected by adding an index-rebuilding capability to the fsck utility, or by implementing a coalesce-on-delete feature as described in the penultimate section of this paper.

After all necessary splitting and index updating has been completed, and appropriate working variables updated, a new directory entry is created in the appropriate leaf block in the same way as for unindexed Ext2.

4 Comparison to Alternatives

In this section I briefly examine three alternative directory index implementation techniques that offer similar functionality to HTrees. All three of these techniques have been used successfully in other filesystems, but each of them has some flaw that makes it less than perfect for Ext2's requirements and design philosophy.

BTrees

The BTree ("balanced tree") algorithm offers good average and worst case search times with reasonably efficient insertion and deletion algorithms. Some variation on the BTree structure is typically the choice for a directory indexing design, and indeed BTree indexing is used in at least a number of Linux's supported filesystems.

Linux's ReiserFS[9] uses B*Trees which offer a 1/3rd reduction in internal fragmentation in return for slightly more complicated insertions and deletion algorithms. Keys in ReiserFS BTrees are fixed-length hashes of the indexed strings, therefore duplicate keys are allowed to accommodate key collisions. and the B*Tree algorithms are modified accordingly. SGI's XFS uses B+Trees with hashed keys. IBM's JFS (now ported to Linux) uses B+Trees with full-length key strings in the leaf nodes and minimal prefixes of the keys in the interior nodes. This variant is called a Prefix BTree.

Though directory performance is seldom specifically tested, all three of these filesystems are known for their good performance with large directories. Two of these three filesystems use hash directory keys for the same reason HTree uses them: the small fixed key size gives a high branching factor and thus a shallow tree.

The main difference between an HTree and a BTree is that the leaves of an HTrees are all at the same depth. Leaf nodes do not have to be specially marked and rebalancing is unnecessary, saving considerable complexity. A second distinction is the an HTree has one index entry for each leaf block whereas a BTree has one index entry for each directory entry. This means that an HTree has far fewer index blocks than a BTree and is therefore roughly one level shallower than a BTree with the same number of entries. In fact, the high branching factor and block granularity together make it improbable that an HTree will ever need to have more than two index levels, which are sufficient to contain several tens of millions of entries.

As names are inserted into and deleted from a BTree it may happen that some of the leaf nodes end up significantly further from the root than others. If such imbalance becomes too extreme then average search times may begin to suffer. To combat this, BTree algorithms incorporate rebalancing steps that detect excessive imbalance resulting from an insert or delete operation and correct it by rearranging nodes of the tree. Such rebalancing algorithms can become complex in implementation, especially when hash key collisions or additional requirements of B+Trees and B*Trees need to be handled.

In summary, the various forms of BTrees have all the functionality required for a directory index, but because of the rebalancing algorithms, are more complex to implement than HTrees. No clear advantage is offered in return.

Hash Tables

A normal hash table is a linear array of buckets, and the hash key directly indexes the bucket which is to be searched. Thus, finding the correct bucket to search is very fast. The chief drawback is that the hash table's size be chosen to be neither too large nor too small. A hash table that is too large will waste space and one that is too small will cause many collisions. Filesystem directories tend to vary in size by many orders of magnitude, so choosing an appropriate size for a hash table is problematic. This problem can be solved by allowing the hash table to grow as the number of strings in the hash table increases. When the hash table passes a certain threshold of fullness its contents are transfered to a larger table, an operation called "rehashing". If an integral factor is used for the expansion then hash values do not need to be recomputed and the process is efficient.

A linear hash table with a rehashing operation is thus seems a promising avenue to explore for a directory index.The rehash operation is mildly unappealing for filesystem use - how to store the hash table - how to represent collisions - can't use pointers in the objects - interaction with collisions and rehashing

Cached Index

The above-mentioned structural problems with hash tables can all circumvented neatly if the hash table is not persistent on disk but is instead constructed each time the directory is opened. This approach has been tried with good results by Ian Dowse[7].

Although the index can be maintained incrementally it must be initially constructed in its entirety so that the file creation operation can provide the necessary guarantee of uniqueness. This gives rise to two problems. First, starting with a cold cache the first access to any directory forces all blocks of the directory be read, even if only a single entry needs to be accessed. Thus, randomly accessing files in a large volume containing a large number of directories will be significantly slower than with a persistent index, until the cache has been fully initialized. This could visibly affect latency for an application such as a web server. Second, there is the requirement to cache hash tables for all directories accessed. It is not difficult to construct a case where a single file is access in each directory of a volume, cyclically. With a sufficiently large volume this must cause cache thrashing. Either of these problems could be exploited by a malicious user, and either could cause spikes of performance degradation on certain applications.

