Issues with Selected Scalability Features of the 2.6 Kernel

Dipankar Sarma
Linux Technology Center
IBM India Software Lab
dipankar@in.ibm.com

Paul E. McKenney
Linux Technology Center and Storage Software Architecture
IBM Beaverton
paulmck@us.ibm.com

Abstract

The 2.6 Linux™ kernel has a number of features that improve performance on high-end SMP and NUMA systems. Finer-grain locking is used in the scheduler, the block I/O layer, hardware and software interrupts, memory management, and the VFS layer. In addition, 2.6 brings new primitives such as RCU and per-cpu data, lock-free algorithms for route cache and directory entry cache as well as scalable user-level APIs like `sys_epoll()` and futexes. With the widespread testing of these features of the 2.6 kernel, a number of new issues have come to light that needs careful analysis. Some of these issues encountered thus far are: overhead of multilple lock acquisitions and atomic operations in critical paths, possibility of denial-of-service attack on subsystems that use RCU-based deferred free algorithms and degradation of realtime response due to increased softirq load.

In this paper, we analyse a select set of these issues, present the results, workaround patches and future courses of action. We also discuss applicability of some these issues in new features being planned for 2.7 kernel.

1 Introduction

Support for symmetric multi-processing (SMP) in the Linux kernel was first introduced in 2.0 kernel. The 2.0 kernel had a single `kernel_flag` lock AKA Big Kernel Lock (BKL) which essentially single threaded almost all of the kernel [Love04a]. The 2.2 kernel saw the introduction of finer-grain locking in several areas including signal handling, interrupts and part of I/O subsystem. This trend continued in 2.4 kernel.

A number of significant changes were introduced in during the development of the 2.6 kernel that helped boost performance of many workloads. Some of the key components of the kernel were changed to have finer-grain locking. For example, the global `runqueue_lock` lock was replaced by the locks on the new per-cpu runqueues. Gone was `io_request_lock` with the introduction of the new scalable `bio`-based block I/O subsystem. BKL was peeled off from additional commonly used paths. Use of data locking instead of code locking became more widespread. In addition, Read-Copy Update(RCU) [McK98a, McK01a] allowed further optimization of critical sections by avoiding locking while reading data structures which are updated less often. RCU enabled lock-free lookup of the directory-entry cache and route cache, which provided considerable performance benefits [Linder02a, Blanchard02a, McK02a]. While these improvements targeted high-end SMP and NUMA systems, the vast majority of the Linux-based systems in the computing world are small uniprocessor or low-end SMP systems that remain the main focus of the Linux kernel. Therefore, scalability enhancements must not cause any performance regressions in these smaller systems, and appropriate regression testing is required [Sarma02a]. This effort continues and has since thrown light on interesting issues which we discuss here.

Also, since the release of the 2.6 kernel, its adoption in many different types of systems has called attention to some interesting issues. Section 2 describes the 2.6 kernel's use of fine-grained locking and identifies opportunities in this area for the 2.7 kernel development effort. Section 3 discusses one such important issue that surfaced during Robert Olsson's router DoS testing. Section 4 discusses another issue important for real-time systems or systems that run interactive applications. Section 5 explores the impact of such issues and their workarounds on new experiments planned during the development of 2.7 kernel.

2 Use of Fine-Grain Locking

Since the support for SMP was introduced in the 2.0 Linux kernel, granularity of locking has gradually changed toward finer critical sections. In 2.4 and subsequently 2.6 kernel, many of the global locks were broken up to improve scalability of the kernel. Another scalability improvement was the use of reference counting in protecting kernel objects. This allowed us to avoid long critical sections. While these features benefit large SMP and NUMA systems, on smaller systems, benefit due to reduction of lock contention is minimal. There, the cost of locking due to atomic operations involved needs to be carefully evaluated. Table 1 shows cost of atomic operations on a 700MHz Pentium™ III Xeon™ processor. The cost of atomic increment is more than 4 times the cost of an L2 hit. In this section, we discuss some side effects of such finer-grain locking and possible remedies.

Operation	Cost (ns)
Instruction	0.7
Clock Cycle	1.4
L2 Cache Hit	12.9
Atomic Increment	58.2
cmpxchg Atomic Increment	107.3
Atomic Incr. Cache Transfer	113.2
Main Memory	162.4
CPU-Local Lock	163.7
cmpxchg Blind Cache Transfer	170.4
cmpxchg Cache Transfer and Invalidate	360.9

Table 1: 700 MHz P-III Operation Costs

2.1 Multiple Lock Acquisitions in Hot Path

Since many layers in the kernel use their own locks to protect their data structures, we did a simple instrumentation (Figure 1) to see how many locks we acquire on common paths. This counted locks in all variations of spinlock and rwlock. We used a running counter which we can read using a system call `get_lcount()`. This counts only locks acquired by the task in non-interrupt context.

