Linux Scheduler Latency

Clark Williams, Red Hat, Inc.

March 2002

Abstract

Back in early November 2001, I started following a discussion between two factions of the Linux kernel community. The gist of the discussion was over what was the best solution to the Linux scheduler latency problem, that is the delay between the occurrence of an interrupt and the running of the process that services the interrupt. There were two main factions, the *preemption patch* faction and the *low−latency patch* faction. Both groups were very passionate (i.e. vocal) about the superiority of their solution. Since one of my jobs at Red Hat is to evaluate and recommend new techniques for embedded Linux solutions and since scheduler latency is one of the biggest complaints that hard realtime champions have about Linux, I decided to evaluate both patches and see which one came out on top.

Since I'm primarily interested in embedded Linux and since most embedded Linux systems are uniprocessor systems, my testing focused exclusively on uniprocessors systems. I tested both patches on a 2.4.17 kernel under heavy load and while both patches significantly reduced kernel latency, the testing shows that the low−latency patches yielded the best reduction of Linux scheduler latency. The low−latency patches had a maximum recorded latency of 1.3 milliseconds, while the preemption patches had a maximum latency of 45.2ms.

A 2.4.17 kernel patched with a combination of both preemption and low−latency patches yielded a maximum scheduler latency of 1.2 milliseconds, a slight improvement over the low−latency kernel. However, running the low−latency patched kernel for greater than twelve hours showed that there are still problem cases lurking, with a maximum latency value of 215.2ms recorded. Running the combined patch kernel for more than twelve hours showed a maximum latency value of 1.5ms. This data seems to indicate that the best solution for reducing Linux scheduler latency is to combine both sets of patches.

What is Scheduler Latency?

What the heck does scheduler latency mean anyway? Latency is really a shorthand term for the phrase latent period, which is defined by webster.com to be "the interval between stimulus and response". In the context of the Linux kernel, scheduler latency is the time between a wakeup (the stimulus) signaling that an event has occurred and the kernel scheduler getting an opportunity to schedule the thread that is waiting for the wakeup to occur (the response). Wakeups can be caused by hardware interrupts, or by other threads. For simplicity's sake, this article will concentrate on device interrupts when discussing wakeups, but it should be noted that thread−based wakeups are handled similarly.

Another thing to remember as you read this article, is that when I talk about the problems surrounding thread preemption, I'm talking about preempting kernel threads. User space threads are fully preemptable and have been since very early versions of Linux. If you don't know what I'm talking about, you should probably pick up one of the numerous books on the workings of the Linux kernel.

Scheduling and Interrupts

When a thread issues an I/O request via a kernel system call (such as a read() call), if the device driver that handles the system call cannot satisfy the request for data, it puts the thread on a wait queue. This puts the requesting thread to sleep so that it will not be eligible to run while waiting for the device to provide the data. The driver then insures that the I/O request will generate an interrupt when the device completes the request and then the driver returns control to the kernel. So at this point, the requesting thread is asleep and the device is performing the requested I/O.

The kernel goes on its merry way, running all threads that are eligible to run (this excludes the thread that originated the I/O requests we're interested in). At some point after the I/O request is issued, the device will complete the request and assert an interrupt line. This will cause an interrupt service routine to run in the kernel, which runs the device driver's interrupt handler. The driver will figure out if the data from the device will satisfy the read request from the originating thread and if so, will remove the thread from the driver wait queue, making it ready to run (this 'making a thread ready to run' is what we've been calling a wakeup). As a consequence of completing the I/O request and removing the thread from the wait queue, the kernel code will check to see if the thread that has been awakened can run and if it can, the kernel will indicate that a scheduling pass needs to be made by setting the need_resched flag in the current task structure.

When the kernel gets to a point where scheduling is allowed, it will note that need_resched is set and will call the function schedule(), which will determine what thread should run next. If the thread that issued the read() has a high enough goodness value (a value calculated from priority, amount of CPU−time used, and other things), the kernel will context switch to it and when the kernel transitions back to user space the thread will run and process the received data. The amount of time that elapses from when the interrupt is asserted to when the thread that issued the I/O request runs, is what I call kernel response time.

Components of Response Time

There are a four components that make up the kernel's response time to an event:

1. Interrupt Latency

2. Interrupt handler duration

3. Scheduler Latency

4. Scheduling duration

The first component, interrupt latency is the amount of time that elapses between the physical interrupt signal being asserted and the interrupt service routine running. The second component interrupt handler duration , is the amount of time spent in the routine that actually handles the interrupt. The interrupt handler is the routine that sets the need_resched flag, signifying that a scheduling pass is needed. The third component, scheduler latency is the amount of time that elapses between the interrupt service routine completing and the scheduling function being run. On an SMP system, this component may be nonexistent, since the execution of the interrupt handler and the execution of the scheduler may actually overlap. Since we're focusing on uniprocessor systems, overlap of interrupt handler and scheduling will not be discussed further. The fourth component, scheduling duration is the amount of time that elapses inside the scheduler proper to decide what thread to run and context switch to it.

This article will focus on the third component, scheduler latency and save interrupts and scheduler discussions for another day. The reason is that the other three components typically have very low duration's, when compared the scheduler latency. Information from Ingo Molnar is that interrupt latency for recent x86 hardware is on the order of 10µs, while interrupt handler duration is typically tens to at most low−hundreds of microseconds, and scheduling duration is a few microseconds. With maximum scheduler latency weighing in at tens to hundreds of milliseconds on an unmodified Linux kernel, you can see that the other three components are not of primary interest.

Why Reduce Scheduler Latency?

Having a large scheduler latency means that the kernel doesn't respond very quickly to I/O events. If you're building a Personal Digital Recorder (PDR), then you are going to be processing the heck out of MPEG audio/video streams and you will want the kernel to schedule your MPEG decoder process as quickly as possible. If somebody misses the juicy bits from their favorite B−movie because the video stream glitched, they're not going to care that it's because the kernel was slowed down by a particularly slow write to the internal disk. They're just gonna be mad that the movie looks jerky.

Another example is companies that are implementing DSL modems with minimal hardware. To reduce the cost of the device they want to do away with the Digital Signal Processors (DSPs) that are used to process the analog signals on a phone line into DSL cells or frames. They do this by offloading the signal processing algorithms to the main processor (similar to the things that the infamous ?WinModems do). To successfully implement this sort of scheme requires that the interrupt dispatch latency and thread scheduling latency be minimized in the kernel.

