To appear in the 2003 IEEE International Conference on Cluster Computing (Cluster 2003), December 1-4, 2003, Hong Kong

Optimized Implementation of Extendible Hashing To Support Large File System Directory

Rongfeng Tang, Dan Meng, Sining Wu
National Research Center for Intelligent Computing Systems
Institute of Computing Technology,Chinese Academy of Sciences
{rf_tang, md,wsn}@ncic.ac.cn

Abstract

Extendible hashing is a kind of fast indexing technology; it provides with a way of storing structural data records so that each of them can be gotten very quickly. In this paper, we present our own specially optimized implementation of using extendible hashing in cluster file system in order to support large directory[1].

1. Introduction

More and more applications such as scientific computing and E-business will produce or access vast amount of files in just very short period of time and they tend to create large directories on their base file systems. However, most traditional file systems perform badly when dealing with this situation, since they store directory entries as an unsorted linear list on the second storage. On average, the file system need search through half of the directory to find a certain entry. This is tolerable to small directories, but it becomes too slow for big ones, for it not only causes excessively disk access but exhausts lots of CPU times, which lead to degression of system performance.

In order to address the problem described above, some other more efficient schemes have been put forward in several file systems: B-tree and dynamic hashing.

In theory, the average retrieval time of B-tree scheme is O (lgn), where n is the scale of the directory in unit of block. XFS [8] and Reiserfs [7] have adopted this scheme. However, there are some drawbacks exist, e.g. it need reorganization regularly. Moreover, the movement of entries is orderless, which bring problems for sequential directory scan when it is used in distributed environment.

Dynamic hashing scheme is another kind of data structures for efficient searching. It's not only keep the efficient searching advantage of static hashing, also has some essencial improvement to it: the hash table can be increased when the capacity is not fit for the requirement or some of the buckets overflow due to collision (two or more records mapping to the same hash table bucket). GPFS [4] and OpenGFS [2] have adopted this scheme, but implemented differently to each other and both have their own limitation and their implementation can not be directly applied to logical cluster file system such as PVFS and DCFS[5](Dawning Cluster File System developed by **National Research Center for Intelligent Computing Systems** of Institute of Computing Technology, Chinese Academy of Sciences).

It's limited by the host systems, e.g. for 32-bit linux with ext2 file system, the largest size of a single file is 2GB, since uniformly distributed hash function cannot be found, this limitation can sooner be reached by skew data , but the ratio of space utilization is very low, in some cases, a single directory can only contain few directory entries.

Also, as B-tree scheme, the splitting of bucket will often cause the directory entries to be moved elsewhere, that makes trouble for sequential scan operation, as a result, GPFS added more context information into the bucket and directory entry which make it incompatible with POSIX standard.

We've implemented this technique to organize and operate directories in the DCFS in our own way called Limited Multilevel Extendible Hashing (LMEH) and that can address the problems of dynamical Two files are assigned to each directory, one for hash table, and the other for directory entries, in this way, more entries can be contained in the entry file than others when facing with skewed hash values and it greatly improved the space utilization. Different hash table entry can correspond to variable number of virtual value so that we needn't re-initialize the hash table when buckets overflow. Furthermore, we've devised particular regulation for sequential scan in which only an additional field added into index structure and the bucket number at the head of each bucket to guarantee all entries to be returned only once and none existing will be missed.

The rest of this paper is organized as follows. In the next section we'll discuss in detail about LMEH. In section 3, we analyze the space utilization of LMEH by comparison. Section

[1] This work was supported by the National High-Technology Research and Development Program of China (863 Program) under the grant No.2002AA1Z2102 and the grant No.2002AA104410.

4 concludes our paper.

2. LMEH

In LMEH, two files are assigned to contain a single directory, one is index file for hashtable (one index structure for each hash table entry), and the other is directory entry file for hash buckets, i.e. directory entry blocks, so that they can be stored much more denser and greatly increase the ratio of space utilization. Different buckets can be indexed by different number of hash value bits and it is called a Multi-Level Way. The delayed-splitting scheme is introduced which can reduce the times of hash table split to some extent; We devise a special searching regulation, so that only a little context information stored in the index structure to support the sequential directory scan operation. Also, in order to cope with the skewed-hash-value problem, we limit the maximum bits number of hash value used for indexing just like the OpenGFS does.