With this instrumented kernel, we measured writing 4096 byte buffers to a file on ext3 filesystem. Figure 2 shows the test code

```
+static inline void _count_lock(void)
+{
+    if ((preempt_count() & 0x00ffff00) == 0) {
+            current_thread_info()->lcount++;
+    }
+}

....

 #define spin_lock(lock)          \
 do { \
+    _count_lock(); \
     preempt_disable(); \
     _raw_spin_lock(lock); \
 } while(0)
```

Figure 1: Lock Counting Code

```
if (get_lcount(&lcount1) != 0) {
        perror("get_lcount 1 failed\n");
        exit(-1);
}
write(fd, buf, 4096);
if (get_lcount(&lcount2) != 0) {
        perror("get_lcount 2 failed\n");
        exit(-1);
}
```

Figure 2: Lock Counting Test Code

that reads the lock count before and after the `write()` system call.

4K Buffer	Locks Acquired
0	19
1	11
2	10
3	11
4	10
5	10
6	10
7	10
8	16
9	10
Average	11.7

Table 2: Locks acquired during 4K writes

Table 2 shows the number of locks acquired during each 4K write measured on a 2-way Pentinum IV HT system running 2.6.0 kernel. The first write has a lock acquisition count of 19 and an average of 11.7 lock round-trips per 4K write. This does not count locks associated with I/O completion handling which is done from interrupt context. While this indicates scalability of the code, we still need to analyze this to see which locks are acquired in such hot path and check if very small adjacent critical sections can be collapsed into one. The modular nature of some the kernel layers may however make that impossible without affecting readability of code.

```
struct file * fget(unsigned int fd)
{
        struct file * file;
        struct files_struct *files =
                        current->files;

        read_lock(&files->file_lock);
        file = fcheck(fd);
        if (file)
                get_file(file);
        read_unlock(&files->file_lock);
        return file;
}
```

Figure 3: fget() Implementation

2.2 Refcounting in Hot Path

As described in Section 2.1, atomic operations can be costly. In this section, we discuss such an issue that was addressed during the development of the 2.6 kernel. Andrew Morton [Morton03a] pointed out that in 2.5.65-mm4 kernel, CPU cost of writing a large amount of small chunks of data to an ext2 file is quite high on uniprocessor systems and takes nearly twice again as long on SMP. It also showed that a large amount of overheads there were coming from `fget()` and `fput()` routines. A further look at Figure 3 shows how `fget()` was implemented in 2.5.65 kernel.

Both `read_lock()` and `read_unlock()` involve expensive atomic operations. So, even if there is no contention for `->file_lock`, the atomic operations hurt performance [McKenney03a]. Since most programs do not share their file-descriptor tables, the reader-writer lock is usually not really necessary. The lock need only be acquired when the reference count of the `file` structure indicates sharing. We optimized this as shown in Fig-

```
struct file *fget_light(unsigned int fd,
        int *fput_needed)
{
    struct file *file;
    struct files_struct *files = current->files;

    *fput_needed = 0;
    if (likely((atomic_read(&files->count)
                                == 1))) {
        file = fcheck(fd);
    } else {
        read_lock(&files->file_lock);
        file = fcheck(fd);
        if (file) {
            get_file(file);
            *fput_needed = 1;
        }
        read_unlock(&files->file_lock);
    }
    return file;
}
```

Figure 4: fget_light() Implementation

ure 4.

By optimizing the fast path to avoid atomic operation, we reduced the system time use by 11.2% in a UP kernel while running Andrew Morton's micro-benchmark with the command `dd if=/dev/zero of=foo bs=1 count=1M`. The complete results measured in a 4-CPU 700MHz Pentium III Xeon system with 1MB L2 cache and 512MB RAM is shown in Table 3

Kernel	sys time	Std Dev
2.5.66 UP	2.104	0.028
2.5.66-file UP	1.867	0.023
2.5.66 SMP	2.976	0.019
2.5.66-file SMP	2.719	0.026