Scheduler Latency Causes

Large scheduler latencies have numerous causes. One culprit are drivers that do lots of processing in their entry points or interrupt service routine (ISR). Another is kernel code that stays in a block of code for a long period of time, without explicitly introducing scheduling opportunities. Both of these things cause one problem: the kernel does not have the opportunity to perform scheduling calculations for a long period of time. Since the scheduler is the mechanism for determining what process/thread should run, it needs to be run with a relatively high frequency for the kernel to be responsive. The bottom line goal of someone trying to reduce scheduler latency is to insure that opportunities to run the scheduler occur regularly, so that tasks needed to service events get a chance to run as quickly as possible.

When discussing scheduler latency, there are two data points that most people are interested in. The first is the maximum latency, which is the largest value of scheduler latency measured in a test run. Although this value is almost always statistically insignificant, it is a killer when you're trying to decide whether to use Linux in a device. Why? Because there are tasks where you cannot, ever, be late. Closing control valves in a chemical plant, moving the control surfaces on a missile, things like that. Oh, you can be late, but something catastrophic will happen if you are. So the maximum latency is usually very interesting to system designers.

The other factor is the scheduler latency value below which five−9's worth of samples have occurred. Five−9's is just marketing−speak for 99.999%, so the interesting point is where 99.999% of the samples have occurred. In addition to being overused by sales and marketing types, this value is used by system designers when they're designing a system that needs a scheduler to be good enough, and can tolerate some jitter in scheduler response.

How to Reduce Scheduler Latency?

A simple answer to that question is to run the scheduler more frequently. Unfortunately, it's not quite that simple, since if you run the scheduler too fast, you spend all your CPU cycles trying to figure out what should run and not enough of them actually running threads. So it's really more important that scheduling calculations be run regularly, not necessarily at a high rate. The actual answer is to run the scheduler as soon as possible after an event (such as an interrupt) indicates the need for a scheduler pass.

Most Real Time Operating Systems (RTOS) have kernel code and drivers that are written to be preemptable and applications are written to minimize delays. Most RTOS's prioritize device interrupts so that important interrupts ("shut down the reactor NOW!") are serviced first and lower priority interrupts ("time to make the doughnuts") are serviced later. Threads are created into a fairly rigid priority hierarchy and the thread scheduler ruthlessly schedules the highest priority threads first. When an event occurs indicating that a thread needs to run, the RTOS runs the scheduler, which context switches to the thread that needs to service the event (assuming that the event is the highest priority event currently in need of service).

In the Linux kernel, all interrupts are created equal; any prioritization of interrupts is left to the hardware. The Linux kernel is not a preemptable kernel, in that the assumption is made that once the kernel has been entered from a trap or
 syscall, the "current process" will not change unexpectedly (the current process can only be rescheduled voluntarily). Linux kernels use the thread scheduling policies SCHED_FIFO and SCHED_RR to indicate that a task should be scheduled ahead of ones using the default policy SCHED_OTHER, but there is not a fine−grained priority hierarchy that would be available on an RTOS. When an event occurs that requires a thread to run, the Linux kernel sets a flag in the current processes state structure, need_resched , that is checked by kernel code to determine if a scheduling pass is needed.

Linux was not designed to be a RTOS. If you have a hard realtime target in the 10's of microseconds, you shouldn't be looking at vanilla Linux as a solution; a modified kernel such as FSM Labs ?RTLinux, or a dedicated RTOS are probably more suitable for you.

If however you have a situation where having five−9’s worth of samples below a particular latency value and that value is in the low milliseconds or high hundreds of microseconds, Linux may be what you want.

Preemption Patches

One attempt at improving the responsiveness of the Linux scheduler came from embedded Linux vendor Monta Vista (www.mvista.com). They introduced two kernel modifications, a preemption patch and a realtime scheduler. The preemption patches have gone through a number of releases and have been picked up by the Open Source community. They are currently maintained by Robert M. Love (http://www.tech9.net/rml/linux).

The basic idea behind the preemption patches is to create opportunities for the scheduler to be run more often and minimize the time between the occurrence of an event and the running of the schedule() kernel function. The preemption patches do this by modifying the spinlock macros and the interrupt return code so that if it is safe to preempt the current process and a rescheduling request is pending, the scheduler is called.

What do I mean by "safe to preempt the current process"? Originally, the Linux kernel code assumed that upon entry to the kernel, be it from a trap or interrupt, the current process would not be changed until the kernel decides that it’s safe to reschedule. This assumption was a simplifying assumption that allowed the kernel to modify kernel data structures without requiring that mutual exclusion primitives (such as spinlocks) be used to protect the modifications. Over time, the amount of code that modified kernel data structures without protecting those structures has been reduced, to the point that the preemption patches assume that if the code being executed is not an interrupt handler and no spinlocks are being held, then it is safe to context switch away from the current thread context.

The preemption patches added a variable to the task structure (the structure that maintains state for each thread) named preempt_count . The preempt_count field is modified by the macros preempt_disable(), preempt_enable() and preempt_enable_no_resched() . The preempt_disable() macro increments the preempt_count variable, while the preempt_enable*() macros decrement it. The preempt_enable() macro checks for a reschedule request by testing the value of need_resched and if it is true and the preempt_count variable is zero, calls the function preempt_schedule() . This function marks the fact that a preemption schedule has occurred by adding a large value (0x4000000) to the preempt_count variable, calls the kernel schedule() function, then subtracts the value from preempt_count. The scheduler has been modified to check preempt_count for this active flag and so short−circuit some logic in the scheduler that is redundant when being called from the preemption routine.

The macro spin_lock() was modified to first call preempt_disable() , then actually manipulate the spinlock variable. The macro spin_unlock() was modified to manipulate the lock variable and then call preempt_enable() , and the macro spin_trylock() was modified to first call preempt_disable() and then call preempt_enable() if the lock was not acquired.

In addition to checking for preemption opportunities when releasing a spinlock, the preemption patches also modify the interrupt return path code. This is assembly language code in the architecture specific directory of the kernel source (e.g. arch/arm or arch/mips) that makes the same test done by preempt_enable() and calls the preempt_schedule() routine if conditions are right.

The effect of the preemption patch modifications is to reduce the amount of time between when an wakeup occurs and sets the need_resched flag and when the scheduler may be run. Each time a spinlock is released or an interrupt routine returns, there is an opportunity to reschedule. Early versions of the preemption patches were strictly uniprocessor, since the SMP code did not protect per−CPU variables for performance reasons. Recent versions of the preemption patches now protect per−CPU variables and other non−spinlock protected areas of SMP code that assume non−preemption, so are being considered SMP safe. There are still some problems, such as device initialization code that assumes non−preemption, but that is fixable by disabling preemption during device initialization.

