Scalability of the Directory Entry Cache

Hanna Linder
IBM Linux Technology Center
hannal@us.ibm.com http://www.ibm.com/linux
Dipankar Sarma
IBM Linux Technology Center
dipankar@in.ibm.com http://www.ibm.com/linux
Maneesh Soni
IBM Linux Technology Center
maneesh@in.ibm.com http://www.ibm.com/linux

Abstract

This paper presents work that we have done to improve scalability of the directory entry cache (dcache). We investigated scalability problems resulting from many cache lookups, global lock contention, a possibly non-optimal eviction policy, and cacheline bouncing due to global reference counters. This paper provides an overview of solutions we tried, such as fast path walking, utilizing the read-copy update mutual exclusion mechanism[McKenney], and lazy LRU updates. We conclude with performance results showing scalability improvements.

1 Introduction

Every file and directory has a path. The path must be followed to do a lookup in the dcache to get the correct inode number of the dentry. A path such as /etc/passwd contains three dentries: '/', 'etc', and 'passwd'. Each dentry in a lookup path has a reference counter called d_count, which is atomically incremented and decremented as the dcache is being checked. This keeps the dentry from being put on the least recently used (LRU) list.

Currently, the dcache is protected by a single global lock, dcache_lock. This lock is held during lookup of dentries (d_lookup) as well as all manipulations of the dentry cache and the assorted lists that maintain hierarchies, aliases and LRU entries. The global dcache_lock seems to be an issue as the number of CPUs increase. We experimented with various ways to improve scaling the dentry cache.

2 Workload and Measures

We have used three main workloads for measuring scaling of the dentry cache: dbench[Pool] (with settings to avoid I/O) , httperf[Mosberger], profiles[Hawkes] of Linux(R) kernel compiles, and lockmeter[Hawkes]. The system used is an 8-way Pentium(R)-III Xeon(TM) with 1MB L2 cache and 2 GB of RAM (unless otherwise noted).

2.1 Summary of Baseline Measurements

The baseline measurements show that dcache_lock suffers from an increasing level of contention for some benchmarks. Although other locks such as the Big Kernel Lock (kernel_flag) and lru_list_lock are much higher in the total contention numbers, once those are dealt with, dcache_lock will move up the list.

The following work focuses on ways to increase scalability of the dcache. While looking at the

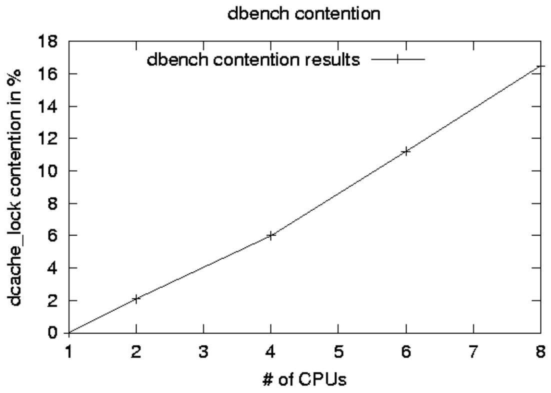

Figure 1: Baseline contention with dbench

distribution of lock acquisitions for these workloads, it becomes obvious that d_lookup() is the routine to optimize since it is the routine where the global lock is acquired most often.

2.2 Dbench Results of Baseline

The dbench results from our initial investigations [Sarma] show that lock utilization and contention grow steadily with an increasing number of CPUs. On an 8-way system running 2.4.16 kernel, dbench results show 5.3% utilization with 16.5% contention on this lock (see Figure 1).

One significant observation with the lockmeter output is that for this workload d_lookup() is the most common operation.

