UC San Diego
Technical Reports

Title

Improved Linux File System Hashing

Permalink
https://escholarship.org/uc/item/5nn680xq

Authors
Chen, Ying
Burkhard, Walter
Palmer, John

Publication Date
2001-07-16

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

submitted to the *First Conference on File and Storage Technologies* (FAST'02), Monterey, California, January 28-29, 2002.

Improved Linux File System Hashing

Ying Chen[°†] Walter A. Burkhard[*] John D. Palmer[°]

[*]Gemini Storage Systems Laboratory
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114 USA

[°]Almaden Research Center
IBM Research Division
650 Harry Road
San Jose, CA 95120-6099 USA

July 16, 2001

Abstract

The Linux kernel utilizes caches to speed up accesses to the high-usage objects that are normally stored on disks, such as data cache pages for files, inodes, and directory entries. These cache data types are implemented using separate chaining. In our experience, we found that some of the hash functions used in the circa 2000 Linux kernel can result, for certain access patterns, in very skewed data distribution over the chains. Better hash functions are desirable. In this paper, we analyze the data distribution and run-time efficiencies of a set of hash functions for Linux inode and directory caches. Our analysis utilizes data traces from both an industry standard benchmark and two real world computing environments. Although some of the results were anticipated by Lever, who has carried out a similar study, we show that his results can be further improved.

1 Introduction

Modern operating systems, such as Linux, typically contain caches for various data objects to improve access performance. For example, most of the UNIX operating systems implement inode, directory entry, and data buffer caches. In Linux, these cache data types are implemented as tables with separate chaining collision resolution [2, 3]. In this paper, we examine tables designed for caching inodes and directory entries in Linux.

Since Linux is able to select the table size, that is, the number of chains in the table, based on the available memory in the system during system boot-time, the use of tables provides an expeditious compromise between fast access times on large memory machines and little waste space on smaller memory machines. We refer to the directory entry cache as the *dcache* while the inode cache retains its nomenclature – *inode cache*. Our previous experiments with the Linux kernel running large scale I/O industry standard benchmarks indicated that the circa 2000 Linux hash functions used for the inode cache and the dcache can result in significantly skewed data distributions over the table chains; this greatly slows average cache accesses. This experience led to our curiosity in the Linux implementation of these tables and their effectiveness for various cache access patterns.

Our study evaluates a set of hash functions that could be used for Linux dcache and inode cache. Our evaluation is based on traces gathered from both the industry standard benchmark and two real workloads. Our experiments will be of two varieties. The first measures structural results – how the to-be-hashed data is distributed among the table

[†] Ying Chen is currently with Zambeel Inc., Fremont; `ying@zambeel.com`

chains. We present the mean chain length, the chain length standard deviation, as well the maximum chain length for various table sizes. The second measures hash function run-times. We show that with carefully chosen hash functions, the data distribution skew can be largely eliminated for different types of cache access streams. Moreover, such hash functions are computationally inexpensive on relatively modern processor architectures such as Intel Pentium class machines. Our results improve the Linux kernel hash functions.

The paper is organized in four sections as follows. We describe our evaluation of different hash function in section 2. This is followed by a brief description of related work on Linux hashing in section 3. Finally, we draw conclusions in section 4.

2 Performance

The Linux dcache and inode cache are two of the most important cache structures for file system access. The inode cache is designed to speed up the accesses to inodes stored on disk while the dcache is designed to speed up file/directory name lookups.

To examine the effectiveness of the hash functions for the inode cache and dcache, we compare performance of the hash functions used in the circa 2000 Linux kernel with several other hash functions. The other functions are based upon *i*) multiplicative overflow arithmetic and *ii*) shift & add operations. Both varieties are discussed in Knuth [3]. We also considered a function based upon the standard library `fmod` function which worked very well in both the inode cache as well as the dcache; however, the `fmod` run-time is slower by a factor of approximately one-hundred. Here is the `fmod` function schema:

```
unsigned long hash ( int value )
{
     return  (unsigned)( fmod(0.6180339887*value, 1.0) * tablesize ) ;
}
```

We consider performance as defined by two measures:

i. The data distribution over the tables of differing sizes measured by the chain length standard deviation and the maximum chain length.

ii. The run-time efficiency of the hash functions.

Although these two do not necessarily translate into the performance of a particular application or workload, we believe that they characterize the average system behavior well.