An advantage of the cached index approach is that the implementor is relieved of responsibility for maintaining structural compatibility of the index format across future revisions. A persistent index can be added at a later date, buying time to study and perfect design alternatives. The disadvantages of a cached index - cache thrashing and latency problems - do not affect the common cases unduly. Most users will be please with the afforded performance increase as compared to linear directory searching. However, if a persistent index design is available which performs well and does not have the disadvantages of a cached index, then it is hard to see why it should not be adopted.

In summary, the cached index approach is considered to be a worthwhile acceleration strategy, the value of which lies in providing performance enhancement for common cases over the short term.

5 Hashing

Hashing, in its specific application to directory indexing, is a subject to which an entire paper and perhaps should be devoted. Here, I will merely touch on a few of the relevant details.

The most important goal of a hash function is to distribute the output values across the output range. Secondary goals are speed and compactness.

Uniformity of distribution is especially important to the HTree algorithm. A nonuniform hash function could lead to very uneven splitting of the hash key space, which could dramatically increase the danger of directory fragmentation. It should be noted that some directory index designers have sought to exploit nonrandomness in hash functions, with a view to improving cache coherency for operations applied across entire directories. It is my opinion that such a goal is difficult to attain and in any event imposes a needless burden on the user. A better strategy is design the directory operations to be essentially as efficient with a random hash as with a favorably chosen nonrandom hash.

As part of the development process of the HTree indexing code I examined many hash algorithms, with the assistance of a number of others. Surprisingly, I found most hash algorithms in common use to be flawed in fundamental ways. The most common flaw I found is a reliance on randomness in the input string for randomness of the resulting hash value. When tested with nonrandom input strings such as names varying only in their last few characters, such hash function tend to produce very poorly distributed results. Unfortunately, nonrandom strings are all too common in directory index applications.

As an *ad hoc* test of the effectiveness of various hash functions I implemented a small userspace program that creates a large number of directory entries with nonrandom names; specifically, names that are identical in their first N characters, with a linearly increasing counter appended. After creating the entries I computed statistics on the leaf nodes to determine how evenly filled they were. In general, only one statistic matters: average fullness. In theory, perfectly uniformly distributed hash values would result in all leaf nodes being 75% full. In practice, I have seen as high as 71% average fullness and as low as 50%, the worst possible result.

As a result of this testing process I made the following empirical observations:

- If any step in a hash algorithm loses information then a final "mixing" step that attempts to improve randomness cannot repair the damage, and the result will be a poor distribution. The hash algorithm must attempt to use all information in the input string as fairly as possible.
- Where the hash value is built using a byte at a time from the input string, it is desirable that each byte affect the full range of bits of the intermediate result.
- CRC32 produced relatively poorly distributed results, apparently by design: it is not supposed to produce uniformly distributed results, but to detect bursts of bit errors.
- The best performing algorithms where based on theoretically sound pseudorandom generators, where at each step the random value is combined with a portion of the input string and used as the seed value for the next step.

After a number of marginally succesful experiments I hit on the idea of using a linear shift feedback register to generate a pseudo-random sequence which to combine progressively with the input string. At each step, a character is taken from the input string, multiplied by a relatively prime constant and *xor*ed with the current value of the pseudorandom sequence, which forms the seed for the next step. Whatever its theorectical basic, or lack thereof, this hash function produced very good and uniform results.

Later I invested some time surveying hash functions from the literature and around the web. None that I tested was able to outperform my early effort, to which I gave the name "dx_hack_hash". Interestingly, the hash function that came nearest in its ability to generate consistently uniform random results was obtained from the source code of Larry McVoy's BitKeeper source code management system. It too, is based on a pseudo-random number generator, although of a slightly different kind. The performance of various hash functions and associated theorectical basis needs to be investigated further.

The HTree index currently relies on the dx_hack_hash has function. It is necessary to subject this apparently well performing hash function to rigorous testing before the Ext2 HTree directory extension enters production use, because, in a sense, the hash function really forms part of the ondisk data structure. So it must perform well right from the start.