Low−Latency Patches

A different strategy for reducing scheduler latency called the low−latency patches was introduced by Ingo Molnar and is now maintained by Andrew Morton. Rather than attempting a brute−force approach (ala preemption) in a kernel that was not designed for it, these patches focus on introducing explicit preemption points in blocks of code that may execute for long stretches of time. The idea is to find places that iterate over large data structures and figure out how to safely introduce a call to the scheduler if the loop has gone over a certain threshold and a scheduling pass is needed (indicated by need_resched being set). Sometimes this entails dropping a spinlock, scheduling and then reacquiring the spinlock, which is also known as lock breaking.

The low−latency patches are a simple concept, but one not so simple to implement. In fact, they are somewhat high−maintenance. Finding and fixing blocks of code that contribute to high scheduler latency is a time−intensive debugging task. Given the dynamic nature of the state of the Linux kernel, the job of finding and fixing high−latency points in kernel code could be a full−time job.

So, how does one find a high−latency block of code? One tool is Andrew Morton's rtc−debug patch. This patch modifies the real−time−clock driver to look for scheduler latencies greater than a specified threshold and when one occurs, dumps an oops stack backtrace to the system log file. Examining the syslog file and looking at the routines that show up the most leads to the long latency code blocks. From there, a programmer must examine the logic that is causing the latency and either avoid the circumstances or insert a preemption point.

What does a preemption point look like? Here’s a function from the kernel source file fs/dcache.c:

```
void prune_dcache(int count)
{
    spin_lock(&dcache_lock);
    for (;;) {
        struct dentry *dentry;
        struct list_head *tmp;

        tmp = dentry_unused.prev;

        if (tmp == &dentry_unused)
            break;
        list_del_init(tmp);
        dentry = list_entry(tmp, struct dentry, d_lru);

        /* If the dentry was recently referenced, don’t free it. */
        if (dentry−>d_vfs_flags & DCACHE_REFERENCED) {
            dentry−>d_vfs_flags &= ~DCACHE_REFERENCED;
            list_add(&dentry−>d_lru, &dentry_unused);
            continue;
        }
        dentry_stat.nr_unused−−;

        /* Unused dentry with a count? */
        if (atomic_read(&dentry−>d_count))
            BUG();

        prune_one_dentry(dentry);
        if (!−−count)
            break;
    }
    spin_unlock(&dcache_lock);
}
```

This function iterates over the dcache list, attempting to reclaim cached dentry structures. Note that the bulk of the function’s body is an infinite loop, that iterates over a variable length list. The loop terminator is either when count dentry’s have been reclaimed or when the list entry removed from the list is the list header, meaning that the entire circular list has been processed. When the rtc−debug kernel showed that prune_dcache was a high−latency point, a lock−breaking preemption point was added:

```
void prune_dcache(int count)
{
    DEFINE_RESCHED_COUNT;

redo:
    spin_lock(&dcache_lock);
    for (;;) {
        struct dentry *dentry;
        struct list_head *tmp;

        if (TEST_RESCHED_COUNT(100)) {
            RESET_RESCHED_COUNT();
            if (conditional_schedule_needed()) {
                spin_unlock(&dcache_lock);
                unconditional_schedule();
                goto redo;
            }
        }

        tmp = dentry_unused.prev;

        if (tmp == &dentry_unused)
            break;
        list_del_init(tmp);
        dentry = list_entry(tmp, struct dentry, d_lru);

        /* If the dentry was recently referenced, don’t free it. */
        if (dentry−>d_vfs_flags & DCACHE_REFERENCED) {
            dentry−>d_vfs_flags &= ~DCACHE_REFERENCED;
            list_add(&dentry−>d_lru, &dentry_unused);
            continue;
        }
        dentry_stat.nr_unused−−;

        /* Unused dentry with a count? */
        if (atomic_read(&dentry−>d_count))
            BUG();

        prune_one_dentry(dentry);
        if (!−−count)
            break;
    }
    spin_unlock(&dcache_lock);
}
```

Note the addition of the macro DEFINE_RESCHED_COUNT , which defines a counter variable. There is also a label, redo, and the conditional block at the start of the loop. The TEST_RESCHED_COUNT(100) macro increments the counter variable, tests it against the argument and returns true if the variable is greater than or equal to the input argument. So, after 100 iterations of the loop, the variable will be true and the if statement body will be executed. The body resets the counter value to zero, then checks to see if low−latency is enabled and a rescheduling request is pending (current−>need_resched != 0). If a rescheduling pass is needed, the dcache lock is dropped, the scheduler is called (via the low−latency routine unconditional_schedule()) and the code then jumps to the label, which reclaims the dcache lock and starts the process again. This style of lock−breaking works because there is no order imposed on the list. All this code is trying to do is reclaim count number of dentries from the dcache. It doesn’t matter that we restart from the head of the list when we drop the lock. If there were an order to the list, we would have to reclaim the lock after the call to unconditional_schedule() so that we would maintain our position in the list. Assuming of course that the list didn’t change around us when we context switched to another thread...

You might ask, why count loop iterations at all? Why not just check need_resched each time at the top of the loop and call the scheduler if it’s set? The reason is that we want to avoid a condition known as livelock, which could occur if the scheduling pressure on the system is very high. If the scheduling pressure is high (that is, there are many scheduling events being generated and many threads available to run) it would be possible for the system to do nothing but loop between the redo label and the test of need_resched. By only testing need_resched after a set of loop iterations,
we insure that some work gets done.

Measuring Scheduler Latency

To measure the worst case scheduler latency of a particular kernel modification, I first needed to generate a heavy system load. I wanted to have a heavy load running on the test system so that scheduling pressure would be high and many kernel code paths would be taken. As the load generator, I used the cerberus burn−in suite from VA Linux. Cerberus is a set of programs that absolutely hammers a Linux system’s various subsystems. Red Hat uses a configuration of cerberus that is delivered in an RPM package named stress−kernel.

After a bit of experimentation, I set up stress−kernel to run the following programs:

NFS−COMPILE

TTCP

FIFOS_MMAP

P3_FPU

FS

CRASHME

The NFS−COMPILE script is the repeated compilation of a Linux kernel, via an NFS filesystem exported over the loopback device. The TTCP (Test TCP) program sends and receives large data sets via the loopback device. FIFOS_MMAP is a combination test that alternates between sending data between two processes via a FIFO and mmap'ing and operating on a file. The P3_FPU test does operations on floating point matrices. The FS test performs all sorts of unnatural acts on a set of files, such as creating large files with holes in the middle, then truncating and extending those files. Finally the CRASHME test generates buffers of random data, then jumps to that data and tries to execute it.