This snippet of lockmeter output for an 8-way shows that 84% of the time dcache_lock was acquired by d_lookup(). Out of about fifteen million holds of the dcache_lock, d_lookup() comprised twelve million of them. The simple explanation for this is that d_lookup is the main point into the dcache. It does the looping search to find the parent in the hash, then atomically increments the d_count reference of the dentry before returning it, all while the dcache_lock is held.

```
SPINLOCKS         HOLD            WAIT
  UTIL  CON    MEAN(  MAX )   MEAN(  MAX )(% CPU)     TOTAL SPIN  NAME
  5.3% 16.5%  0.6us(2787us)  5.0us(3094us)(0.89%)  15069563 16.5% dcache_lock
 0.01% 10.9%  0.2us( 7.5us)  5.3us( 116us)(0.00%)    119448 10.9%  d_alloc+0x128
 0.04% 14.2%  0.3us(  42us)  6.3us( 925us)(0.02%)    233290 14.2%  d_delete+0x10
 0.00%  3.5%  0.2us( 3.1us)  5.6us(  41us)(0.00%)      5050  3.5%  d_delete+0x94
 0.04% 10.9%  0.2us( 8.2us)  5.3us(1269us)(0.01%)    352739 10.9%  d_instantiate+0x1c
  4.8% 17.2%  0.7us(1362us)  4.8us(2692us)(0.76%)  12725262 17.2%  d_lookup+0x5c
 0.02% 11.0%  0.9us(  22us)  5.4us(1310us)(0.00%)     46800 11.0%  d_move+0x38
 0.01%  5.1%  0.2us(  37us)  4.2us(  84us)(0.00%)    119438  5.1%  d_rehash+0x40
 0.00%  2.5%  0.2us( 3.1us)  5.6us(  45us)(0.00%)      1680  2.5%  d_unhash+0x34
 0.31% 15.0%  0.4us(  64us)  6.2us(3094us)(0.09%)   1384623 15.0%  dput+0x30
 0.00% 0.82%  0.4us( 4.2us)  6.4us( 6.4us)(0.00%)       122 0.82%  link_path_walk+0x2a8
 0.00%    0%  1.7us( 1.8us)   0us                         2    0%  link_path_walk+0x618
 0.00%  6.4%  1.9us( 832us)  5.0us(  49us)(0.00%)      3630  6.4%  prune_dcache+0x14
 0.04%  9.4%  1.0us(1382us)  4.7us( 148us)(0.00%)     70974  9.4%  prune_dcache+0x138
 0.04%  4.2%   11us(2787us)  3.8us(  24us)(0.00%)      6505  4.2%  select_parent+0x20
```

2.3 Httperf Results of Baseline

The httperf results from our initial investigation show a moderate utilization of 6.2% with 4.3% contention in an 8 CPU environment.

A snippet of lockmeter output showing the distribution of acquisition of dcache_lock follows:

```
SPINLOCKS         HOLD            WAIT
  UTIL  CON    MEAN(  MAX )   MEAN(  MAX )(% CPU)     TOTAL SPIN   NAME
  6.2%  4.3%  0.8us( 390us)  2.7us( 579us)(0.12%)  20243025 4.3%  dcache_lock
 0.02%  6.5%  0.5us(  45us)  2.7us( 281us)(0.00%)    100031 6.5%   d_alloc+0x128
 0.01%  4.9%  0.2us( 4.6us)  2.9us(  58us)(0.00%)    100032 4.9%   d_instantiate+0x1c
  5.0%  4.5%  0.8us( 387us)  2.8us( 579us)(0.09%)  15009129 4.5%   d_lookup+0x5c
 0.02%  5.8%  0.6us(  34us)  3.1us(  45us)(0.00%)    100031 5.8%   d_rehash+0x40
 0.19%  8.8%  0.5us( 296us)  2.8us( 315us)(0.01%)    933218 8.8%   dput+0x30
 0.89%  2.3%  0.6us( 390us)  2.5us( 309us)(0.01%)   4000584 2.3%   link_path_walk+0x2a8
```

This shows that 74% of the time the global lock is acquired from d_lookup(). Again, out of about twenty million acquisitions of the dcache_lock, d_lookup took fifteen million of them.

3 Avoiding Global Lock in d_lookup()

In the paper by Paul E. McKenney, Dipankar Sarma, and Orran Krieger [McKenney] they described the Read Copy Update mutual exclusion mechanism (RCU). To summarize, RCU provides support for reading an item without holding a lock and a special callback method to update all references to the data when it is written.

The dcache_lock is held while traversing the d_hash list and while updating the Least Recently Used (LRU) list if the dentry found by d_lookup has a zero reference count. By using RCU we can avoid dcache_lock while reading d_hash list [1].