We examine the hash table data distribution and run-times by feeding each hash function with two types of data streams gathered from two local computing environments. The first set is gathered from an running industry standard benchmark suite. The second set is gathered from two local file servers shared by a group of researchers and engineers in IBM Almaden Research laboratory.

2.1 Workload characteristics

2.1.1 SPEC SFS 2.0

The industry standard benchmark suite SPEC SFS 2.0 [1] is designed to stress network file servers by issuing a mix of NFS (Network File System) requests such as lookup, getattr, read, write, etc. We briefly describe the characteristics relating to dcache and inode cache usage. SPEC SFS 2.0 allows users to measure the network file server performance under differing levels of I/O loads. Under high I/O loads, SPEC SFS 2.0 not only issues I/O requests at faster speed, it also scales-up its working set by accessing larger directories and more files. For instance, when the requested I/O rate is 9000 IOPS (input/output operations per second), SPEC SFS will create a working set of 3.5 million files and 117,000 directories during the benchmark initialization phase, 10% of which are accessed during the benchmark run. At 2000 and 5500 I/O operations per second, there are proportionally fewer files and directories. Such workloads place high demands on the run-time and space performance of dcache and inode cache management.

For our inode hashing evaluation, we extracted the inode numbers for all data objects accessed during a particular run of SPEC SFS 2.0 at a given I/O rate; these runs take five minutes each. Any particular file can have differing inode numbers even within a particular file system run since the file may be created and deleted from cache numerous times. We present the statistical analysis using the inode number trace gathered at 2000 and 5500 IOPS . The total number of distinct inodes accessed at 2000 IOPS is 814,348. We also gathered 95,607 inode numbers at the rate of 5500 I/O operations per second.

For the dcache evaluation, we extracted the file names from the SPEC SFS 2.0 2000 I/O operations per second run as described above. The SPEC SFS 2.0 file names are of the form "file_en.xxxxx", where "xxxxx" is a number starting from "00000"; similarly, the directory names are "dir_ent.xxxxx." We used these file names in the dcache hash function evaluation. Clearly, these names are not representative of most file systems.

2.1.2 Local file server

We gathered the inode numbers and dcache names from two of the local file servers used by a small group of researchers and engineers (approximately ten people) in IBM Almaden Research Center. The first file server is an OS/2 file server, which serves as a repository for software packages, tools, development source codes shared by the group for most of the Microsoft Windows environments. It also stores different types of data for members of the group. This file server has been in use for more than three years. We extracted the file names from this file server for our dcache hash function evaluation.

Another local file server under examination was a Linux-based file server. This file server stores similar types of data to those stored in the OS/2 server, except that the file server stores most of the Linux-related software and information. We used the file names and inode numbers from this file server for our hash function evaluation.

2.2 Inode and Dcache hash functions

The `hash` and `d_hash` file system functions found in `inode.c` and `dcache.c` were created in 1997 and initially released within Linux version 2.2.0. During the four subsequent years, variations on these functions have occurred in

the various Linux releases up to the current version 2.4.3. Our experimental work considers the relative performance of these varieties of Linux kernel hash functions together with an alternative hash function.

There are four varieties of `hash` during this four year interval; each is presented with an associated label designating which version(s) of Linux contain the function. The label are `2.2.0-18` designates versions 2.2.0 through 2.2.18, `2.2.19` designates version 2.2.19, `2.4.0-1` designates 2.4.0 and 2.4.1, and `2.4.2-3` designates the the latest two releases.

```
unsigned long hash2.2.0-18 ( super_block * sb , unsigned long i_ino )
{
  unsigned long tmp = i_ino | (unsigned long) sb ;
  tmp = tmp + (tmp >> HASH_BITS) + (tmp >> HASH_BITS*2) ;
  return tmp & HASH_MASK ;
}

unsigned long hash2.2.19 ( super_block * sb , unsigned long i_ino )
{
  unsigned long tmp = i_ino | (unsigned long) sb ;
  tmp = tmp + (tmp >> HASH_BITS) ;
  return tmp & HASH_MASK ;
}

unsigned long hash2.4.0-1 ( super_block * sb , unsigned long i_ino )
{
  unsigned long tmp = i_ino | ((unsigned long) sb / L1_CACHE_BYTES ) ;
  tmp = tmp + (tmp >> I_HASHBITS) + (tmp >> I_HASHBITS*2) ;
  return tmp & I_HASHMASK ;
}

unsigned long hash2.4.2-3 ( super_block * sb , unsigned long i_ino )
{
  unsigned long tmp = i_ino | ((unsigned long) sb / L1_CACHE_BYTES ) ;
  tmp = tmp + (tmp >> I_HASHBITS) ;
  return tmp & I_HASHMASK ;
}
```

