The Cursor Wiggles Faster: Measuring Scheduler Performance

Rick Lindsley
IBM Linux Technology Center
ricklind@us.ibm.com

Abstract

Trying to pin down whether changes to the 2.5 and 2.6 scheduler have helped or hurt performance, especially on interactive programs, has been both difficult to quantify and very subjective. One favored test has been to create your favorite load and then move your cursor around and observe how slow or fast it is. Another one is to drag a window across your desktop and see how quickly it gets redrawn. And I would certainly be skewered if I didn't mention what is probably the favorite: playing your favorite music while under load and listening intently for skips.

Unfortunately, all these measurements are subjective, and even, at times, argumentative. With scheduler statistics installed, one can accurately measure such things as the amount of time processes are spending on the processor or the amount of time they are waiting for the processor. This means that on SMP and NUMA machines, load balancing efforts can be objectively evaluated, and process migration decisions more effectively reviewed. And all of this can be done with no measurable impact to the system.

This paper will describe what information can be captured, use that information to characterize some simple loads, and describe how that same information may be coordinated with other system measurements both to characterize new loads, and to more clearly identify scheduler shortcomings.

1 Introduction

As the 2.5 code revisions came out in mid- to late 2003, the scheduler, like much of the 2.5 release, became more and more stable. True, there was still work to be done in some areas, like SMP and NUMA. Although an increasing number of dual-CPU desktops and even laptops introduced more users to the world of SMP, it was the high end users with 16, 32, 128, or even more CPUs that really were stretching the existing SMP and NUMA code. The increasing load on the existing infrastructure was causing developers to realize that code paths they previously thought "impossible" were really "rarely," and paths deemed "infrequent" were unfortunately morphing to "once or twice a day."

And an odd thing happened on the way to better code for the high end machines. Those pesky desktop and laptop users got in the way. With every fix that would demonstrably improve the situation for the big iron, dozens of desktop and laptop owners would immediately

pick up the new code, try it out, and more often than not, pronounce it faulty. Why? Because *their* 2-proc SMP machines were used very differently than the file servers and web servers that the 128-proc systems had become. The testing and measurements that had gone into verifying the patch did not test the system the same way the desktop users did. Consequently, these desktop users saw very different results, and formed very different opinions about the correctness and usefulness of these high-end SMP fixes.

And while their opinions mattered, of course, addressing their concerns was difficult. They were using human eyes and ears – notoriously unreliable biological components known to be fraught with frequent failure and highly subjective readouts – to detect problems with code. These observations needed to be backed up with numbers somehow.

2 Why is the wiggle so important?

So why weren't the big iron folks seeing the same problems as the desktop people if they were both utilizing the same code? The answer lay in usage patterns. People with laptops and desktops did not run two dozen instances of a server daemon that depended on ultra fast cache and great amounts of parallelism. They did not have petabytes of disk, and typically did not have gigabytes of memory either. They didn't read terabytes of disk per minute, nor expect to fully utilize their bus bandwidth on a regular basis.

These folks browsed the web, sorted mail, and compiled kernels while, in the background, they listened to their favorite playlist. While doing this, they would notice that with the new scheduler mods, their windows took longer to redraw. Or their cursor moved more sluggishly under this relatively heavy load. Or their music skipped now and then because their music player didn't get back on the CPU soon enough to catch the next few notes.

That's not to make light of their complaints; they were uncovering real problems that existing testing was inadequate to find. In fact, there were two main problems that needed to be solved. One was to close the testing hole by reliably repeating the tests that the desktop users were running, and repeating them on as wide a variety of hardware as the original patches had been run on. The other was that even the desktop users quibbled among themselves, sometimes, about whether wiggles, skips, and redraws had degraded. It was important to find a way to measure this "wiggle effect" in some quantifiable, objective way so you could reliably tell whether a new patch worsened it or improved it.