In this, we were able to do a d_hash lookup lock free but had to take the dcache_lock while updating the LRU list. The patch does provide some decrease in lock hold time and contention level. Following are lockmeter statistics for 2.4.16 without any patches while running dbench:

```
SPINLOCKS         HOLD            WAIT
  UTIL  CON    MEAN(  MAX )   MEAN(  MAX )(% CPU)     TOTAL SPIN   NAME
  6.3%  9.2%  0.4us(1659us)  3.4us(1648us)( 1.3%)  23182304  9.2% dcache_lock
 0.01% 10.1%  0.2us( 7.6us)  2.9us(  45us)(0.01%)     96649 10.1%   d_alloc+0x124
 0.03% 11.0%  0.2us(  70us)  2.9us( 316us)(0.01%)    184690 11.0%   d_delete+0x10
 0.04%  8.8%  0.2us(  95us)  2.7us( 175us)(0.01%)    281340  8.8%   d_instantiate+0x1c
  3.8% 12.7%  0.5us( 123us)  3.4us(1648us)(0.80%)  10074944 12.7%   d_lookup+0x58
 0.02%  9.9%  0.8us(  24us)  2.8us(  56us)(0.00%)     37050  9.9%   d_move+0x34
 0.01%  3.6%  0.2us(  32us)  3.4us(  58us)(0.00%)     96639  3.6%   d_rehash+0x3c
 0.00%  4.2%  0.2us( 1.5us)  2.7us( 9.4us)(0.00%)      1330  4.2%   d_unhash+0x34
  2.3%  6.4%  0.3us( 120us)  3.3us(1379us)(0.48%)  12336769  6.4%   dput+0x18
 0.00%  5.2%  2.0us( 882us)  3.9us(  50us)(0.00%)      3006  5.2%   prune_dcache+0x10
 0.02%  4.8%  6.1us( 836us)  3.2us(  23us)(0.00%)      5280  4.8%   select_parent+0x18
```

Following is the same dbench run with this first RCU patch applied:

```
SPINLOCKS         HOLD            WAIT
  UTIL  CON    MEAN(  MAX )   MEAN(  MAX )(% CPU)     TOTAL SPIN  NAME
  4.3%  7.5%  0.3us(1436us)  3.0us(1222us)(0.88%)  23103201  7.5% dcache_lock
 0.01%  5.6%  0.2us(  18us)  2.3us(  54us)(0.00%)    104404  5.6%   d_alloc+0x128
 0.03%  8.1%  0.2us(  20us)  2.4us( 322us)(0.01%)    184690  8.1%   d_delete+0x10
 0.04%  6.9%  0.2us(  30us)  2.2us(  79us)(0.01%)    289095  6.9%   d_instantiate+0x1c
  2.1% 10.6%  0.3us( 491us)  3.0us(1222us)(0.54%)   9961665 10.6%   d_lookup+0xd8
 0.02%  7.4%  0.7us( 4.8us)  2.3us( 209us)(0.00%)     37050  7.4%   d_move+0x34
 0.01%  3.4%  0.2us( 4.8us)  3.0us(  43us)(0.00%)    104394  3.4%   d_rehash+0x3c
 0.00%  2.5%  0.2us( 1.3us)  2.9us( 8.6us)(0.00%)      1330  2.5%   d_unhash+0x34
  2.0%  5.1%  0.2us( 108us)  3.0us(1080us)(0.32%)  12342240  5.1%   dput+0x18
 0.04%  3.2%  0.9us(1436us)  3.1us(  74us)(0.00%)     65770  3.2%   prune_dcache+0x140
 0.02%  4.1%  6.6us( 926us)  2.7us( 8.3us)(0.00%)      5275  4.1%   select_parent+0x18
```

Spinning on the dcache_lock via d_lookup went from 12.7% to 10.6%. This demonstrated that simply doing the lock-free lookup of the d_hash was not enough because d_lookup() also acquired the dcache_lock to update the LRU list if the newly found dentry previously had a zero reference count. This often was the case with the dbench workload, hence we ended up acquiring the lock after almost every lock-free lookup of the hash table in d_lookup().

From there we decided we needed to avoid acquiring dcache_lock so often. Therefore, we tried different algorithms to get rid of this lock from d_lookup(), such as a separate lock for the LRU list.