Table 3: fget_light() results

However, the reader-writer lock must still be acquired in `fget_light()` fast path when the file descriptor table is shared. This is now being further optimized using RCU to make the file descriptor lookup fast path completely lock-free. Optimizing file descriptor look-up in shared file descriptor table will improve performance of multi-threaded applications that do a lot of I/Os. Techniques such as this are extremely useful for improving performance in both low-end and high-end SMP systems.

```
static __inline__ void rt_free(
        struct rtable *rt)
{
    call_rcu(&rt->u.dst.rcu_head,
      (void (*)(void *))dst_free,
     &rt->u.dst);
}

static __inline__ void rt_drop(
            struct rtable *rt)
{
    ip_rt_put(rt);
    call_rcu(&rt->u.dst.rcu_head,
        (void (*)(void *))dst_free,
        &rt->u.dst);
}
```

Figure 5: `dst_free()` Modifications

3 Denial-of-Service Attacks on Deferred Freeing

[McK02a] describes how RCU is used in the IPV4 route cache to void acquiring the per-bucket reader-writer lock during lookup and the corresponding speed-up of route cache lookup. This was included in the 2.5.53 kernel. Later, Robert Olsson subjected a 2.5 kernel based router to DoS stress tests using `pktgen` and discovered problems including starvation of user-space execution and out-of-memory conditions. In this section, we describe our analysis of those problems and potential remedies that were experimented with.

3.1 Potential Out-of-Memory Situation

Starting with the 2.5.53 kernel, the IPv4 route cache uses RCU to enable lock-free lookup of the route hash table.

The code in Figure 5 shows how route cache entries are freed. Because each route cache entry's freeing is deferred by `call_rcu()`, it is not returned to its slab immediately. However

```
CLONE_SKB="clone_skb 1"
PKT_SIZE="pkt_size 60"
COUNT="count 10000000"
IPG="ipg 0"
PGDEV=/proc/net/pktgen/eth0
echo "Configuring $PGDEV"
pgset "$COUNT"
pgset "$CLONE_SKB"
pgset "$PKT_SIZE"
pgset "$IPG"
pgset "flag IPDST_RND"
pgset "dst_min 5.0.0.0"
pgset "dst_max 5.255.255.255"
pgset "flows 32768"
pgset "flowlen 10"
```

Figure 6: pktgen parameters

the route cache imposes a limit of total number of in-flight entries at `ip_rt_max_size`. If this limit is exceeded, subsequent allocation of route cache entries are failed. We reproduced Robert's experiment in a setup where we send 100,000 packets/sec to a 2.4GHz Pentium IV Xeon 2-CPU HT system with 256MB RAM running 2.6.0 kernel set up as a router. Figure 6 shows the parameters used in `pktgen` testing. This script sends 10000000 packets to the router with random destination addresses in the range 5.0.0.0 to 5.255.255.255. The router has an outgoing route set up to sink these packets. This results in a very large number of route cache entries along with pruning of the cache due to aging and garbage collection.

We then instrumented RCU infrastructure to collect lengths of RCU callback batches invoked after grace periods and corresponding grace period lengths. As indicated by the graph plotted based on this instrumentation (Figure 7), it is evident that every spike in RCU batch length as an associated spike in RCU grace period. This indicates that prolonged grace periods are resulting in very large numbers of pending callbacks.

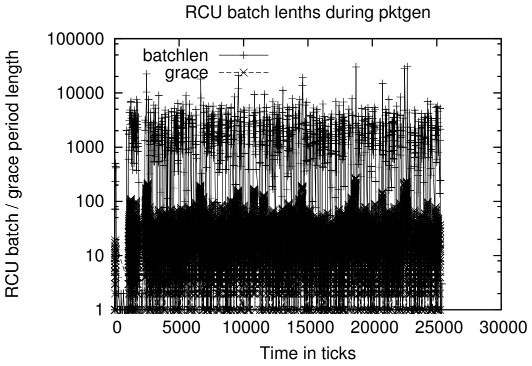

Figure 7: Effect of pktgen testing on RCU

Next we used the same instrumentation to understand what causes long grace periods. We measured total number of softirqs received by each cpu during consecutive periods of 4 `jiffies` (approximately 4 milliseconds) and plotted it along with the corresponding maximum RCU grace period length seen during that period. Figure 8 shows this relationship. It clearly shows that all peaks in RCU grace period had corresponding peaks in number of softirqs received during that period. This conclusively proves that large floods of softirqs holds up progress in RCU. An RCU grace period of 300 milliseconds during a 100,000 packets/sec DoS flood means that we may have up to 30,000 route cache entries pending in RCU subsystem waiting to be freed. This causes us to quickly reach the route cache size limits and overflow.