2.1. Description

The index file, containing hash table of relevant directory, consists of index structures corresponding to every hash bucket number ever used. Each index structure stored at the offset $n * s$, where n is the hash bucket number, s is the size of index structure.

The index structure defined as:

```
Struct index_structure {
        Local_depth,
        Offset_within_file,
        Offset_within_bucket
};
```

The Local_depth is the number of bits currently used in this buckets for indexing entries; Offset_within_file records the offset of bucket corresponding to this index structure within the directory entry file; Offset_within_bucket is used for sequential scan, we'll describe it in the following sections.

Only a few least significant bits of hash value are used at first. When a bucket overflows, we allocate another one at the tail of the entry file and initialize the index structure points to this bucket: assign the offset of bucket in entry file to Offset_within_file and currently used bit number of hash value to Local_depth; other index structures relating to non-used hash values needn't be considered at present and just leave a "hole" in the index file.

We illustrate our scheme with an example showed by figure1(Note that all the numbers shown in index file are in binary format).

In figure 1, the black pane in index file indicates its corresponding bucket is full and the next inserting will cause it to be split; the one in off-white points to bucket with some entries but still has capacity remain for new ones; and the colorless pane means a "hole" in index file, to which related hash value has not been used yet. The number inside each pane in binary is the hash number and bit number indicates the Local_depth of the bucket.

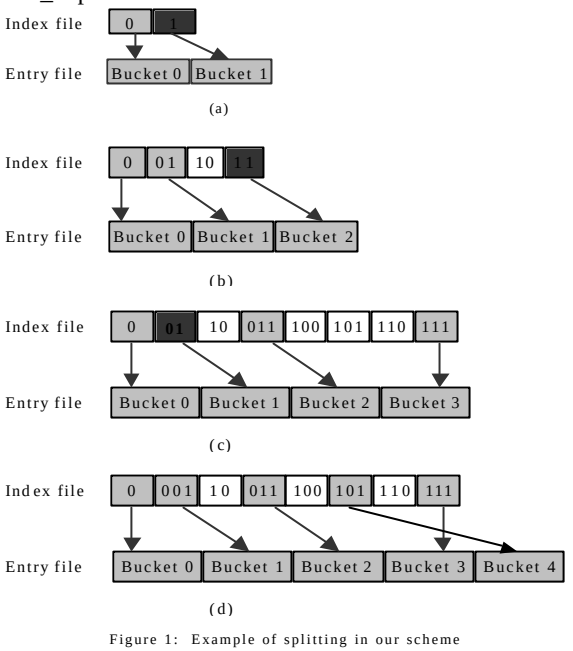

Figure 1: Example of splitting in our scheme

Figure 1 (a) shows only one bit of hash value is effective and there are two buckets --bucket0 and bucket 1-- assigned to hash number 0 and 1 respectively. Now the bucket 1 (in black) is overflowing, so it must be split. The index file is doubled, because new index structure for hash number 11 is added, and new bucket—bucket2—is allocated at the end of entry file, then all the entries in bucket1 with the bits pattern 1* are moved to bucket2, see figure 1(b). Since hash value 10 is non-used at present, we just leave it alone. We only deal with the new index structure newly created and needn't to reinitialize the whole hash table as OpenGFS does, which is time-consuming. Figure 1(c) shows the splitting result of bucket 2 in figure 1(b).

Just like every bucket has its own local_depth, the whole hash table has a global_depth, which equals to the maximum of bit number used by existing buckets. The global_depth can be calculated from the size of index file by follow formulation:

$$\left\lceil \log_2 \frac{fSize}{iSize} \right\rceil$$

where *fSize* is the index file size and *iSize* is the size of index structure, the outcome value is rounded up to next integer.