4 Separate Lock for the LRU List

The motivation behind having a separate lock for the d_lru list was that as d_lookup() only updates the LRU list, we could relax contention on the dcache_lock by introducing a separate lock for LRU lists. This resulted in most of the load being transferred to the LRU list lock. Many routines held the dcache_lock as well, such as prune_dcache, select_parent, d_prune_aliases, because they read or write other lists apart from the LRU list [2].

```
SPINLOCKS         HOLD            WAIT
  UTIL  CON    MEAN(  MAX )   MEAN(  MAX )(% CPU)     TOTAL SPIN   NAME
  3.7%  5.7%  0.3us(1475us)  3.0us(1551us)(0.63%)  22434872  5.7% d_lru_lock
  1.7%  7.9%  0.3us(  90us)  3.1us(1489us)(0.39%)   9956382  7.9%   d_lookup+0xc8
  2.0%  3.9%  0.2us( 144us)  3.0us(1551us)(0.23%)  12346145  3.9%   dput+0x18
 0.04%  2.8%  0.5us(  22us)  3.5us(  79us)(0.00%)    127045  2.8%   prune_dcache+0x150
 0.03%  3.6%  9.2us(1475us)  3.1us( 112us)(0.00%)      5300  3.6%   select_parent+0x18
 0.26% 0.14%  0.2us(1474us)  1.7us( 204us)(0.00%)   1915750 0.14% dcache_lock
 0.01% 0.51%  0.1us( 1.9us)  1.8us( 204us)(0.00%)    109702 0.51%   d_alloc+0x124
 0.02% 0.15%  0.2us( 9.3us)  2.6us( 169us)(0.00%)    184690 0.15%   d_delete+0x10
 0.03% 0.16%  0.1us(  11us)  1.6us(  57us)(0.00%)    294393 0.16%   d_instantiate+0x1c
 0.02% 0.12%  0.7us(  27us)  1.3us( 5.5us)(0.00%)     37050 0.12%   d_move+0x34
 0.01% 0.12%  0.1us(  61us)  1.7us( 3.5us)(0.00%)    109692 0.12%   d_rehash+0x3c
 0.00% 0.23%  0.1us( 1.6us)  1.1us( 1.7us)(0.00%)      1330 0.23%   d_unhash+0x34
 0.14% 0.09%  0.2us(  38us)  1.5us( 141us)(0.00%)   1099648 0.09%   dput+0x4c
 0.01% 0.26%  0.2us(  18us)  1.4us( 5.5us)(0.00%)     69655 0.26%   prune_dcache+0x7c
 0.03% 0.26%  8.7us(1474us)  1.2us( 2.6us)(0.00%)      5300 0.26%   select_parent+0x24
```

5 Lazy Updating of the LRU List

Given that lock-free traversal of hash chains did not significantly decrease dcache_lock acquisitions, we looked at the possibility of removing dcache_lock acquisitions completely from d_lookup(). After using RCU based lock-free hash lookup, the only remaining use of the dcache_lock in d_lookup() was to update the LRU list.

Our next approach was to relax the rules of an LRU list by allowing dentries with non-zero reference counts to remain in the list for a short delay before being removed in the update [3]. The beneficial side-effect was that multiple dentries could be processed during the update. Previously, the global dcache_lock was held then dropped for every single entry as each dentry was removed from the list during the update.

To implement this new functionality, we introduced another flag (DCACHE_DEFERRED_FREE) and a per-dentry lock (d_lock) in struct dentry to maintain consistency between the flag and the reference counter (d_count). For all other lists in struct dentry, the reference counter continued to provide mutual exclusion.

Allowing additional dentries to remain in the lru_list could lead to an unusually large number of dentries, causing a lengthy deletion process during updates. We proposed two different approaches to circumvent this problem:
1. Use a timer to kick off periodic updates.
2. Periodically update the d_lru list while already traversing it.

5.1 Timer Based Lazy Updating

A timer was used to remove the referenced dentries from the d_lru list so that it would be kept manageable. To take the dcache_lock from the timer handler we had to use spin_lock_bh() and spin_unlock_bh() for dcache_lock. This created problems with cyclic dependencies in dcache.h.

This approach did not prove to be any better than the non-timer approach. However, the patch is worth looking at as proper tuning of timer frequency may give better results [4].