The manifest constant `L1_CACHE_BYTES` is set to 32 in the source. The manifest constants `HASH_BITS` and `HASH_MASK` are set to 8 and 255 in the source. The manifest constants `I_HASHBITS` and `I_HASHMASK` encode variables to allow system sizing.

There are three structural varieties of `d_hash` within the four year interval; each is presented here also with a label to identify the associated Linux version(s).

```
unsigned long d_hash2.2.0-13 ( struct dentry * parent , unsigned long hash )
{
  hash += (unsigned long) parent ;
  hash = hash ^ (hash >> D_HASHBITS) ^ (hash >> D_HASHBITS*2) ;
  return dentry_hashtable + ( hash & D_HASHMASK ) ;
}
```

```
unsigned long d_hash2.2.14-18 ( struct dentry * parent , unsigned long hash )
{
  hash += (unsigned long) parent / L1_CACHE_BYTES ;
  hash = hash ^ (hash >> D_HASHBITS) ^ (hash >> D_HASHBITS*2) ;
  return dentry_hashtable + ( hash & D_HASHMASK ) ;
}

unsigned long d_hash2.2.19 ( struct dentry * parent , unsigned long hash )
{
  hash += (unsigned long) parent / L1_CACHE_BYTES ;
  hash = hash ^ (hash >> D_HASHBITS) ;
  return dentry_hashtable + ( hash & D_HASHMASK ) ;
}


unsigned long d_hash2.4.0-1 ( struct dentry * parent , unsigned long hash )
{
  hash += (unsigned long) parent / L1_CACHE_BYTES ;
  hash = hash ^ (hash >> D_HASHBITS) ^ (hash >> D_HASHBITS*2) ;
  return dentry_hashtable + ( hash & D_HASHMASK ) ;
}

unsigned long d_hash2.4.2-3 ( struct dentry * parent , unsigned long hash )
{
  hash += (unsigned long) parent / L1_CACHE_BYTES ;
  hash = hash ^ (hash >> D_HASHBITS) ;
  return dentry_hashtable + ( hash & D_HASHMASK ) ;
}
```

The `L1_CACHE_BYTES` is the same as before. While the manifest constant `D_HASHBITS` and `D_HASHMASK` names do not change, beginning with version 2.2.14 they too become variables. All of our experiments will utilize variables for both the "bits" and the "mask" parameters in these functions. The `dentry_hashtable` reference has no impact on our structural results; accordingly, it is set to zero in our experiments. The `d_hash2.4.2-3` and `d_hash2.2.19` functions have the same structure but the `hash` argument differs as we note subsequently. The `d_hash2.2.14-18` and `d_hash2.4.0-1` functions have identical structure as well as arguments. Each of the function definitions in this section is preceded by the phrase `static inline` providing privacy and improved runtime performance; the phrase, omitted above, has no effect on our structural results.

2.3 Hash function evaluation environment

Our analysis of the hash chain length is relatively independent of the underlying platform configuration, such as processor speed and memory size (as long as the table is not larger than the amount of memory available in the system). Our SPEC SFS 2.0 trace data was gathered on a Dell PowerEdge 6300/550 4-way SMP server running four 550 MHz Xeon processors. At the time of our evaluation, we ran the RedHat 6.2 Linux 2.4-test9 development kernel.

The run-time analysis of hash functions depends on the system configuration. Our timing evaluations are carried out on a 500 MHz Pentium III system running the RedHat 6.2 Linux 2.4-test9 development kernel configuration as well as a Pentium III system running the Slackware 7.1 Linux 2.2.16 kernel configuration.