The local_depth assigned to the same value as global_depth each time when the bucket is created and split. If the local_depth of bucket to be split equals to the global_depth, which means the current number of bits used is not sufficient to differentiate all existing entries, the hash table must be extended,

otherwise, we just insert the index structure into the hole of index file and the table size is remain unchanged. E.g. assume bucket1 in figure 1(c) overflows. Currently its local_depth is 2 and global depth is 3, then we just allocate *bucket4* to index structure *101*, and move all entries with hash value *101* to *bucket4* (see figure 1 (d)), now global_depth is not incremented and the local_depth of two interrelated buckets is equal to global_depth. It shows we split the bucket only when it must be, and by this delayed-splitting way, we can reduce the number of splitting to some extent, thus, can increase system performance and save storage space as well.

In order to deal with the problem due to skewed hash value, we limit the hash table to a certain size, i.e. only a subset of hash value bits are used and new buckets is chained to the overflowed one. Searching through a long chain of buckets will decrease performance, so we must find an efficient way to organize the entries with same hash value.

In the next three sections, we'll give detailed descriptions for generic directory operations: insertion, lookup and sequential scan with our scheme.

2.2. Insertion

When we execute shell command such as mkdir, new directory entry will be added to the parent directory. This processes is not very difficult. First of all, we get the hash value *v* of the entry by its name and compute the bucket number *n* as the global_depth least significant bits of *v*, then its corresponding index structure is located at offset n*(index structure size) within index file. There are two cases must be distinguished. If the index structure is not null, the bucket is positioned at offset indicated by Offset_within_file in the entry file and read in from it, if it's full, we must take care of the overflows, otherwise, put the new entry into it. If we get a null index structure, we must re-count the hash value by fewer numbers of bits and then repeat above process.

we allow every bucket corresponds to different number of effective hash bit and delay split ituntil it is full, which, to some extend, can reduce total splitting times and save space. Also, only the index structures of overflowing bucket and its siblings newly created are affected, so that can avoid additional I/O accesses as in OpenGFS.

2.3. Lookup

The purpose of lookup is to find the entry with the given name in a directory. We begin with computing the hash value, then get the target bucket which is sure to hold the entry if it exists. When a "hole" encountered, we re-count the bucket number with fewer bits of hash value.

if the entry exists, the the procedure returns it, otherwise, it returns null.

The lookup operation in our implementation usually includes two I/O accesses: one for index structure, and the other for bucket. Since we limit the hash value from growing too large, the size of index file is very small (2 MB at most), thus, it is often cached by operating system after first reading and the read request afterwards can be satisfied quickly.

2.4. Sequential Scan

The POSIX standards have defined interfaces for fetching a set of entries from a directory starts with a certain offset, such as readdir. in most cases, it's very simple, for we just read in the entries from the entry file directly. However, the split between sequential scans will move the entries to other bucket so that it is likely to return the same entry twice or missing some ones in existence. It's all right when split happens in bucket that we'll read in future, because the entries are only moved to bucket with higher number. The special consideration is required when it happens after the scan has proceeded past the affected buckets or when the sequential scan has returned some but not all of the entries in bucket affected by the split. In order to cope with this problem, an additional field called Off_within_bucket is added in the index structure. When a bucket split and new one allocated, Off_within_bucket in index structure corresponding to new bucket records the offset of the last entry moved from affected bucket into it. We present these two cases respectively.

There is a particular characteristic in our scheme: if the local_depth of bucket is little than global_depth, all its sibling buckets are not allocated at present. E.g. if global_depth is 5 and bucket number is 101 in binary (local_depth is 3), then those corresponding to hash number 01101, 10101 and 11101 are not used currently. What's more, entries in the splitting bucket are moved to the buckets with larger hash number. For example, if bucket101 overflows, its entries can only to be moved to bucket01101, 10101 and 11101. So we can devise our scan regulation that, if the local_depth of affected bucket is little than global_depth, its sibling buckets will be skipped during this time of opening, otherwise, some of the entries will be returned twice. Such as in above example, if bucket101 is affected, will skip reading entries in bucket01101, 10101 and 11101.