Once I had a way to load up the system, I had to come up with a way to measure latency. I found a program on Andrew Morton's website, called realfeel , that looked like it would do what I wanted. Realfeel was written by Mark Hahn and modified by Andrew Morton, to specifically measure scheduler latency. It is very Intel IA32 centric, in that it uses the Time−Stamp Counter register of Pentium II's and later, to measure elapsed time. The program first changes its scheduler policy to SCHED_FIFO , making it a realtime thread. This insures that it will be scheduled before all other SCHED_OTHER tasks (the default scheduling policy for Linux). It then locks itself into memory, so page faults won't be an issue. and then determines a cycles−per−second conversion between the output of the rdtsc instruction (a 64−bit value) and wallclock time from gettimeofday(). It then sets up the Real Time Clock (RTC) driver to generate a 2KHz stream of interrupts. Finally, it reads an initial value from the time−stamp register and falls into the measuring loop.

The measuring loop first issues a blocking read on /dev/rtc . The realtime clock device completes reads when an RTC interrupt occurs, so the reads should complete 2048 times per second. The first action taken after the read completes is to get another value from the timestamp register, calculate the delta between the previous read and the current read, then convert that number of cycles in to a value in seconds. Finally, since the ideal cycle time would be 1/2048 or 488µs, the program calculates the difference between the measured interval and the ideal interval. This value is the total scheduler latency (including interrupt latency). The delta between the ideal and actual delay is then mapped to a histogram bucket (bucket size is 100µs) and the value of that bucket is incremented. I modified realfeel to allow for a fixed number of samples, so at the completion of the number of samples, the histogram is written to a file. This file is then statistically analyzed by a perl script and plotted using the gnuplot utility.

I thought very hard about trying to break out the interrupt latency, interrupt duration and scheduler duration components of the value measured by realfeel. To do so though would require some fairly extensive instrumentation of the kernel and couldn't convince myself that I wouldn't skew the numbers with the added instrumentation. I finally decided that I could live with treating all of these as a constant.

The tests were run on a 700MHz AMD Duron system with 360MB RAM and a 20GB Western Digital IDE drive attached to a VIA Technologies ?VT82C686 IDE controller. In each case, the kernel to be tested was booted and the root account was logged onto three consoles (X Windows was not run). On the first console, the 'top' utility was run. On the second console the RH stress−kernel package was run. On the third console, the program 'realfeel' was run.

Kernel Configurations

When I started this project, the Linux kernel was in the throes of a VM system upheaval. After trying a few kernels, I ended up settling on the 2.4.17 kernel tree. I testing with three Linux kernel configurations:

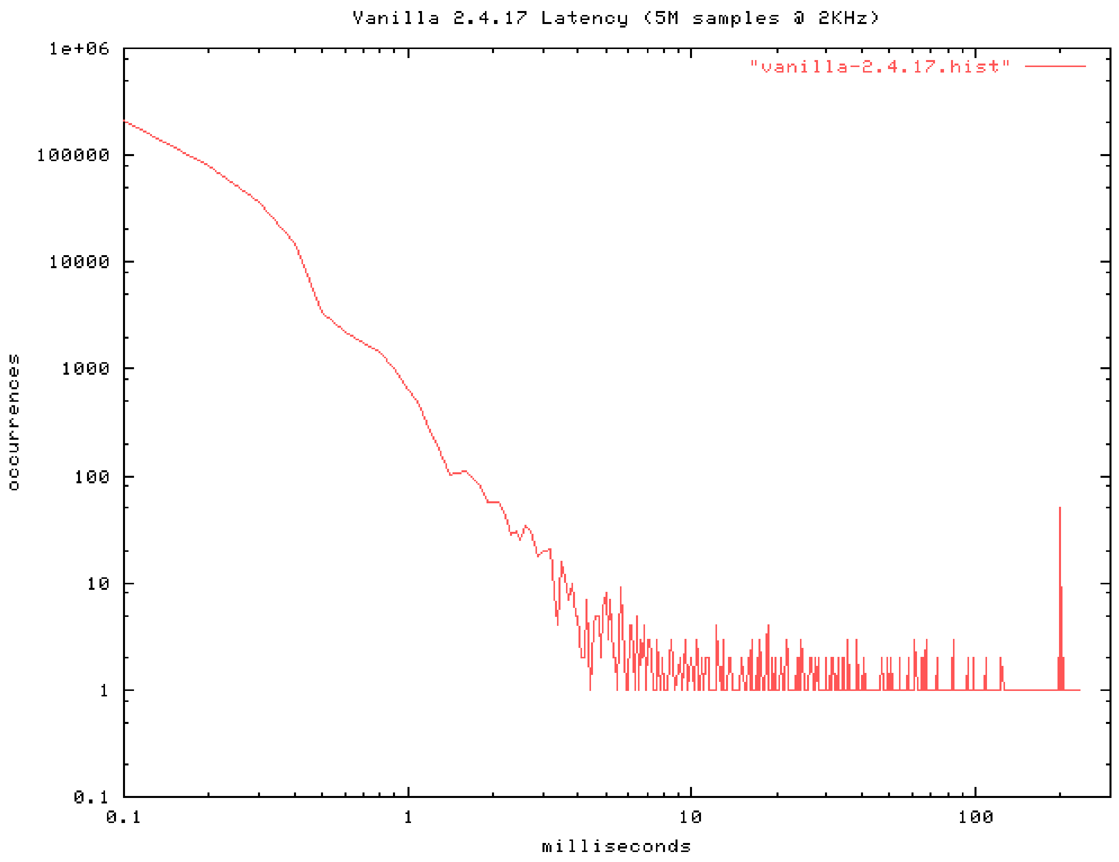
1. A vanilla 2.4.17 kernel
2. A preempt 2.4.17 kernel
3. A low−latency 2.4.17 kernel

The vanilla configuration is the kernel as it comes from kernel.org. The preempt kernel is the vanilla kernel with the preempt−kernel−rml−2.4.17−1.patch patch applied from Robert Love’s site. The low−latency kernel is the vanilla kernel with the 2.4.17−low−latency.patch applied from Andrew Morton’s site.