2.4 Hashing analysis

2.4.1 Inode cache

The Linux inode hash table function uses as parameters 32-bit inode numbers and the memory address of the superblock structure containing the inode. The inode hash function returns the chain location in the table. Note that our inode number traces did not include the superblock address information. In Linux, there is one superblock per file system; inodes from the same file system will normally have the same superblock address. In general, the set of superblock addresses for a given file server will be relatively small (probably in the order of tens). In our SPEC SFS 2.0 tests, all the files reside on the same file system, so there is only one superblock address. In our hash evaluation, we used a set of superblock addresses extracted across several system reboots from the SPEC SFS 2.0 runs as well as several randomly generated addresses (within a given memory address space range). We determine the performance of the four inode hash functions presented in section 2.2 as well as `hashPhi` which is defined in figure 1. The `hashPhi` function is based upon multiplicative overflow arithmetic [3]. We present results depending upon the superblock address and table size.

```
unsigned long hashPhi ( struct super_block * sb , unsigned long i_ino )
{
  i_ino += (unsigned long) sb ;
  return ( (i_ino*2654435761UL) >> (32-I_HASHBITS) ) & I_HASHMASK ;
}
```

Figure 1: Inode `hashPhi` hash function

The first set of inode results, in figure 2, based upon the SPEC SFS 2.0 benchmark data provides structural information regarding the data dispersal throughout a table. For each size of table, determined by the number of chains, the mean chain length as well as the chain length standard deviation and the maximum chain length are presented.

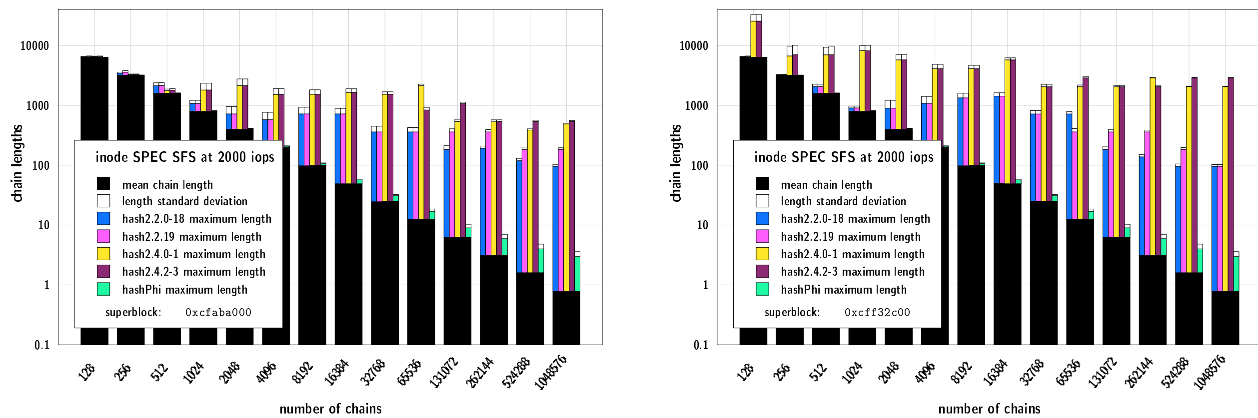

Figure 2: Inode hash function structural results for SPEC SFS at 2000 IOPS
superblock addresses: `0xcfaba000` and `0xcff32c00`

The distribution of inode data among the chains is most uniform for the integer overflow arithmetic hash function

hashPhi. Our results for hashPhi did not vary significantly with different superblock addresses; however, the circa 2000 hash2.4.0-1 and hash2.4.2-3 functions exhibit sensitivity to superblock address especially for smaller table sizes. The hashPhi function maximum chain lengths are generally smaller with smaller standard deviations. This behavior is similar for a larger collection of possible superblock addresses; similar trends are seen in another SPEC SFS trace gathered at 5500 I/O operations per second (figure 3) as well as the inode data gathered from the local Linux file server (figure 4).