In order to avoid reaching the route cache entry limits, we needed to reduce the length of RCU grace periods. We then introduced a new mechanism named *rcu-softirq* [Sarma04a] that considers completion of a softirq handler a *quiescent* state. It introduces a new interface `call_rcu_bh()`, which is to be used when the RCU protected data is mostly used from softirq handlers. The update function will be invoked as soon as all CPUs have performed a context switch or been seen in the

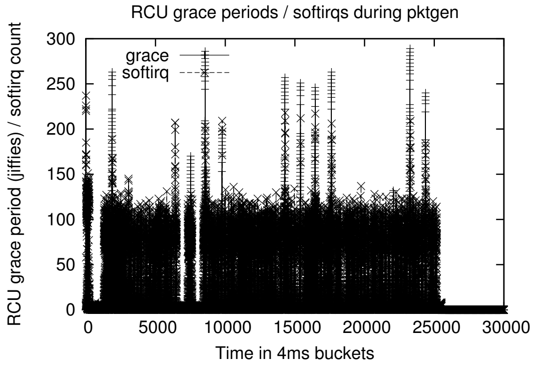

Figure 8: Softirqs during pktgen testing

idle loop or in a user process or or has exited a softirq handler that it may have been executing. The reader side of critical section that use call_rcu_bh() for updating must be protected by `rcu_read_lock_bh()` and `rcu_read_unlock_bh()`. The IPv4 route cache code was then modified to use these interfaces instead. With this in place, we were able to avoid route cache overflows at the rate of 100,000 packets/second. At higher packet rates, route cache overflows have been reported. Further analysis is being done to determine if at higher packet rates, current softirq implementation doesn't allow route cache updates to keep up with new route entries getting created. If this is the case, it may be necessary to limit softirq execution in order to permit user-mode execution to continue even in face of DoS attacks.

3.2 CPU Starvation Due to softirq Load

During the `pktgen` testing, there was another issue that came to light. At high softirq load, user-space programs get starved of CPU. Figure 9 is a simple piece of code that can be used to test this under severe `pktgen` stress. In our test router, it indicated user-space starvation for periods longer that 5 seconds. Application of the *rcu-softirq* patch reduced it by a few seconds. In other words, introduction of quicker

```
gettimeofday(&prev_tv, NULL);

for (;;) {
    gettimeofday(&tv, NULL);
    diff = (tv.tv_sec - prev_tv.tv_sec)*
        1000000 +
        (tv.tv_usec - prev_tv.tv_usec);
        if (diff > 1000000)
            printf("%d\n", diff);
    prev_tv = tv;
}
```

Figure 9: user-space starvation test

RCU grace periods helped by reducing size of pending RCU batches. But the overall softirq rate remained high enough to starve user-space programs.

4 Realtime Response

Linux has been use in realtime and embedded applications for many years. These applications have either directly used Linux for soft realtime use, or have used special environments to provide hard realtime, while running the soft-realtime or non-realtime portions of the application under Linux.

4.1 Hard and Soft Realtime

Realtime applications require latency guarantees. For example, such an application might require that a realtime task start running within one millisecond of its becoming runnable. Andrew Morton's `amlat` program may be used to measure an operating system's ability to meet this requirement. Other applications might require that a realtime task start running within 500 microseconds of an interrupt being asserted.

Soft realtime applications require that these guarantees be met *almost* all the time. For example, a building control application might require that lights be turned on within 250 milliseconds of motion being detected within a given room. However, if this application occasionally responds only within 500 milliseconds, no harm is likely to be done. Such an application might require that the 250 millisecond deadline be met 99.9% of the time.

In contrast, hard realtime applications require that guarantees *always* be met. Such applications may be found in avionics and other situations where lives are at stake. For example, Stealth aircraft are aerodynamically unstable in all three axes, and require frequent computer-controlled attitude adjustments. If the aircraft fails to receive such adjustments over a period of two seconds, it will spin out of control and crash [Rich94]. These sorts of applications have traditionally run on "bare metal" or on a specialized realtime OS (RTOS).

Therefore, while one can validate a soft-realtime OS by testing it, a hard-realtime OS must be validated by inspection and testing of *all* non-preemptible code paths. Any non-preemptible code path, no matter how obscure, can destroy an OS's hard-realtime capabilities.

4.2 Realtime Design Principles

This section will discuss how preemption, locking, RCU, and system size affect realtime response.

4.2.1 Preemption

In theory, neither hard nor soft realtime require preemption. In fact, the realtime systems that one of the authors (McKenney) worked on in the 1980s were all non-preemptible. However, in practice, preemption can greatly reduce the amount of work required to design and validate a hard realtime system, because while one must validate *all* code paths in a non-preemptible system, one need only validate all *non-preemptible* code paths in a preemptible system.