5.2 Periodic Updates During Traversal

The d_lru list is made up to date through select_parent, prune_dcache and dput. While traversing the d_lru list in these routines, the dentries with non-zero reference counts are removed. This is the solution we chose to include in the lazy LRU patches due to its simplicity.

5.3 Notes on Lazy LRU Implementation

Per dentry lock(d_lock) is needed to protect the d_vfs_flags and d_count in d_lookup. There is very little contention on the per dentry lock, so this will not lead to a bottleneck. With this patch the DCACHE_REFERENCED flag does more work. It is being used to indicate the dentries which are not supposed to be on the d_lru list. Right now apart from d_lookup, the per dentry lock (d_lock) is used whereever d_count or d_vfs_flags are read or modified. It is probably possible to tune the code more and relax the locking in some cases.

We have created a new function include/linux/dcache.h: d_unhash() that sets the DCACHE_DEFERRED_FREE bit in d_vfs_flags, which marks the dentry for deferred freeing. Also, before unlinking the dentry from the d_hash list we have to update the d_nexthash pointer. We changed the name for fs/namei.c: d_unhash() to

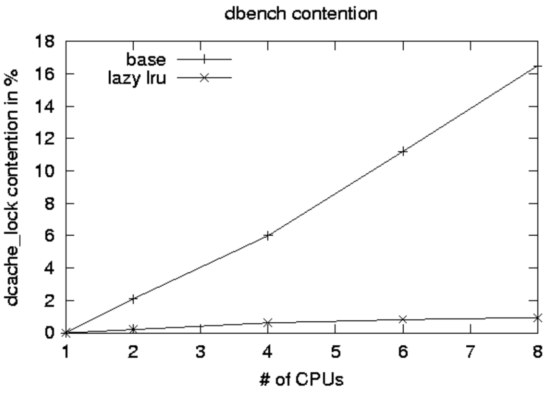

Figure 2: Lazy LRU contention from dbench

fs/namei.c: d_vfs_unhash().

As we do lockless lookup, rmb() is used in d_lookup to avoid out of order reads for d_nexthash and wmb() is used in d_unhash to make sure that d_vfs_flags and d_nexthash() are updated before unlinking the dentry from the d_hash chain.

Every dget() marks the dentry as referenced by setting DCACHE_REFERENCED bit in d_vfs_flags. This forced us to hold the per dentry lock in dget. Therefore, dget_locked is not needed.

5.4 Lazy LRU Patch Results

Contention for the dcache_lock reduced in all routines. However, the routines: prune_dcache and select_parent take more time because the d_lru list is longer. This is acceptable as both routines are not in the critical path.

We ran dbench and httperf to measure the effect of lazy dcache and the results were very good. By doing a lock-free d_lookup(), we were able to substantially cut down on the number of dcache_lock acquisitions. This resulted in substantially decreased contention as well as lock utilizations.

```
SPINLOCKS         HOLD            WAIT
  UTIL  CON    MEAN(  MAX )   MEAN(  MAX )(% CPU)     TOTAL SPIN   NAME
 0.89% 0.95%  0.6us(6516us)   19us(6411us)(0.03%)   2330127 0.95% dcache_lock
 0.02%  1.7%  0.2us(  20us)   17us(2019us)(0.00%)    116150  1.7%  d_alloc+0x144
 0.03% 0.42%  0.2us(  49us)   35us(6033us)(0.00%)    233290 0.42%  d_delete+0x10
```