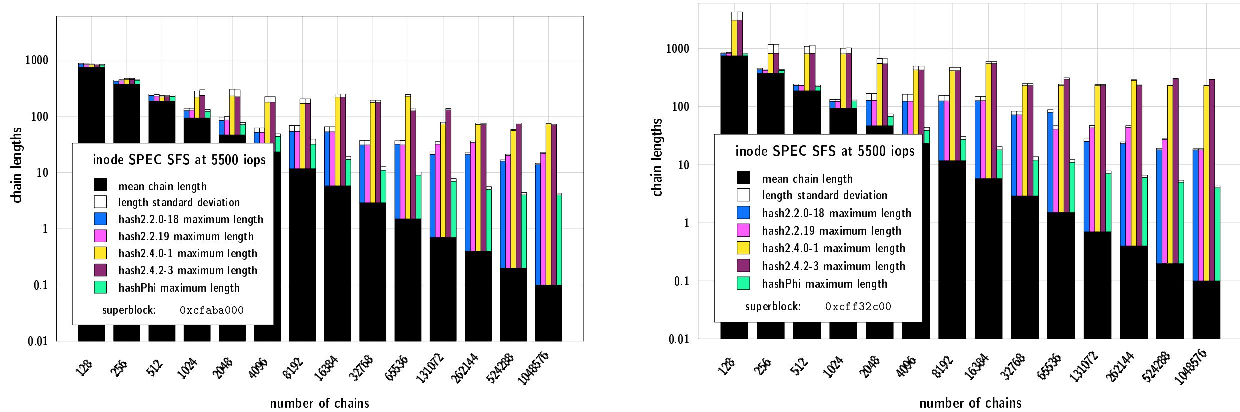

Figure 3: Inode hash function structural results for SPEC SFS at 5500 IOPS
superblock addresses: 0xcfaba000 and 0xcff32c00

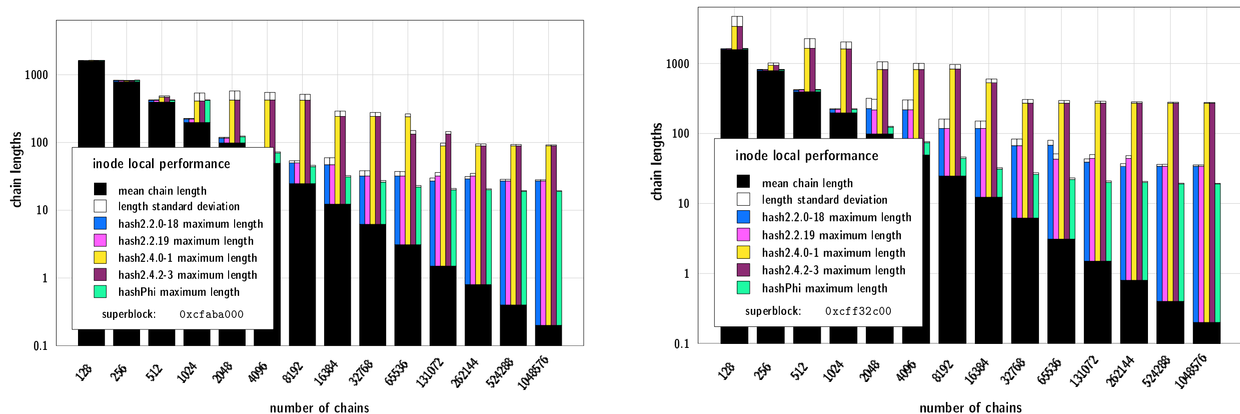

Figure 4: Inode hash function structural results for local Linux file server
superblock addresses: 0xcfaba000 and 0xcff32c00

The run-time performance for the inode hash functions is obtained via a short test suite employing the timeval data structure with 100,000,000 invocations of each hash function. The functions, compiled using gcc using one of three optimizations O1, O2, O3, obtain essentially identical results. The run-times, for a 233MHz Pentium III, are approximately 12 nanoseconds per evaluation and, on the 550MHz Xeon processor, are approximately 6 nanoseconds per evaluation. The O0 "optimization" obtains significantly slower results; 85 nanoseconds per evaluation on the Xeon and 320 nanoseconds on the Pentium. With the essentially identical run-times of the hash functions, the function with the best distribution properties should be selected. The hashPhi function is much better than the previous inode hash

functions for the SPEC SFS 2.0 as well as our local Linux file system inode data.