To accomodate the possibility that the initially adopted hash function might prove to be inadequate in the long run, either because weak spots in its performance are discovered or a superior hash function is developed, a simple scheme was devised whereby a new hash function could be added to the HTree code at a later date, and the old one retained to be used with any directories originally created with that hash. The newly incorporated hash function would assigned an ID number, one higher than the hash function before it, and each directory created thereafter would have the ID of the new hash function recorded in its header, so that any subsequent accesses to the same directory would use the same hash function.

Security: Guarding Against Hash Attacks

Modern computer systems must be proof not only against failure in normal course of operation, but when manipulated by a determined attacker with malicious intent. Where a hash algorithm is being used, if the attacker knows the hash algorithm then they might be able to induce a system to create a large number of directory entries all hashing to the same value, thus forcing long linear searches in the directory operations. It is conceivable that an effective denial of service attack could be developed by this means. Or perhaps the attacker would be able to fragment an HTree directory intentionally by creating a series of names all hashing to a given value until the block splits, then deleting the names and repeating with a new series hashing to an immediately adjacent value.

It turned out to be possible to devise a method that prevents an attacker from predicting the hash values of strings, even if the attacker has access to the source code and knows the algorithm. This method relies on the hash algorithm having at least one variable parameter that can be randomly generated at the time the directory is first created. This generated parameter is stored in the root index block and used for every operation on that directory. Since the attacker cannot predict the value of the random parameter, they cannot carry out the attack.

It is open for consideration whether this level of paranoia is justified.

6 Further Work

Further work is planned in number of areas including coalescing of partially empty directory blocks, improvements to cache efficiency for directory traversal operations on very large directories.

Coalescing

Traditionally, Ext2 has never performed any kind of coalescing on partially empty directory blocks. On the other hand, Ext2 directories are seldom very large, in part due to its poor performance on large directories. The larger directories made practical by efficient indexing make the issue of coalescing more important.

Coalescing presents a problem in that it is an inherently nonlocal operation. It is not desirable to impose a requirement of examining several neighboring blocks at each deletion step to see if they can be coalesced. I felt that the operation could be made much more efficient by recording some information about the fullness of each leaf block, directly in the index. It would then be possible to test neighbor blocks for suitability for coalescing without having the leaf blocks themselves in memory. To be effective, just a few bits of descriptive information would be needed. At the same time, it is clear that 32 bits is a far larger range than will ever be required for the logical blocks of a directory. Accordingly, I set aside the top 8 bits to be used as hints to help accelerate directory block coalescing, should this feature be implemented.

For forward compability these high order bits are masked off by the current code. This means that, should a volume with indexes created by a later version of the indexing code be remounted by an earlier version that knows nothing about coalescing, the advisory bits will simply be ignored. The later code will have to accommodate the possibility that the advisory bits may be wrong, but since this is just an optimization that does not present a serious problem.

Coalescing applies to the hash bucket divisions as well as to the data of leaf blocks. In other words, if enough insertions and deletions were performed in a directory the hash space could be cut into many small fragments. In turn, leaf blocks corresponding to the small fragments of hash space would likely be underfilled. This could lead to significant growth in the size of a directory. In practice, this has not been observed. Accordingly, further work on coalescing has been deferred for the time being.

Cache Efficiency

Tests using a directory of one million files on a machine with 128 MB of memory showed that mass file deletion was slower than mass file creation by roughly a factor of four, whereas for smaller directories (below a few hundred thousand entries) deletion was roughly as fast as creation. After some investigation the difference was found to be due to the mismatch between the storage order of directory entries and inodes. Inodes are 128 byte records, packed 32 per 4K block. During creation, inode numbers tend to be allocated sequentially so that after each inode blocks fills completely it is never referenced again. On the other hand, mass deletion is performed in the order in which directory entries are stored in directory leaf blocks. This is random by design. Unfortunately, each delete operation requires not only that the directory entry be cleared but that the inode be marked as deleted as well. Thus, the inode table blocks are touched in random order. This does not present a problem if sufficient cache is available, but if that is not the case then sometimes a block with undeleted inodes will have to be written out to make room for some other block on which an inode is to be deleted. In other words, thrashing will result.

To confirm this theory I wrote a test program that would first read all the directory entries, sort them by inode number, then delete then in the sorted order. The thrashing effects disappeared. While this served to prove the the problem had been correctly identified, it is not a practical solution since we cannot in general control the order in which user programs will carry out mass deletion: it will normally be in the order that directory entries are retrieved via a readdir operation. The source of the problem thus identified, it became apparent that not only deletion, but any directory traversal involving inode operations would be affected.