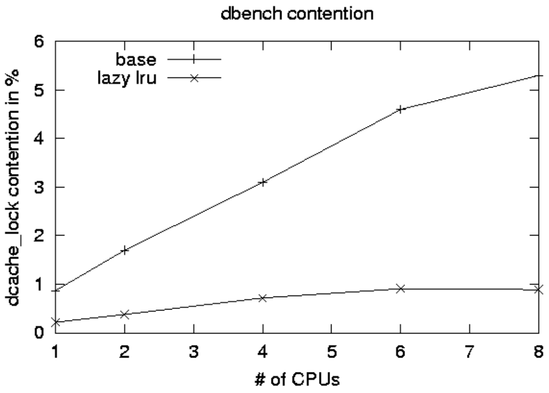

Figure 3: Lazy LRU dcache_lock utilization from dbench

```
 0.00% 0.14%  0.8us(  12us)  3.4us( 8.5us)(0.00%)      5050 0.14%  d_delete+0x98
 0.03% 0.40%  0.1us(  32us)   34us(5251us)(0.00%)    349441 0.40%  d_instantiate+0x1c
 0.05% 0.30%  1.7us(  44us)   22us(1770us)(0.00%)     46800 0.30%  d_move+0x38
 0.01% 0.16%  0.1us(  21us)  4.5us( 334us)(0.00%)    116140 0.16%  d_rehash+0x40
 0.00% 0.65%  0.7us( 3.7us)  8.4us(  57us)(0.00%)      1680 0.65%  d_vfs_unhash+0x44
 0.56%  1.1%  0.7us(  84us)   18us(6411us)(0.02%)   1383859  1.1%  dput+0x30
 0.00% 0.88%  0.4us( 2.3us)  1.3us( 1.3us)(0.00%)       114 0.88%  link_path_walk+0x2d8
 0.01%  4.4%  4.3us(6516us)  4.8us(  32us)(0.00%)      3566  4.4%  prune_dcache+0x14
 0.07%  2.3%  1.8us(6289us)  4.4us( 718us)(0.00%)     67591  2.3%  prune_dcache+0x150
 0.11% 0.79%   29us(4992us)   28us(1116us)(0.00%)      6444 0.79%  select_parent+0x24
```

5.5 Dbench Results of Lazy LRU

dbench results showed that lock utilization and contention levels remain flat with lazy dcache as opposed to steadily increasing with the baseline kernel. So for 8 processors, contention level is 0.95% as opposed to 16.5% for the baseline (2.4.16) kernel.

One significant observation is that maximum lock hold time for prune_dcache() and select_parent() are high for this algorithm. However, these are not frequent operations for this workload.

A comparison of baseline (2.4.16) kernel and lazy dcache contention and utilization while running dbench can be seen in Figures 2 and 3.

The throughput results show marginal differences (statistically insignificant) for up to four CPUs, of 1% (statistically significant) on eight CPUs. There is no performance regression in the lower end and the gains are small in the higher end.

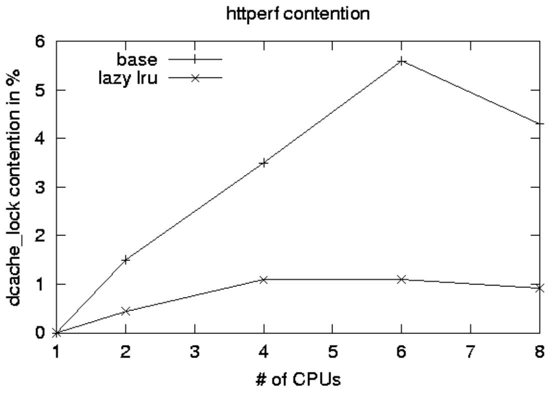

Figure 4: Lazy LRU contention from httperf

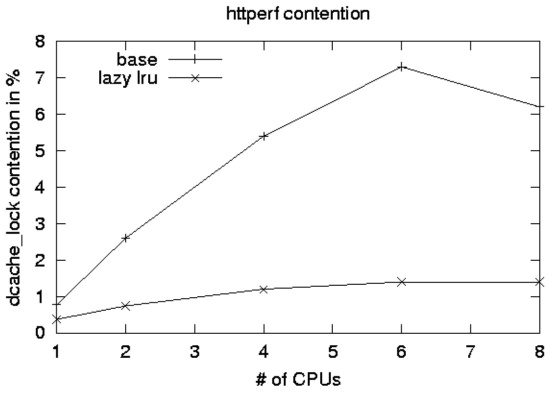

Figure 5: Lazy LRU dcache_lock utilization from httperf

5.6 Httperf Results of Lazy LRU

The httperf results showed a similar decrease in lock contention and lock utilization. With 8 CPUs, it showed significantly less contention.

```
SPINLOCKS         HOLD            WAIT
  UTIL  CON    MEAN(  MAX )   MEAN(  MAX )(% CPU)     TOTAL  SPIN   NAME
  1.4% 0.92%  0.7us( 577us)  2.2us( 617us)(0.00%)   4821866 0.92%  dcache_lock
 0.02%  2.2%  0.6us(  30us)  1.9us( 7.8us)(0.00%)    100031  2.2%   d_alloc+0x144
 0.01%  1.7%  0.2us(  12us)  2.2us( 9.2us)(0.00%)    100032  1.7%   d_instantiate+0x1c
 0.03%  1.5%  0.7us( 9.2us)  2.3us(  10us)(0.00%)    100031  1.5%   d_rehash+0x40
 0.24%  2.1%  1.2us( 577us)  1.9us( 283us)(0.00%)    521329  2.1%   dput+0x30
  1.1% 0.70%  0.7us( 366us)  2.4us( 617us)(0.00%)   4000443 0.70%   link_path_walk+0x2d8
```