2.4.2 Dcache

The Linux dcache hash function consists two parts. In the first part, a given file name is mapped, via `full_name_hash` found in `dcache.h`, from a string to an integer value. Then the integer value, together with the parent pointer of the file/directory being hashed, serve as parameters to a second hash function `d_hash` which determines the chain number. Throughout the four year interval, there have been two slightly different versions of `full_name_hash`; each is presented here also with a label to identify the associated Linux version(s). The label `Old` designates Linux versions 2.2.0 through 2.4.1 and the label `New` designates the current versions 2.4.2 and 2.4.3. The function `full_name_hash` is identical for both the `New` and the `Old` schemes; only the implementation functions `partial_name_hash` and `end_name_hash` differ as noted here. The function `init_name_hash()` has constant value zero. Each of these functions is preceded by the phrase `static inline` as in the previous section.

```
unsigned int full_name_hash ( const unsigned char * name , unsigned int len )
{
  unsigned long hash = init_name_hash() ;
  while ( len-- )
    hash = partial_name_hash( *name++ , hash ) ;
  return end_name_hash( hash ) ;
}

unsigned int partial_name_hashNew ( unsigned long c , unsigned long prevhash )
{
  return ( prevhash + ( c << 4 ) + ( c >> 4 ) )*11 ;
}

unsigned long end_name_hashNew ( unsigned long hash )
{
  return (unsigned int) hash ;
}


unsigned int partial_name_hashOld ( unsigned long c , unsigned long prevhash )
{
  prevhash = ( prevhash << 4 ) | ( prevhash >> (8*sizeof(unsigned long)-4) ) ;
  return prevhash ^ c ;
}

unsigned long end_name_hashOld ( unsigned long hash )
{
  if ( sizeof(hash) < sizeof(unsigned int) )
    hash += hash >> 4*sizeof(hash) ;
  return (unsigned int) hash ;
}
```

We considered two other table driven functions, described elsewhere [6] [5] for the string to numeric conversion process. One of these other functions had structural behavior similar to that of `full_name_hash` and the other was noticeably less capable. Accordingly, we consider only the pair of `full_name_hash` functions in this context.

Our file name traces did not include the memory addresses of the parent directories. Instead, we used a set of 24 randomly generated memory addresses for the parent directories since we created 24 directories within the SPEC SFS runs. Our evaluation shows that these parent directory addresses do not substantially affect the distribution results.

We consider four different dentry hash functions as well as our alternative `d_hashPhi`; the labeling in figures 6 and 7 is identical to that of section 2.4.1. The `d_hashPhi` function is presented here; our remarks in section 2.2 regarding `dentry_table` are pertinent.

```
unsigned long d_hashPhi ( struct dentry * parent , unsigned long hash )
 {
   hash += (unsigned long) parent ;
   return dentry_hashtable +
      ( ( (hash*2654435761UL) >> (32-D_HASHBITS) ) & D_HASHMASK ) ;
 }
```

Figure 5: Dcache `d_hashPhi` hash function

These results are obtained from the SPEC SFS 2.0 benchmark suite which created approximately 34,000 dentry items within our execution environment as well as two local OS/2 file systems `E` and `F` name sets of sizes 42,132 and 78,365. Figure 6 presents the SPEC SFS dcache hash function structural results. Figure 7 presents the dcache structural results for the local file server.

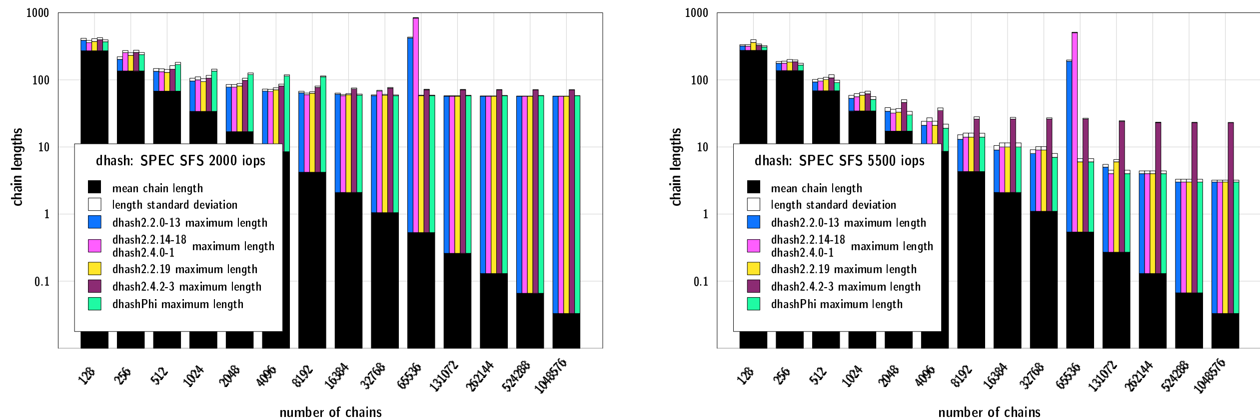

Figure 6: SPEC SFS 2.0 dentry hash function structural results at 2000 & 5500 IOPS

Each of these samples shows the anomalous behavior of some Linux hash functions for tables of size 65536. The SPEC data, in which the file names have a single structure – a common suffix “`file_en.`” followed by a number or “`dir_ent.`” followed by a number – is particularly demonstrative of this weakness. The most recent version 2.4.2-3 performs fairly well in this regard. However the `d_hashPhi` function provides relatively small standard deviation of chain length as well as excellent maximum chain length results throughout.