Server software, for its part, didn't need music to function, didn't need cursors to point with, and it sure didn't care how fast windows were redrawn. These highly interactive activities had no place in server evaluations. It was typically all about *throughput*, and placing stress on some subsystem or another: disk, memory, or network, typically. Stress on the scheduler was a given. Even though dozens of benchmarks exist for measuring the throughput of high-end machines, producing megabytes or even gigabytes of analysis and data, there was no easy way to automate the type of subjective human observation that desktop users were using. There was no way to have weekly regression tests pick it up, nor any way to precisely duplicate the environment in which these observations were being made. In short, there was no way to quantify the observations being made, so no existing tests could detect regressions in this area.

Previous scheduler modifications had labeled applications that tended to spend a lot of time

waiting for I/O as "interactive," and attempted to give scheduler bonuses to those tasks when the I/O they had been waiting for completed. This was *supposed* to provide the exact behavior the desktops were *not* seeing. The suspicion was that either these types of applications were not being correctly recognized, or they were not being given sufficient bonuses.

3 Isolating the wiggle

The first part of the solution was recognizing that the "wiggle effect" comes from tasks not regaining the CPU fast enough. The second part was recognizing that the audible stutter from a music player, or the delay in redrawing a window, were showing the same problem as wiggling the cursor.

In the case of a cursor, coordinates from a serial mouse are presented as a stream of input to the windowing system. If the task that moves the cursor is not brought to a CPU quickly enough, there will be a lag between the time the movement is initiated and the time it appears on the screen. With all the input consumed, the task again goes to sleep even though a split second later more input appears as the mouse continues to move. While this is an efficient way to handle a serial mouse, it is dependent on hitting the processor quickly enough to guarantee the input stream doesn't back up too much. If the consuming task does not get to run quickly enough, the cursor will appear to move across the screen in a staccato fashion, even though the mouse itself is being moved smoothly.

In the case of a music player, the application (say, *xmms*) will read a certain amount of input from a file, but it will take longer to play it to the speaker. Even though this is, in general, a very I/O-intensive task, there are times when *xmms* will go to sleep either waiting for output to drain to the speaker or input to come from the file. Waking up too slowly from these self-imposed interruptions is what causes the music to pause or stutter.

Slow window redrawing is a case of applications taking too long after notification to wake up and redraw. This *might* also be attributed to slow interprocess communication or slow signal delivery, but it should be easy to rule out these causes if we were to measure the time a task spent in a queue waiting for a processor.

A patch for scheduler statistics has been available since 2.5.59 [1]. However, it was with the 2.6.0-test5 release in September of 2003 that it was updated to include code to measure task latency. The task is given a new timestamp when it is placed in a run queue, placed on a processor, or removed from a processor. This makes it trivial to determine how long the task spent in the run queue before making it to the processor. It has the side effect of allowing us to also measure, on average, how long a task remains on the processor before relinquishing it, usually voluntarily. This allows us to easily characterize the kind of load a benchmark may place on a system.

Adding statistics counting to the scheduler path was a dicey task. This is one of the most heavily used paths in the system, and anything that slows down this path can have a catastrophic effect on the system as a whole. Consequently, the statistics patch tries to do what it can to gather accurate statistics without the use of a lock.

- Per-CPU counters are used, and incremented only by their respective CPU. This makes update collisions (and loss of data) impossible.

[1] http://oss.software.ibm.com/developerworks/opensource/linux/patches/?patch_id=730

- Even so, when possible, these counters are incremented while a per-CPU runqueue lock is already acquired.
- Counters are only incremented, so minor variations from unflushed caches that may be observed while reading another CPU's counters can be safely ignored. (The counters are declared unsigned long, so user-level utilities on 32-bit architectures must take note that the counters could wrap. While theoretically possible on 64-bit machines, wrapping is far less likely than on 32-bit machines.)

Measurements were taken across several different releases using several different benchmarks to see if any statistical impact could be found on the benchmarks when scheduler statistics were utilized. To date, none have been found.

