

---

Subject: [RFC][PATCH 1/2] r/o bind mounts: NUMA-friendly writer count  
Posted by [Dave Hansen](#) on Wed, 21 Feb 2007 02:03:52 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

I've been working on the read-only bind mount patches, and one of their requirements is that we track the number of writers to a particular filesystem. This allows us to quickly determine whether it is OK to make rw->ro transitions.

It was noted that the previous approach of using a spinlock to protect the per-mount write count caused some pretty serious scaling problems on NUMA machines. The same kinds of problems would very likely occur if we just used atomic\_ts as well.

This patch should take global locks out of the fast path acquiring and dropping writes for mounts. It uses per-cpu atomic\_ts (it could be per-node, but we don't have a nice alloc\_pernode()) to track writers.

All cpus start out in a "denied write" state, and must grab a spinlock and set it bit before they can go into the "allowed to write" state. They stay in this state until somebody goes and tries to remount the mount read-only, which also requires the spinlock.

There is also a slow path in the mnt\_drop\_write() case. If a write is acquired on one cpu, then dropped on another, the write count could be imbalanced. So, the code grabs the spinlock, and goes looking for another cpu's writecount to decrement. During a kernel-compile on a 4-way non-NUMA machine, these "misses" happened about 400 times, but all from \_\_fput(). The next patch will show a little hack to greatly reduce their frequency.

Note that these apply on top of the r/o bind mount patches that I have. If anyone wants to actually try them, I'll send you the entire set.

---

```
lxc-dave/fs/namespace.c      | 118 ++++++-----  
lxc-dave/include/linux/mount.h | 10 +++  
2 files changed, 125 insertions(+), 3 deletions(-)
```

```
diff -puN fs/namespace.c~numa_mnt_want_write fs/namespace.c
```

```

--- lxc/fs/namespace.c~numa_mnt_want_write 2007-02-20 17:50:32.000000000 -0800
+++ lxc-dave/fs/namespace.c 2007-02-20 17:58:54.000000000 -0800
@@ -51,8 +51,11 @@ static inline unsigned long hash(struct
    return tmp & hash_mask;
}

+static int MNT_DENIED_WRITE = -1;
+
+struct vfsmount *alloc_vfsmnt(const char *name)
+{
+ int cpu;
+ struct vfsmount *mnt = kmem_cache_zalloc(mnt_cache, GFP_KERNEL);
+ if (mnt) {
+     atomic_set(&mnt->mnt_count, 1);
@@ -64,6 +67,13 @@ struct vfsmount *alloc_vfsmnt(const char
    INIT_LIST_HEAD(&mnt->mnt_share);
    INIT_LIST_HEAD(&mnt->mnt_slave_list);
    INIT_LIST_HEAD(&mnt->mnt_slave);
+ mnt->writers = alloc_percpu(atomic_t);
+ if (!mnt->writers) {
+     kmem_cache_free(mnt_cache, mnt);
+     return NULL;
+ }
+ for_each_possible_cpu(cpu)
+     atomic_set(per_cpu_ptr(mnt->writers, cpu), -1);
+ if (name) {
+     int size = strlen(name) + 1;
+     char *newname = kmalloc(size, GFP_KERNEL);
@@ -78,17 +88,118 @@ struct vfsmount *alloc_vfsmnt(const char

int mnt_want_write(struct vfsmount *mnt)
{
- if (__mnt_is_readonly(mnt))
-     return -EROFS;
- return 0;
+ int ret = 0;
+ atomic_t *cpu_writecount;
+ int cpu = get_cpu();
+retry:
+ /*
+  * Not strictly required, but quick and cheap
+  */
+ if (__mnt_is_readonly(mnt)) {
+     ret = -EROFS;
+     goto out;
+ }
+ cpu_writecount = per_cpu_ptr(mnt->writers, cpu);
+ if (atomic_add_unless(cpu_writecount, 1, MNT_DENIED_WRITE))