The run-times of these functions are essentially the same as that for the inode functions given in section 2.4.1.

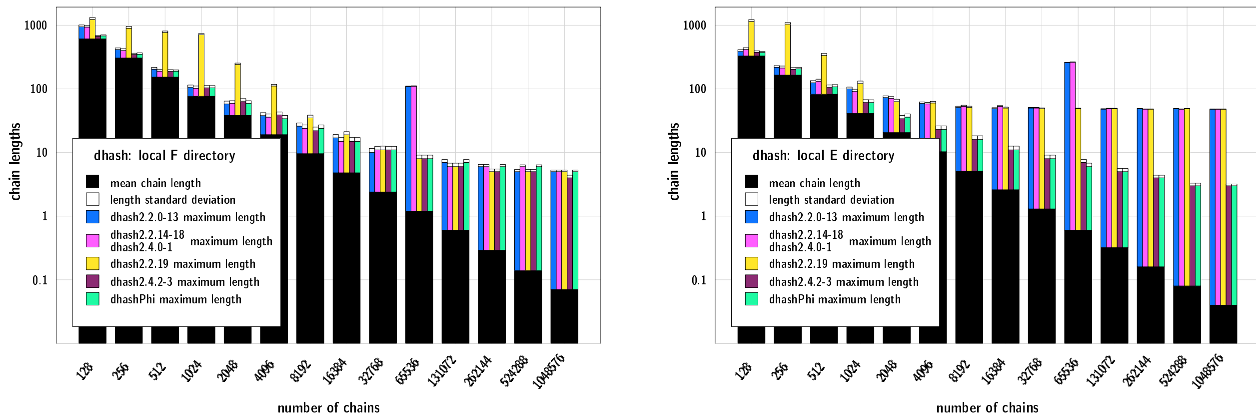

Figure 7: Dentry hash function structural results for the E and F file systems, local OS/2 server

3 Related Work

File system hashing has improved with the evolution of Linux; the most recent changes were introduced within version 2.2.0 in 1997. The Linux source code, even in version 2.2.0 and later, contains comments stating the hash functions are logically correct but that improvements may be possible and even desirable.

Recently, Chuck Lever has noted also that improvements may be possible [4]. He makes the case for the phi-based multiplicative overflow arithmetic hash functions by recalling Don Knuth's analysis [3]. Lever goes on to present some throughput comparisons for a small set of hash table configurations.

4 Conclusions

This work evaluated a set of hash functions that could be used for Linux dcache and inode cache. Our study shows that the current hash functions used in the circa 2000 Linux kernels can introduce significant data distribution skews. With better hash functions, data can be much more evenly spread over the hash chains, hence reducing the average chain walk time. Moreover, such hash function is computationally inexpensive on more recent computer architectures, such as the Intel Pentium class machines. Our results go beyond Lever's and suggest that even better hash functions, such as the hash phi function should be used in Linux inode cache and dcache.

References

[1] Standard Performance Evaluation Corporation, SFS 2.0, 1997. p. 77, Manasses, VA.

[2] T. Aivazian. Linux kernel internals. August 2000. www.RedHat.com/mirrors/LDP/guides.html.

[3] D.E. Knuth. *The Art of Computer Programming, Volume 3 / Sorting and Searching*. Addison-Wesley, Reading, second edition, 1998.

[4] C. Lever. Linux kernel hash table behavior: Analysis and improvements. In *Fourth Annual Linux Showcase (ALS'2000)*, October 10-14, 2000. www.citi.umich.edu/projects/linux-scalability.

[5] D. Moore. Rehashing Pearson's string hash. *C++ Report: letters section*, pages 6–15, February 1995.

[6] P.K. Pearson. Fast hashing of variable-length text strings. *Communications of the Association for Computing Machinery*, 33:677–680, June 1990.