Still Another case exists that we've just read some of entries from the bucket when it's split. Similar to the case above, some of entries will be moved forwards, so the second sequential scan may miss reading directory entries. We use a cursor = (orig_b, orig_depth) to solve this problem, Here, orig_b is the bucket number affected by previous call and orig_depth is the local_depth of it for that time. The cursor is input in every sequential call and the first one after the directory opened is (-1, -1). There are two different cases must take care of when the call invoked:

(1) If the number of bucket to be read is equal to orig_b, and its local_depth equal to orig_depth as well, than we just read in the entries, for it hasn't been split ever since last call.
(2) If the bucket to be affected is equal to orig_b and its local_depth is little than orig_depth, which means the bucket has been split after the last call, then we must fetch

the entries from all sibling buckets and rebuild its original content. The Off_within_bucket of index structure point to the position of last entry moved into this bucket during splitting.

3. Space Utilization Analysis

Here we define space occupancy efficiency as:

$$\frac{eff\_space}{total\_space};$$

Where eff_space is the size of space actually used by directory (we don't consider the fragments within bucket) and total_space is the all space assigned to. As to GPFS, the total_space is the size of its sparse file, for, although the holes inside file may occupy no storage on device but it still possesses the logical space of file. While in IMEH, it also includes the index file. The efficiency number indicates the fraction of the space allocated for the directory that actually used to contain directory entries and its index structures.

In LMEH, the "holes" resulted by skewed hash value are limited into the index file only, so the space utilization will be improved compared with the directoryless scheme, since the index structure occupies much less space than bucket does.

Suppose the same hash function used, so that for the identical set of entries, the number of "holes" generated is equal. Assume the number is *n*, *isize* and *bsize* are the size of index structure and bucket respectively. For LMEH, its space efficiency can be computed as formulation (1):

$$\frac{(2^{global\_depth} - n) * bsize}{2^{global\_depth} * (isize + bsize)} \qquad \text{--------} \quad (1)$$

and as to directoryless scheme, its space efficiency is computed as formulation(2):

$$\frac{(2^{global\_depth} - n) * bsize}{2^{global\_depth} * bsize}$$

$$= \frac{(2^{global\_depth} - n)}{2^{global\_depth}} \qquad \text{-------} \quad (2)$$

Since the coefficient of formulation (1) is less than that of formulation (2), which means, given the same size of file space, the LMEH can store much more directory entries than directoryless scheme, thus, the LMEH has much better space utilization than that use directoryless scheme.

4. Conclusion

In this paper, we presented our own implementation of extendible hashing for supporting large file system directories. It's efficient, simple and compatible both forward and backward.

In next step, we'll try our best to devise a more time-saving technique to organize the entries corresponding to the same bucket number and, furthermore, find a way to couple the hashing scheme into file system better.

Acknowledge

We thank Xiong Jin and He Jin; they are main forces of developing DCFS and provide us with an excellent platform to do our experiment.

References

[1] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger,and H. Raymond Strong. Extendible Hashing – a fast access method for dynamic files. *ACM Transactionson Database Systems*, 4(3):315-344, September1979.

[2] Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow,Grant M. Erickson, Erling Nygaard, Christopher J. Sabol,Steven R. Soltis, David C. Teigland, and Matthew T. O'Keefe. A 64-bit, shared disk file system for linux. appear in the Sixteenth IEEE Mass Storage Systems Symposium held jointly with the Seventh NASA Goddard Conference on Mass Storage Systems & Technologies, March 15–18, 1999, San Diego, California.

[3] R.J.Enbody, H.C.Du. Dynamic hashing schemes. 1988 ACM 0360-0300/88/0600-0085.

[4] a Frank Schmuck, Roger Haskin. GPFS: a shared-disk file system for large computing clusters. Proceedings of the Conference on File and Storage Technologies (FAST'02),28–30 January 2002, Monterey, CA, pp. 231–244. (USENIX, Berkeley, CA.).

[5] He Jin. Research on improving performance and correctness of cluster file system. Ph.D. dissertation.

[6] Sven Helmer, Thomos Neumann, Guido Moerkotte.A robust scheme for multilevel extendible hashing. Reihe Informatic 19/2001.

[7] Reiserfs. http://www.namesys.com

[8] XFS detailed design – Directory structures. http://oss.sgi.com/projects/xfs/design_docs/.