```

```

+ goto out;
+ spin_lock(&vfsmount_lock);
+ if (cpu_isset(cpu, mnt->cpus_that_might_write)) {
+ /*
+  * Somebody is attempting to deny writers to this
+  * mnt and we raced with them.
+  */
+ spin_unlock(&vfsmount_lock);
+ goto retry;
+ }
+ cpu_set(cpu, mnt->cpus_that_might_write);
+ /*
+  * actually allow the cpu to get writes
+  */
+ atomic_set(cpu_writecount, 0);
+ spin_unlock(&vfsmount_lock);
+ goto retry;
+out:
+ put_cpu();
+ return ret;
+ }
EXPORT_SYMBOL_GPL(mnt_want_write);

void mnt_drop_write(struct vfsmount *mnt)
{
+ static int miss = 0;
+ atomic_t *cpu_writecount;
+ int cpu;
+ int borrowed = 0;
+ int retries = 0;
+retry:
+ cpu = get_cpu();
+ cpu_writecount = per_cpu_ptr(mnt->writers, cpu);
+ if (atomic_add_unless(cpu_writecount, -1, 0)) {
+ put_cpu();
+ return;
+ }
+ spin_lock(&vfsmount_lock);
+ /*
+  * Holding the spinlock, and only checking cpus that
+  * have cpus_that_might_write set means that we should
+  * only be checking values that are positive here.
+  *
+  * The spinlock won't help us catch an elevated
+  * write count on the first run through because other
+  * cpus are free to do inc/dec without taking that lock
+  * We might have to try this loop more than once.
+  */

```

```

+ for_each_cpu_mask(cpu, mnt->cpus_that_might_write) {
+   cpu_writecount = per_cpu_ptr(mnt->writers, cpu);
+   WARN_ON(atomic_read(cpu_writecount) < 0);
+   if (atomic_add_unless(cpu_writecount, -1, 0)) {
+       borrowed = 1;
+       break;
+   }
+ }
+ spin_unlock(&vfsmount_lock);
+ miss++;
+ retries++;
+ if (printk_ratelimit()) {
+   printk("%s() retries: %d misses: %d\n", __func__, retries, miss);
+   dump_stack();
+ }
+ if (!borrowed)
+   goto retry;
+ }
EXPORT_SYMBOL_GPL(mnt_drop_write);

+/*
+ * Must hold vfsmount_lock
+ */
+int __mnt_deny_writers(struct vfsmount *mnt)
+{
+   int ret = 0;
+   int cpu;
+   for_each_cpu_mask(cpu, mnt->cpus_that_might_write) {
+       atomic_t *cpu_writecount = per_cpu_ptr(mnt->writers, cpu);
+       /*
+        * This could leave us with a temporarily
+        * over-decremented cpu_writecount.
+        *
+        * mnt_drop_write() is OK with this because
+        * it will just force it into the slow path.
+        *
+        * The only users who care about it being
+        * decremented either hold vfsmount_lock
+        * to look at it.
+        */
+       if (atomic_dec_return(cpu_writecount) != MNT_DENIED_WRITE) {
+           atomic_inc(cpu_writecount);
+           ret = -EBUSY;
+           break;
+       }
+       cpu_clear(cpu, mnt->cpus_that_might_write);
+   }
+ }

```

```

+ return ret;
+}
+
void add_mount_to_sb_list(struct vfsmount *mnt, struct super_block *sb)
{
    spin_lock(&vfsmount_lock);
@@ -113,6 +224,7 @@ void free_vfsmnt(struct vfsmount *mnt)
    list_del(&mnt->mnt_sb_list);
    spin_unlock(&vfsmount_lock);
    kfree(mnt->mnt_devname);
+ free_percpu(mnt->writers);
    kmem_cache_free(mnt_cache, mnt);
}

```

```

diff -puN include/linux/mount.h~numa_mnt_want_write include/linux/mount.h
--- lxc/include/linux/mount.h~numa_mnt_want_write 2007-02-20 17:50:32.000000000 -0800
+++ lxc-dave/include/linux/mount.h 2007-02-20 17:53:28.000000000 -0800
@@ -62,6 +62,16 @@ struct vfsmount {
    atomic_t mnt_count;
    int mnt_expiry_mark; /* true if marked for expiry */
    int mnt_pinned;
+ /*
+  * These are per-cpu, but should be per-NUMA node.
+  * >0 - has an active writer
+  * 0 - has no active writers, but doesn't need to set
+  *    cpus_that_might_write before getting one
+  * -1 - has no active writers, and must set its bit
+  *    in cpus_that_might_write before going to 0
+  */
+ atomic_t *writers;
+ cpumask_t cpus_that_might_write;
};

```

```

static inline struct vfsmount *mntget(struct vfsmount *mnt)

```

---

Containers mailing list  
Containers@lists.osdl.org  
<https://lists.osdl.org/mailman/listinfo/containers>

---