After the patch is applied, the counters can be obtained by reading `/proc/schedstat`. A full description of the statistics collected can be found in `Documentation/schedstats.txt` in the kernel source. The patch itself introduces a config option SCHEDSTATS that is on by default; if it is turned off, all the additional code is compiled out. There are three important fields:

timestamp *N*

This line indicates a timestamp, in jiffies, of when this output was produced. The statistics are most effectively utilized when collected at small regular intervals, since this allows you to more accurately see how the behavior of a load or benchmark may change over its lifetime. Any process reading this file, however, is subject to the same scheduler delays it is trying to measure. Consequently, a simple script like

```
while true
do
    sleep 10
    cat /proc/schedstats >> \
        /tmp/stat.out
done
```

may find it collects statistics roughly every 10 seconds when the system is lightly loaded, but every 15-20 seconds or more when the system is heavily loaded. The code to note the timestamp is just a few lines before the data is totaled in the kernel, and on a non-preemptible kernel is an inexpensive way of identifying the time at which the snapshot was *actually* taken.

cpu*N n n n n n n n n ...*

These are the values of the counters for cpu ***N***. The precise meaning of these counters will vary depending on the version of scheduler statistics being utilized. A few examples of data collected are:

1. number of times some functions were called
2. number of times certain functions were called under certain circumstances (i.e., were the runqueues unbalanced? was this processor idle?)
3. total number of milliseconds that tasks on this processor have used, not including the current one
4. total number of milliseconds that tasks that ran here had to wait in queue

version *N*

identifies the version of output being produced. Since the meaning of fields (and the number of fields) in the **cpu*N*** line, above, can vary in different versions of scheduler statistics, this allows tools to be as flexible or inflexible as desired when processing input.

A sample of the output from `/proc/schedstat` is provided in Appendix A.

4 What would I use statistics for?

Scheduler statistics can serve three basic purposes. In many cases, they are doing no more than providing some detailed code path and profiling data. Knowing, for instance, that a particular function was called 50,000 times during a benchmark run may be key if it is expected to be called a dozen times – or a million. Similarly, knowing that 22,000 of those calls were made while the processor was idle, or made on just one of eight CPUs, may also be quite informative. About half the counters provide this sort of information, and it must be coupled with a knowledge of what to expect given your workload in order to detect anomalies.

Another purpose is to provide information beyond just counting. There is a counter that sums the imbalance found when queues are inspected. Combine this with the number of times you called this function and you can determine the average imbalance between runqueues. In most cases you wouldn't want this to exceed 1. Truth is, though, that a flurry of forking or even I/O completions might suddenly cause a processor to suddenly find itself with significantly more runnable tasks than other processors. Seeing where these spikes happen during the test run, and how often they happen, may help to suggest better "default" behavior in the scheduler or even tuning in the benchmark itself.

The last purpose has already been mentioned – task latency. We already need to note when a task is queued on a processor and when it acquires a processor. By noting one more thing – when it leaves the processor – we can also determine what I call the *runslice*.

The runslice is the amount of time a task spends *on* the processor before yielding it. In contrast, the *timeslice* alloted by the scheduler indicates how long the task may run before it is *forced* off. Processes are usually given generous timeslices (100 ms is the default) but typically don't use all of them at one shot. A task may need to put itself to sleep, perhaps to wait for input, before it has used up that full 100 ms. It will have any unused amount available to it when the event awakens it, but how long it spends on the processor can be an important characteristic of the system load. If a task spends only a few milliseconds before giving up the processor, it may be I/O-bound. By the same token, if it uses its full timeslice every time before being kicked off, then it is CPU-bound.

While many benchmarks are already characterized as CPU- or I/O-bound, they are rarely that way from beginning to end. Seeing this behavior graphed over a period of time can be very informative to a person trying to tune the system or the benchmark.

5 Diagnostic examples

The data that the scheduler statistics collect can be utilized in several different ways.

5.1 Using the function counts to characterize behavior

Recently a colleague remarked that he was running a benchmark that he expected to fully load a machine; yet profiling was reporting that the system was in the idle routine 50% of the time. He increased the load significantly on the machine and idle time only dropped to 49%. He

couldn't believe the machine still had spare cycles, so we used the scheduler statistics to determine what was happening.

From the beginning of the benchmark, we captured the counters in `/proc/schedstat` every 10 seconds with a shell script. When the benchmark exited, we killed the shell script.