4.2.2 Locking

The benefits of preemption are diluted by locking, since preemption must be suppressed across any code path that holds a spinlock, even in UP kernels. Since most long-running operations are carried out under the protection of at least one spinlock, the ability of preemption to reduce the Linux kernel's hard realtime response is limited.

That said, the fact that spinlock critical sections degrade realtime response means that the needs of the hard realtime Linux community are aligned with those of the SMP-scalability Linux community.

Traditionally, hard-realtime systems have run on uniprocessor hardware. The advent of hyperthreading and multicore dies have provided cheap SMP, which is likely to start finding its way into realtime and embedded systems. It is therefore reasonable to look at SMP locking's effects on realtime response.

Obviously, a system suffering from heavy lock contention need not apply for the job of a realtime OS. However, if lock contention is sufficiently low, SMP locking need not preclude hard-realtime response. This is shown in Figure 10, where the maximum "train wreck" lock spin time is limited to:

$$S_{max} = (N_{CPU} - 1)C_{max} \tag{1}$$

where N_{CPU} is the number of CPUs on the system and C_{max} is the maximum critical section length for the lock in question. This maximum lock spin time holds as long as each CPU spends at least S_{max} time outside of the critical section.

It is not yet clear whether Linux's lock contention can be reduced sufficiently to make this

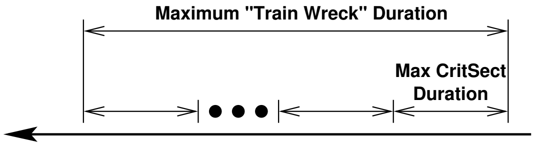

Figure 10: SMP Locking and Realtime Response

level of hard realtime guarantee, however, this is another example of a case where improved realtime response benefits SMP scalability and vice versa.

4.2.3 RCU

Towards the end of 2003, Robert Love and Andrew Morton noted that the Linux 2.6 kernel's RCU implementation could degrade realtime response. This degradation is due to the fact that, when under heavy load, literally thousands of RCU callbacks will be invoked at the end of a grace period, as shown in Figure 11.

The following three approaches can each eliminate this RCU-induced degradation:

1. If the batch of RCU callbacks is too large, hand the excess callbacks to a preemptible per-CPU kernel daemon for invocation. The fact that these daemons are preemptible eliminates the degradation.

2. On uniprocessors, in cases where pointers to RCU-protected elements are not held across calls to functions that remove those elements, directly invoke the RCU callback from within the `call_rcu_rt()` primitive, which is identical to the `call_rcu()` primtive in SMP kernels. The separate `call_rcu_rt()` primitive is necessary because direct invocation is not safe in all cases.

3. Throttling RCU callback invocation so that only a limited number are invoked at a given time, with the remainder being invoked later, after there has been an opportunity for realtime tasks to run.

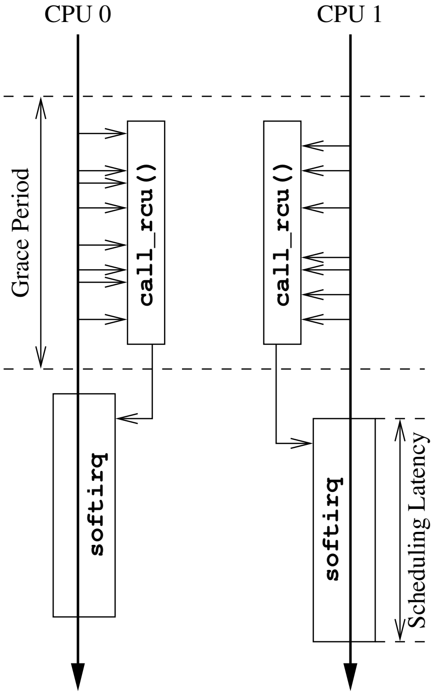

Figure 11: RCU and Realtime Response

The throttling approach seems most attractive currently, but additional testing will be needed after other realtime degradations are resolved. The implementation of each approach and performance results are presented elsewhere [Sarma04b].

4.2.4 System Size

The realtime response of the Linux 2.6 kernel depends on the hardware and software configuration. For example, the current VGA driver degrades realtime response, with multi-millisecond scheduling delays due to screen blanking.