A comparison of the baseline (2.4.16) kernel and lazy dcache contention and utilization while running dbench can be seen in Figures 4 and 5.

The results of httperf (replies/sec for fixed connection rate) showed statisticially insignificant differences between base 2.4.16 and lazy dcache kernels.

6 Avoiding Cacheline Bouncing of d_count

6.1 fast_walk()

On SMP systems and even moreso on some NUMA architectures, repeated operations on the same global variable can cause excessive cacheline bouncing. This is due to the entire cacheline being read into each CPU's hardware cache while it is being used. For some common directories found in many paths such as '/' or 'usr', this exessive cacheline bouncing will be triggered.

Alexander Viro recommended a possible solution that we implemented. He proposed not incrementing and decrementing the reference counter for dentries that are already in the dentry cache. Instead, hold the dcache_lock to keep them from being deleted.

We used the path_lookup function to implement this change [5]:

```
Before:
                read_lock(&current->fs->lock);
                nd->mnt = mntget(current->fs->pwdmnt);
                nd->dentry = dget(current->fs->pwd);
                read_unlock(&current->fs->lock);
        }
        return (path_walk(name, nd));

After:
                read_lock(&current->fs->lock);
                spin_lock(&dcache_lock);
                nd->mnt = current->fs->pwdmnt;
                nd->dentry = current->fs->pwd;
                read_unlock(&current->fs->lock);
        }
        nd->flags |= LOOKUP_LOCKED;
        return (path_walk(name, nd));
```

The atomic increment of d_count is all that dget and mntget do.

The rest of the changes were in path_walk (implemented by link_path_walk). While the dentry is found in the cache, just keep walking the path. If a dentry is not in the cache, then increment the d_count to keep it synchronized and drop the dcache_lock, and then simply continue. For coding simplicity, the dcache_lock is always dropped in the path_walk code instead of returned to path_lookup to be dropped. This patch has been accepted by Linus Torvalds starting with the 2.5.11 kernel.

6.2 path_lookup()

We started with a simple cleanup of replicated code involving path_init, path_walk, and __user_walk[6]. There were sixteen occurrences of the following:

```
        if(path_init(x))
        error = path_walk(x)
Which changed to one call:
        error = path_lookup(x)
In addition there were six occurances
of the following:
        a = getname(b)
        if(error)
                return
        path_lookup(a)
        putname(a)
which changed to an existing call:
        error = __user_walk(b)
```

This patch has been accepted by Alan Cox starting in 2.4.19-pre5-ac2. Marcelo has not merged this patch into mainline 2.4 as of this writing.

6.3 Fast Path Walking Results

6.4 16-way NUMA Results of Fast Walk

Previously, we mentioned d_lookup was the main user of dcache_lock. This is especially noticeable on a 16-way NUMA system. Martin Bligh, in attempting to get the fastest kernel compile, applied this patch on top of a few others [Bligh]. Not only did it reduce time spent spinning on the dcache_lock, it decreased total kernel compile time by 2.5%.

Following is a profile of kernel during make -j32 bzImage on a 16-way NUMA system. This shows