Next I wished to establish the worst case performance impact of this type of thrashing. This is obtained when every single deletion requires two inode table block disk transfers, resulting in a 32 times increase in IO operations. This worst case result can only be approached if available cache memory is very small in relationship to directory size, which might in turn be due to competition for cache from parallel processes.

Inode table thrashing can be controlled by adding memory. For example, the inode table blocks for one million files will fit comfortably in cache on a machine with 256 MB of memory, assuming half the memory was available for caching. At worst, the slowdown observed is only linear, not at all comparable to the quadratic slowdown caused by linear directory searching. However, cache efficiency remains a desirable goal. I carried out preliminary analysis work suggests it is possible to reduce the cache footprint logarithmically, by carefully controlling the allocation policy of inode numbers. This approach is attractive in that it does not require changes in the underlying directory index structure. It is considered practical to defer this work for the time being.

For completeness, I examined alternative index designs to determine whether there exists an equivalently good indexing strategy which is not susceptible to inode table thrashing for common operations. Such an index would necessarily record directory entries in the same order as the corresponding inodes. Recalling that the HTree design uses one index entry per block, this hypothetical design would multiply the number of index entries by the average number of directory entries per block, a factor of 200, conservatively. Since index entries are small, this is not as bad as it sounds. In effect, this would increase the size of the index to somewhat less than half the size of the leaf blocks, in total. Again this is not as bad as it sounds, because there would be no slack space in the leaf blocks. This would narrow the HTree method's size advantage to about 30 percent. However, the individual-index method imposes a new requirement: free space in each leaf block must be tracked. Some kind of persistent free space map would be needed and updating this map would require extra IO operations. The speed of fsck would be decreased measurably by the requirement to examine more index blocks. These size and performance disadvantages considered together leads to the conclusion that the fine-grained index approach's cache advantage in mass directory operations is not sufficient reason to prefer it over the method presented in this paper.

Nonlocal Splitting

A with HTrees, BTree blocks must be split as they become full. A B*Tree is a BTree variant that reduces the amount of unused space in split blocks by splitting groups of two blocks into three. This leaves each block approximately two thirds full, compared to half full in a normal BTree.

This same technique could be used with an HTree, and in fact the implementation is simpler because rebalancing is not required. The expectation would be to improve average block fullness from 3/4 to 5/6. It is for consideration whether this improvement warrants the additional complexity and slightly increased cost of the split operation.

7 Conclusions

The HTree - a uniform-depth hash-keyed tree - is a new kind of data structure that has been employed with apparent success to implement a directory index extension for Linux's Ext2 filesystem. Besides offering good performance and a simple implementation, the HTree structure allows Ext2's traditional directory file format to be retained, providing a high degree of both backward and forward compatibility. After a period of further refinement and testing, it is considered likely that the HTree directory indexing extension will become a standard part of Linux's Ext2 and Ext3 filesystems.

8 Acknowledgements

My heartfelt thanks to all those who helped make this work possible, especially:

innominate AG for employing me to help make Linux better

Uli Luckas for surviving many hours of code walkthroughs and debugging sessions

Andreas Dilger, for whom no detail was too unimportant to go uninvestigated

Mathew Wilcox, for being the first to proofread this paper

Stephen Tweedie for generally being encouraging, knowledgable and an all round nice guy

Ted Ts'o for putting me up to this in the first place

Anna just for being Anna

9 References

References

[1] Design and Implementation of the Second Extended Filesystem, http://e2fsprogs.sourceforge.net/ext2intro.html

[2] Analysis of the Ext2fs structure, http://step.polymtl.ca/~ldd/ext2fs/ext2fs_toc.html

[3] [rfc] Near-constant time directory index for Ext2, http://search.luky.org/linux-kernel.2001/msg00117.html

[4] [RFC] Ext2 Directory Index - File Structure http://lwn.net/2001/0412/a/index-format.php3

[5] HTree Performance: http://nl.linux.org/~phillips/htree/performance.png

[6] Journal File Systems, Juan I. Santos Florido, http://www.linuxgazette.com/issue55/florido.html

[7] BSD Dirhash: http://groups.yahoo.com/group/freebsd-hackers/message/62664

[8] XFS White Paper: http://oss.sgi.com/projects/xfs/papers/xfs_usenix/index.html

[9] ReiserFS Resources: http://www.namesys.com/