In addition, if there are any non-O(1) operations in the kernel, then increased configuration sizes will result in increased realtime scheduling degradations. For example, in SMP systems, the duration of the worst-case locking "train wreck" increases with the number of CPUs. Once this train-wreck duration exceeds the minimum time between release and later acquisition of the lock in question, the worst-case scheduling delay becomes unbounded. Other examples include the number of tasks and the duration of the tasklist walk resulting from `ls /proc`, the number of processes mapping a given file and the time required to truncate that file, and so on.

In the near term, it seems likely that realtime-scheduling guarantees would only apply to a restricted configuration of the Linux kernel, running a restricted workload.

4.3 Linux Realtime Options

The Linux community can choose from the following options when charting its course through the world of realtime computing:

1. "Just say no" to realtime. It may well be advisable for Linux to limit how much realtime support will be provided, but given recent measurements showing soft-realtime scheduling latencies of a few hundred *micro*seconds, it seems clear that Linux has a bright future in the world of realtime computing.

2. Realtime applications run only on UP kernels. In the past, realtime systems have overwhelmingly been single-CPU systems, it is much easier to provide realtime scheduling guarantees on UP systems. However, the advent of cheap SMP hardware in the form of hyperthreading and multi-CPU cores makes it quite likely that the realtime community will choose to support SMP sooner rather than later.

   One possibility would be to provide tighter guarantees on UP systems, and, should Linux provide hard realtime support, to provide this support only on UP systems. Another possibility would be to dedicate a single CPU of an SMP system to hard realtime.

3. Realtime applications run only on small hardware configurations with small numbers of tasks, mappings, open files, and so on. This seems to be an eminently reasonable position, especially given that dirt-cheap communications hardware is available, allowing a small system (perhaps on a PCI card) to handle the realtime processing, with a large system doing non-realtime tasks requiring larger configurations.

4. Realtime applications use only those devices whose drivers are set up to provide realtime response. This also seems to be an eminently reasonable restriction, as open-source drivers can be rewritten to offer realtime response, if desired.

5. Realtime applications use only those services able to provide the needed response-time guarantees. For example, an application that needs to respond in 500 microseconds is not going to be doing any disk I/O, since disks cannot respond this quickly. Any data needed by such an application must be obtained from much faster devices or must be preloaded into main memory.

It is not clear that Linux will be able to address each and every realtime requirement, nor is it clear that this would even be desirable. However, it was not all that long ago that common wisdom held that it was not feasible to address both desktop and high-end server requirements with a single kernel source base. Linux is well on its way to proving this common wisdom to be quite wrong.

It will therefore be quite interesting to see what realtime common wisdom can be overturned in the next few years.

5 Future Plans

With the 2.6 kernel behind us, a number of new scalability issues are currently being investigated. In this section, we outline a few of them and the implications they might have.

5.1 Parallel Directory Entry Cache Updates

In the 2.4 kernel, the directory entry cache was protected by a single global lock `dcache_lock`. In the 2.6 kernel, the look-ups into the cache were made lock-free by using RCU [Linder02a]. We also showed in [Linder02a] that for several benchmarks, only 25% of acquisitions of `dcache_lock` is for updating the cache. This allowed us to achieve significant performance improve by avoiding the lock during look-up while keeping the updates serialized using `dcache_lock`. However recent benchmarking on large SMP systems have shown that `dcache_lock` acquisitions are proving to be costly. Profile for a mutli-user benchmark on a 16-CPU Pentium IV Xeon with HT indicates this:

Function	Profile Counts
`.text.lock.dec_and_lock`	34375.8333
`atomic_dec_and_lock`	1543.3333
`.text.lock.libfs`	800.7429
`.text.lock.dcache`	611.7809
`__down`	138.4956
`__d_lookup`	93.2842
`dcache_readdir`	70.0990
`do_page_fault`	45.0411
`link_path_walk`	9.4866

On further investigation, it is clear that `.text.lock.dec_and_lock` cost is due to frequent `dput()` which uses `atomic_dec_and_test()` to acquire `dcache_lock`. With the multi-user benchmark creating and destroying large number of files in `/proc` filesystem, the cost of corresponding updates to the directory entry cache is hurting us. During the 2.7 kernel development, we need to look at allowing parallel updates to the directory entry cache. We attempted this [Linder02a], but it was far too complex and too late in the 2.5 kernel development effort to permit such a high-risk change.