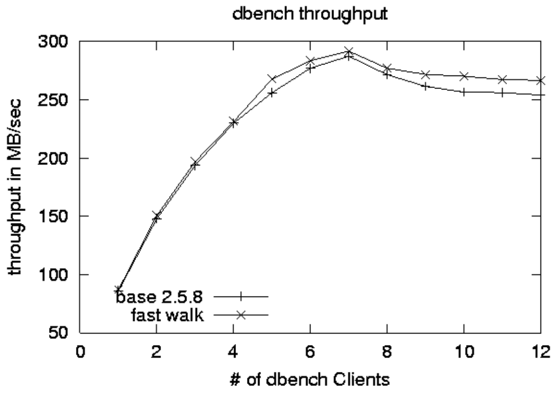

Figure 6: FastWalk increases dbench throughput

an almost 50% reduction in time spinning on the dcache_lock.

```
Kernel compile time is now 23.6 seconds.

Here are the top 10 elements of profile before and after your
patch (left hand column is the number of ticks spent in each
function).

Before:

 22086 total                                          0.0236
  9953 default_idle                                 191.4038
  2874 _text_lock_swap                               53.2222
  1616 _text_lock_dcache                              4.6304
   748 lru_cache_add                                  8.1304
   605 d_lookup                                       2.1920
   576 do_anonymous_page                              1.7349
   511 do_generic_file_read                           0.4595
   484 lru_cache_del                                 22.0000
   449 __free_pages_ok                                0.8569
   307 atomic_dec_and_lock                            4.2639

After:

 21439 total                                          0.0228
  9112 default_idle                                 175.2308
  3364 _text_lock_swap                               62.2963
   790 lru_cache_add                                  8.5870
   750 _text_lock_namei                               0.7184
   587 do_anonymous_page                              1.7681
   572 lru_cache_del                                 26.0000
   569 do_generic_file_read                           0.5117
   510 __free_pages_ok                                0.9733
   421 _text_lock_dec_and_lock                       17.5417
   318 _text_lock_read_write                          2.6949

   ...

   129 _text_lock_dcache                              0.3696
```

7 Conclusions

This paper has demonstrated performance improvements of the dcache via the fast path walking patches and the lazy updating of the LRU patches. We are working with the VFS and kernel maintainers to get these patches accepted.

Although the dcache continues to scale, there is more work to be done, much of it happening as this is being written.

8 Availability of Referenced Patches

As of now, all patches have been tested on ext2, ext3, JFS, and /proc filesystem. Our goal was to experiment with dcache, extending it for use with other filesystems, this is in the pipleline.

dcache patches can be found on SourceForge.net under the Linux Scalability Effort project page.

[1] lockfree read of d_hash
http://prdownloads.sf.net/lse/dcache_rcu-2.4.10-01.patch

[2] separate lock for the lru list
http://prdownloads.sf.net/lse/dcache_rcu-lru_lock-2.4.16-02.patch

[3] Lazy LRU
http://prdownloads.sf.net/lse/dcache_rcu-lazy_lru-2.4.17-06.patch

[4] Lazy LRU updating via timer
http://prdownloads.sf.net/lse/dcache_rcu-lazy_lru-timer-2.4.16-04.patch

[5] Fast Path Walking
http://prdownloads.sf.net/lse/fast_walkA1-2.5.10.patch

[6] Path walking code cleanup
http://prdownloads.sf.net/lse/path_lookupA1-2.4.17.patch

9 Acknowledgments

Mr. Alexander Viro has been a tremendous help to us and we thank him for his input and all his hard work.
SourceForge.net for supporting Open Source development.
Paul Menage for helping to debug.
Martin Bligh for running the NUMA tests.
Hans-Joachim Tannenberger, our manager.
International Business Machines Inc. and the Linux Technology Center.

References

[Sarma] Dipankar Sarma, Maneesh Soni *Scaling the dentry cache* http://lse.sf.net/locking/dcache/dcache.html

[McKenney] Paul E. McKenney, Dipankar Sarma, and Orran Krieger, *Read-Copy Update*

[Mosberger] David Mosberger, Tai Lin, *httperf: A tool for measuring web server performance.* Hewlett-Packard Inc. Research Labs. http://www.hpl.hp.com/personal/David_Mosberger/httperf.html

[Hawkes] John Hawkes *kernprof* Silicon Graphics Inc. http://oss.sgi.com/projects/kernprof

[Hawkes] John Hawkes *lockmeter* Silicon Graphics Inc. http://oss.sgi.com/projects/lockmeter

[Pool] Martin Pool *dbench* Samba.org

[Bligh] Martin J. Bligh's *23 second kernel compile (aka which patches help scalibility on NUMA)*, linux-kernel@vger.kernel.org, March 8, 2002. http://marc.theaimsgroup.com/?l=linux-kernel&m=101565828617899&w=2.

10 Tradmarks