Results

To compare the various kernel configurations, I booted the appropriate kernel, then ran the stress−kernel program. I then started realfeel with arguments to run for 5 million iterations (5 million RTC clock interrupts) at an interrupt frequency of 2048 interrupts per second (2KHz). This meant each run took approximately 41 minutes to complete. When the run finished, realfeel wrote a histogram file of the results. The histogram shows the number of occurrences of a particular latency value, measured in milliseconds and tenths of a millisecond. These histogram files were then fed into gnuplot to generate a line graph of the run. While examining the graphs, note that both axes are logarithmic scales and that the X axis starts at 100µs. This is important, since in all cases, the majority of latency values fell below 100µs.

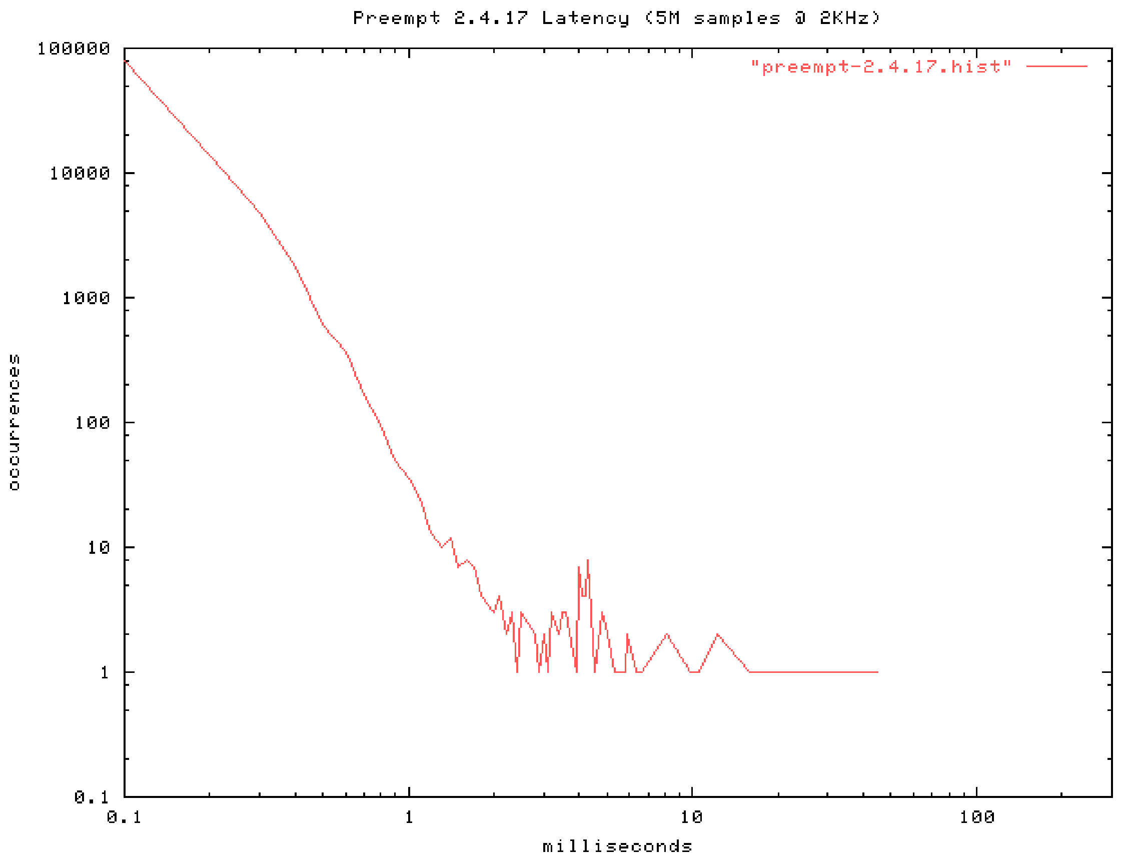
The following graph represents a test run of the vanilla Linux 2.4.17 kernel. Following the graph are some statistics generated by a perl script that analyzes the histogram data.



```
vanilla−2.4.17.hist had 550 buckets containing 5000000 samples
maximum latency: 232.6ms
Mean: 0.0883230599960576ms
Standard Deviation: 2.11903578618566
                 92.84442% of samples < 0.1ms
                 97.08432% of samples < 0.2ms
                 99.73050% of samples < 0.5ms
                 99.84382% of samples < 0.7ms
                 99.94038% of samples < 1ms
                 99.97922% of samples < 5ms
                 99.98096% of samples < 10ms
                 99.98590% of samples < 50ms
                 99.98828% of samples < 100ms
                 100.00000% of samples < 232.7ms
```

The majority of latency measurements fall at or below below 5ms, but the graph shows significant jitter and a large spike near the 200ms point. Not what you'd consider consistent. The maximum latency measured was 232.6ms, the mean latency value was 88µs and 92.84% of the latency samples were below 100 µs.