5.2 Lock-free TCP/UDP Hash Tables

In the Linux kernel, INET family sockets use hash tables to maintain the corresponding `struct sock`s. When an incoming packet arrives, this allows efficient lookup of these per-bucket locks. On a large SMP system with tens of thousands on tcp and ip header information. TCP uses `tcp_ehash` for connected sockets, `tcp_listening_hash` for listening sockets and `tcp_bhash` for bound sockets. On a webserver serving large number of simultaneous connections, lookups into `tcp_ehash` table are very frequent. Currently we use a per-bucket reader-writer lock to protect the hash tables and `tcp_ehash` lookups are protected by acquiring the reader side of these per-bucket locks. The hash table makes CPU-CPU collisions on hash chains unlikely and prevents the reader-writer lock from providing any possible performance benefit. Also, on a large SMP system with tens of thousands of simultaneous connection, the cost of atomic operation during `read_lock()` and `read_unlock()` as well as the bouncing of cache line containing the lock becomes a factor. By using RCU to protect the hash tables, the lookups can be done without acquiring the per-bucket lock. This will benefit bot low-end and high-end SMP systems. That said, issues similar to the ones discussed in Section 3 will need to be addressed. RCU can be stressed using a DoS flood that opens and closes a lot of connections. If the DoS flood prevents user-mode execution, it can also prevent RCU grace periods from happening frequently, in which case, a large number of `sock` structures can be pending in RCU waiting to be free leading to potential out-of-memory situations. The *rcu-softirq* patch discussed in Section 3 will be helpful in this case too.

5.3 Balancing Interrupt and Non-Interrupt Loads

In Section 3.2, we discussed user programs getting starved of CPU time under very high network load. In 2001, Ingo Molnar attempted limiting hardware interrupts based on number of such interrupts serviced during one jiffy [Molnar01a]. Around the same time, Jamal Hadi et al. demonstrated the usefulness of limiting interrupts through *NAPI* infrastructure [Jamal01a]. *NAPI* is now a part of 2.6 kernel and it is supported by a number of network drivers. While *NAPI* limits hardware interrupts, it continues to raise softirqs for processing of incoming packets while polling. So, under high network load, we see user processes starved of CPU. This has been seen with *NAPI* (Robert Olsson's lab) as well as without *NAPI* (in our lab). With extremely high network load like DoS stress, softirqs completely starve user processes. Under such situation, a system administrator may find it difficult to take log into a router and take necessary steps to counter the DoS attack. Another potential problem is that network I/O intensive benchmarks like SPECWeb99™ can have user processes stalled due to high softirq load. We need to look for a new framework that allows us to balance CPU usage between softirqs and process context too. One potential idea being considered is to measure softirq processing time and mitigate it for later if it exceeds its tunable quota. Variations of this need to be evaluated during the development of 2.7 kernel.

5.4 Miscellaneous

1. Lock-free dcache Path Walk: Given a file name, the Linux kernel uses a path walking algorithm to look-up the `dentry` corresponding to each component of the file name and traverse down the `dentry` tree to eventually arrive at the `dentry` of the specified file name. In 2.6 kernel, we implemented a mechanism to look-up each path component in dcache without holding the global `dcache_lock` [Linder02a]. However this requires acquiring a per-`dentry` lock when we have a successful look-up in dcache. The common case of paths starting at the root directory results in contention on the root `dentry` on large SMP systems. Also, the per-`dentry` lock acquisition happens in the fast path (`__d_lookup()`) and avoiding this will likely provide nice performance benefits.

2. Lock-free Tasklist Walk: The system-wide list of tasks in the Linux kernel is protected by a reader-writer lock `tasklist_lock`. There are a number of occasions when the list of tasks need to be traversed while holding the reader side of `tasklist_lock`. In systems with very large number of tasks, the readers traversing the task list can starve out writers. One approach to solving this is to use RCU to allow lock-free walking of the task list under limited circumstances. [McKenney03a] describes one such experiment.

3. Cache Thrashing Measurements and Minimization: As we run Linux on larger SMP and NUMA systems, the effect of cache thrashing becomes more prominent. It would prudent to analyze cache behavior of performance critical code in the Linux kernel using various performance

monitoring tools. Once we identify code showing non-optimal cache behavior, redesigning some of it would help improve performance.

4. Real-time Work—Fix Excessively Long Code Paths: With Linux increasingly becoming preferred OS for many soft-realtime systems, we can further improve its usefulness by identifying excessively long code paths and fixing them.

6 Conclusions

In 2.6 kernel, we have solved a number of scalability problems without significantly sacrificing performance in small systems. A single code base supporting so many different workloads and architectures is an important advantages of the Linux kernel has over many other operating systems. Through this analysis, we have continued the process of evaluating scalability enhancements from many possible angles. This will allow us to run Linux better on many different types of system—large SMP to small TCP/IP routers.