The next graph is the results of running the preempt 2.4.17 kernel

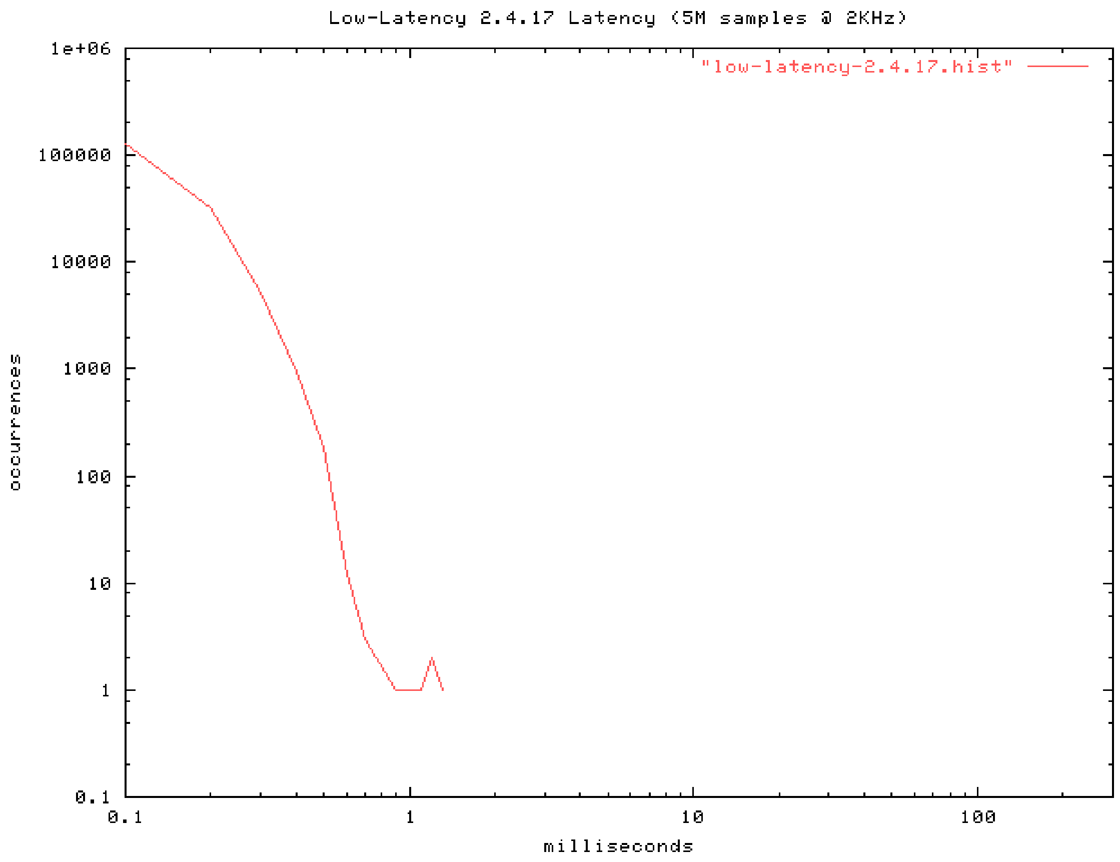


preempt−2.4.17.hist had 62 buckets containing 5000000 samples

maximum latency: 45.2ms
Mean: 0.0528876799956937ms
Standard Deviation: 0.0461523751362407
97.95326% of samples < 0.1ms
99.55722% of samples < 0.2ms
99.97026% of samples < 0.5ms
99.98960% of samples < 0.7ms
99.99650% of samples < 1ms
99.99954% of samples < 5ms
99.99982% of samples < 10ms
100.00000% of samples < 45.3ms

This graph shows that the preemption patches improve the latency of the vanilla kernel by a significant amount. Note that the curve is a bit steeper than the vanilla kernel graph and that the majority of samples fell below 5ms. The maximum latency encountered was 45.2ms, the mean latency value was 53.8µs and 97.95% of the samples fell below 100µs.

Finally, here’s the graph and statistics for the low−latency 2.4.17 kernel.



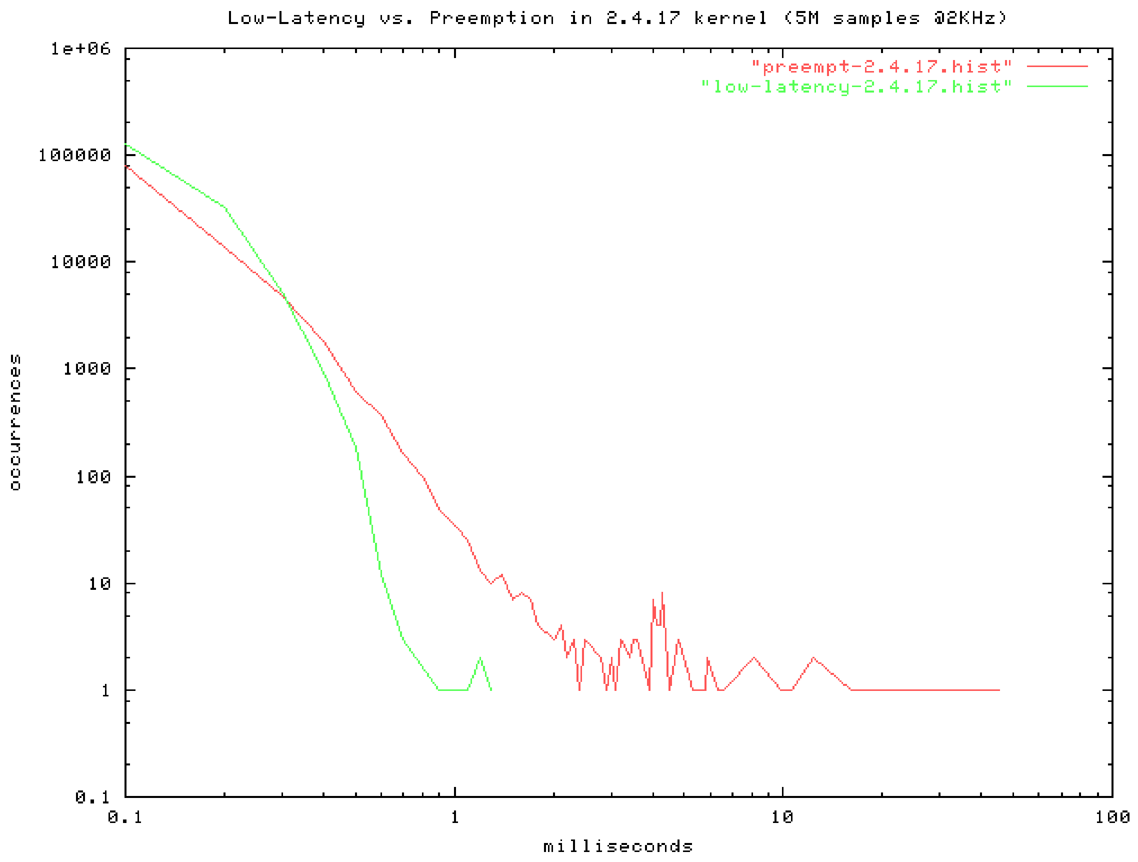
Low−latency−2.4.17.hist had 12 buckets containing 5000000 samples

```
maximum latency: 1.3ms
Mean: 0.0542797399957571ms
Standard Deviation: 0.025220506601371
96.66250% of samples < 0.1ms
99.21158% of samples < 0.2ms
99.99592% of samples < 0.5ms
99.99984% of samples < 0.7ms
99.99992% of samples < 1ms
100.00000% of samples < 1.4m
```

As you can see, the low−latency patches really make a difference in the latency behavior of the kernel. The maximum observed latency value is 1.3ms, the mean latency value is 54.2µs, and 96.66% of the samples occurred below 100µs.