We are continuing to work on some of the core issues discussed in the paper including locking overheads, RCU DoS attack prevention and softirq balancing. We expect to do some of this work in the 2.7 kernel timeframe.

7 Acknowledgments

We owe thanks to Andrew Morton for drawing attention to locking issues. Robert Olsson was the first to show us the impact of denial-of-service attacks on route cache and without his endless testing with our patches, we would have gone nowhere. Ravikiran Thirumalai helped a lot with our lab setup, revived some of the lock-free patches and did some of the measurements. We are thankful to him for this. We would also like to thank a number of Linux kernel hackers, including Dave Miller, Andrea Arcangeli, Andi Kleen and Robert Love, all of whom advised us in many different situations. Martin Bligh constantly egged us on and often complained with large SMP benchmarking results, and for this, he deserves our thanks. We are indebted to Tom Hanrahan, Vijay Sukthankar, Dan Frye and Jai Menon for being so supportive of this effort.

References

[Blanchard02a] A. Blanchard *some RCU dcache and ratcache results*, Linux-Kernel Mailing List, March 2002. `http://marc.theaimsgroup.com/?l=linux-kernel&m=101637107412972&w=2`,

[Jamal01a] Jamal Hadi Salim, R. Olsson and A. Kuznetsov *Beyond Softnet*, Proceedings of USENIX 5th Annual Linux Showcase, pp. 165–172, November 2001.

[Linder02a] H. Linder, D. Sarma, and Maneesh Soni. *Scalability of the Directory Entry Cache*, Ottawa Linux Symposium, June 2002.

[Love04a] Robert Love *Kernel Korner: Kernel Locking Techniques*, *Linux Journal*, Issue 100, August 2002. `http://www.linuxjournal.com/article.php?sid=5833`,

[McK98a] P. E. McKenney and J. D. Slingwine. *Read-copy update: using execution history to solve concurrency problems*, Parallel and Distributed Computing and Systems, October 1998. (revised version available at `http://www.rdrop.com/users/paulmck/rclockpdcsproof.pdf`), [McK01a] P. E. McKenney and D. Sarma. *Read-Copy Update Mutual Exclusion in Linux*, http://lse.sourceforge.net/locking/rcu/rcupdate_doc.html, February 2001.

[McK01b] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, M. Soni. *Read-Copy Update*, Ottawa Linux Symposium, July 2001. (revised version available at http://www.rdrop.com/users/paulmck/rclock/rclock_OLS.2001.05.01c.sc.pdf),

[McK02a] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger *Read-Copy Update*, Ottawa Linux Symposium, June 2002. http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz

[McKenney03a] MCKENNEY, P.E. Using RCU in the Linux 2.5 kernel. *Linux Journal 1*, 114 (October 2003), 18–26.

[Molnar01a] Ingo Molnar *Subject: [announce] [patch] limiting IRQ load, irq-rewrite-2.4.11-B5*, Linux Kernel Mailing List, Oct 2001. http://www.uwsg.iu.edu/hypermail/linux/kernel/0110.0/0169.html,

[Morton03a] Andrew Morton *Subject: smp overhead, and rwlocks considered harmful*, Linux Kernel Mailing List, March 2003. http://www.uwsg.iu.edu/hypermail/linux/kernel/0303.2/1883.html

[Rich94] B. Rich *Skunk Works*, Back Bay Books, Boston, 1994.

[Sarma02a] D. Sarma *Subject: Some dcache_rcu benchmark numbers*, Linux-Kernel Mailing List, October 2002. http://marc.theaimsgroup.com/?l=linux-kernel&m=103462075416638&w=2,

[Sarma04a] D. Sarma *Subject: Re: route cache DoS testing and softirqs*, Linux-Kernel Mailing List, April 2004. http://marc.theaimsgroup.com/?l=linux-kernel&m=108077057910562&w=2,

[Sarma04b] D. Sarma and P. McKenney *Making RCU Safe for Deep Sub-Millisecond Response Realtime Applications*, USENIX'04 Annual Technical Conference (UseLinux Track), Boston, June 2004.

8 Legal Statement

This work represents the view of the authors and does not necessarily represent the view of IBM.

Linux is a trademark of Linus Torvalds.

Pentium and Xeon are trademarks of Intel Corporation.

SPEC™ and the benchmark name SPECweb99™ are registered trademarks of the Standard Performance Evaluation Corporation.

IBM is a trademark of International Business Machines Corporation.

Other company, product, and service names may be trademarks or service marks of others.

Proceedings of the
Linux Symposium

Volume Two

July 21st–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*