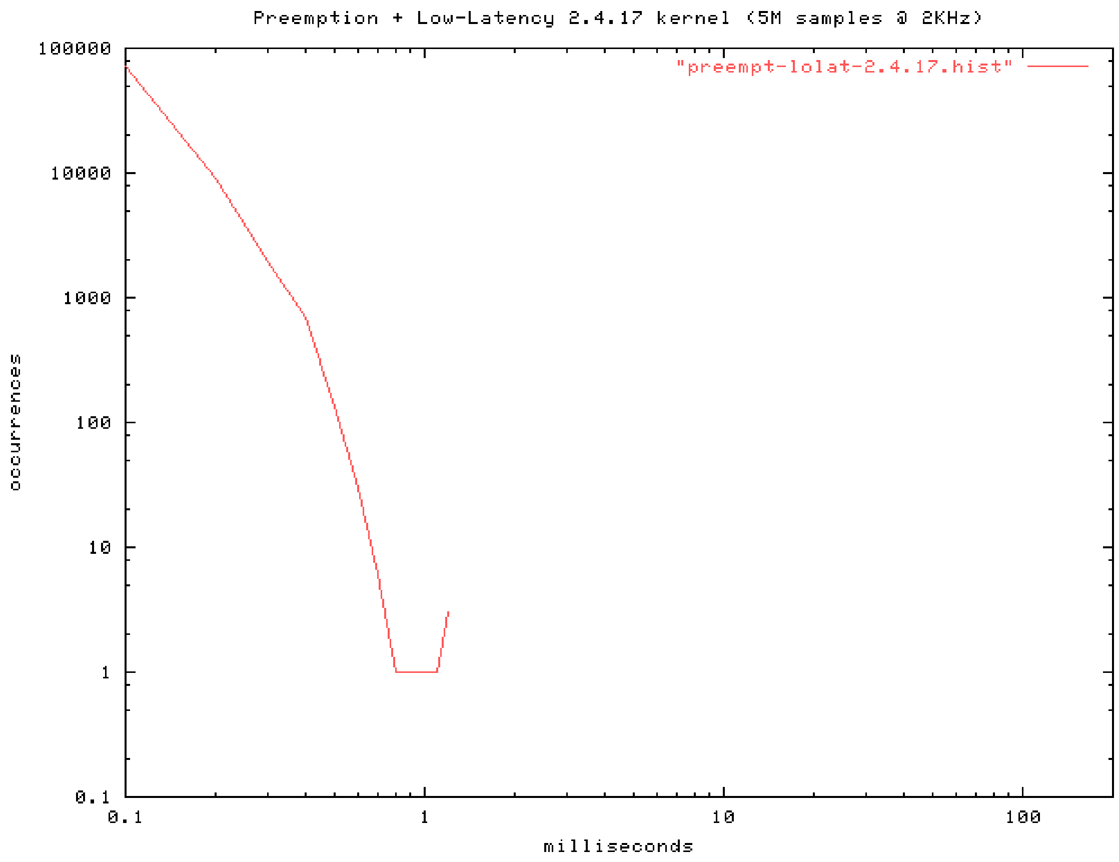
In marketing−speak terms, the vanilla kernel has five−9's worth of samples somewhere above 100ms, the preemption kernel has five−9's worth of samples below 5ms and the low−latency kernel has five−9's worth of samples below 700µs.

Here is a graph that compares the performance of the preemption patches versus the low−latency patches



Dogs and Cats, Living Together?

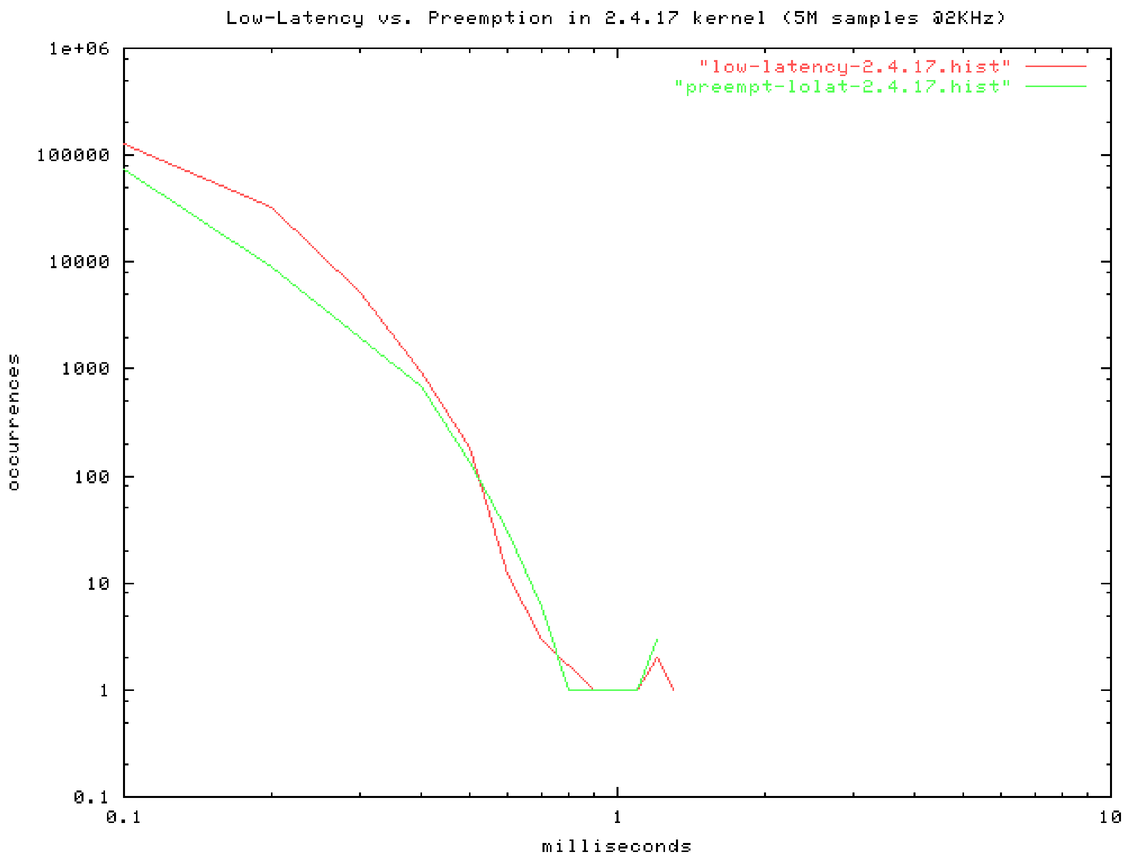
After spending a few weeks staring at the preemption and low−latency patches, it began to dawn on me that they didn’t intersect. By that, I mean that they touch different areas of kernel code and it looked like they could both be applied to the same kernel. So I did. I patched a 2.4.17 kernel with the preemption patch, then added the low−latency patches. After compiling the kernel, I ran it in my test setup and got the following results:



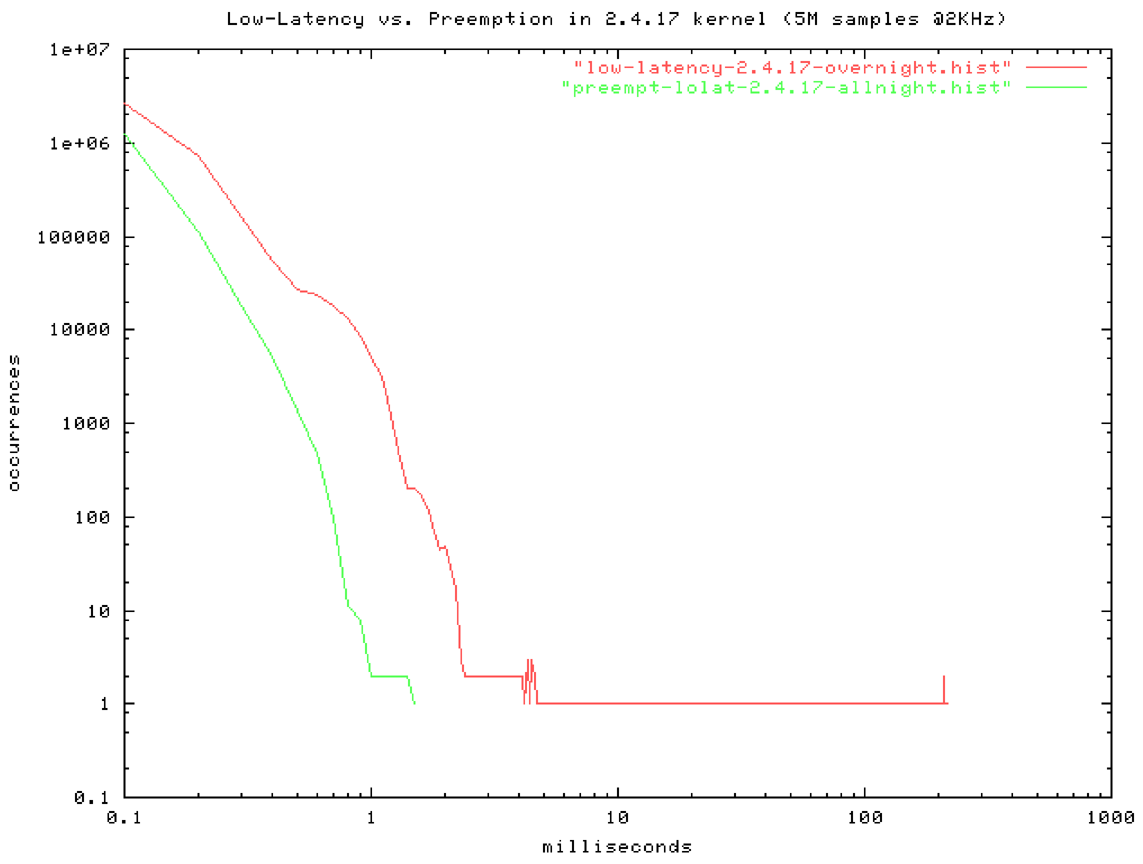
preempt−lolat−2.4.17.hist had 11 buckets containing 5000000 samples

minimum latency: < 0.1ms
maximum latency: 1.2ms
Mean: 0.0520061799956548ms
Median: 0.05ms
Mode: 0.05ms (occurred 4915634 times)
Variance: 0.000282321298762259
Standard Deviation: 0.0168024194318038
98.31268% of samples < 0.1ms
99.76028% of samples < 0.2ms
99.99648% of samples < 0.5ms
99.99978% of samples < 0.7ms
99.99992% of samples < 1ms
100.00000% of samples < 1.3ms

The combined patches actually improved scheduler latency slightly, reducing the maximum latency observed from 1.3ms in the low−latency kernel to 1.2ms in the combined preempt+low−latency patched kernel. While this was not as big a win as the gain from the low−latency patches, it did show that combining the patches was of some benefit. A comparison graph of the low latency performance versus the combined patch performance shows that the combined patches have a slightly tighter curve, bringing more latency values in under the 100µs mark.



Finally, I decided that I needed some runs that were longer than 41 minutes (five million samples at 2048 samples per second), to see if there are some latency conditions that arise after hundreds of millions of samples of the test program. I kicked off a run of the low−latency kernel that ran for 14.5 hours and saw that a corner case had indeed occurred, bumping the maximum latency to 215.2ms. The next night I ran the combined preempt+low−latency kernel for 15.5 hours and got a maximum latency of 1.5ms. The results are show in the following graph:



This graph shows that while the low latency patches give the best performance for effort, there are latency corner cases that have not been isolated. In addition, the increased potential for scheduling opportunities that the preemption patches introduce begin to show their worth as the kernel load increases. Even if the 200ms blip in the low−latency kernel is discounted, adding the preemption patches improves the performance curve by two or three milliseconds.

Conclusions

While both patches reduce latency in the kernel, the low−latency patches are the biggest single win. The biggest problem areas in delaying Linux scheduling decisions are large blocks of kernel code (usually loops) that delay scheduling opportunities. Finding these blocks and introducing preemption points is the most effective mechanism for improving kernel response time.

Using only the preemption patches is not as useful as using only the low−latency patches. However, combining the two patches increases the granularity of scheduler opportunities and seems to offset high latency areas of code as the kernel load increases over long periods of time.

As I was finishing up work on this article, I received word from Ingo Molnar that the principle authors of both patches (Robert Love and Andrew Morton) were working to unify their patches into a single latency patch, for possible inclusion into the 2.5 development kernel. It is unclear how much of this work will be back−ported to the 2.4 production series kernels, but he future of soft−realtime for Linux looks brighter now that everyone is pulling in the same direction.

Future Work

One thing that needs to be done is to repeat this series of measurements on something other than an x86 system. In particular, it should be done on the popular embedded processors, such as ARM, MIPS, PPC and SH. Doing this means solving two problems: creating a realistic load generator for a system with no disk and modifying the realfeel program to use a different time measurement mechanism. Work is currently in process to create a load generator for diskless systems and then the realfeel program will have to be ported to use a different time measurement mechanism.

Another experiment will be to use external instrumentation to measure scheduler latency, as well as other kernel performance values such as interrupt latency. This will involve using a function generator to provide an interrupt source and an ocilloscope to measure response time. A modified device driver and application pair will be needed to respond to interrupts and provide appropriate outputs to trigger the oscilloscope.