The two pieces of information that proved most useful were the number of calls per second (*cps*) for `load_balance()` and `sched_balance_exec()`. In Figure 1, you can see that the *cps* for `load_balance()` varies markedly between plateaus of around 4000-4500, and valleys of 100-200. When the system is idle, it calls `load_balance()` as often as once a millisecond to try to find work. When it is busy, it backs off to five times a second. The graph here is clearly indicating that this benchmark has at least two periods of about 100 seconds each out of about 450 seconds total where it is largely idle.

At about the same time that the *cps* for `load_balance()` is high, the *cps* for `sched_balance_exec()` is low. This function is called when tasks issue the `exec()` system call, and is used to do some opportunistic rebalancing. We observed that just as the system starts to get busy, `sched_balance_exec()` tails off.

The data suggested that this benchmark had a notable rampup and cooldown period. With this information in hand, simple observation of *top(1)* while running the benchmark confirmed what the scheduler statistics suggested. The benchmark had a fairly lengthy single-threaded setup: creating log files, making directories for results, and compiling short programs it would use. It then forked many tasks and set them all running to actually start the benchmark. When the test was over, there was again a single threaded task that collected the data created before several tasks organized the data.

5.2 Using latency and runslice information

In another situation, a disk-intensive benchmark was doing much worse with a different version of the scheduler. Figure 2 shows a measurement of the latency from the two runs.

In the "broken" run, the latencies were nearly twice that of the "working" run. Tasks were taking longer to reach the CPU in the broken case. Yet the runslice information shows comparable (and very short) times spent on the CPUs. If tasks were running very short periods of time, but waiting longer to run, what could have been the cause?

Enlightenment was finally attained by viewing the average imbalance (Figure 3) during each of the runs. On the average, the imbalance was twice as great in the broken run as in the working run. Since the runslice was so small, this suggested that tasks were becoming runnable quickly but simply not being balanced often enough. Some queues were getting quite long while others (presumably) were staying short. Additional debugging showed that tasks were indeed awakening (probably by completed I/O) quite frequently but most of the balancing was happening only when one CPU fell idle and went looking for work. These longer queues in the broken run were persisting longer than those in the working run, and tasks stuck in them were waiting a fraction of a millisecond longer than before.

6 Conclusion

There is still work to do.

Recent scheduler changes present in Andrew Morton's -mm tree will dramatically change what is important to measure in the scheduler. Additionally, these same changes introduce some self-tuning characteristics which

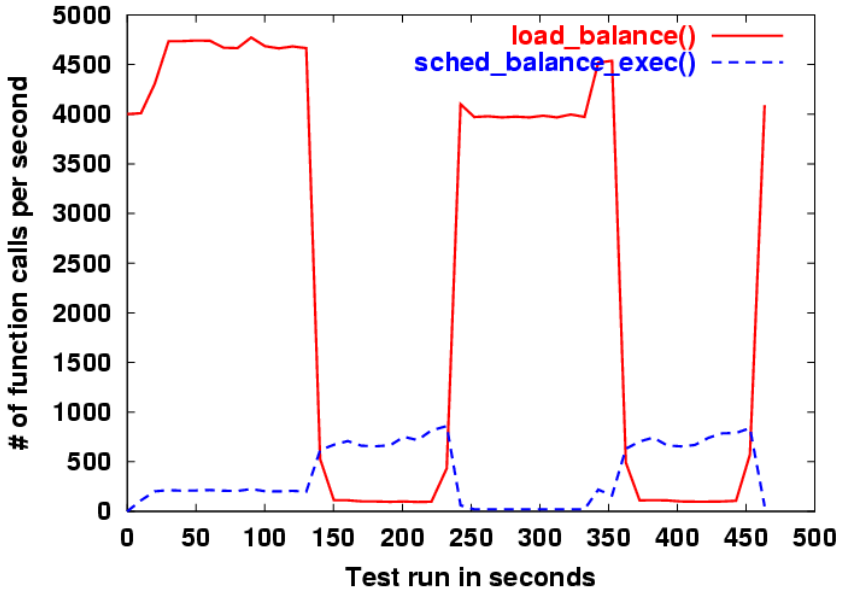

Figure 1: `load_balance()` and `sched_balance_exec()` counts

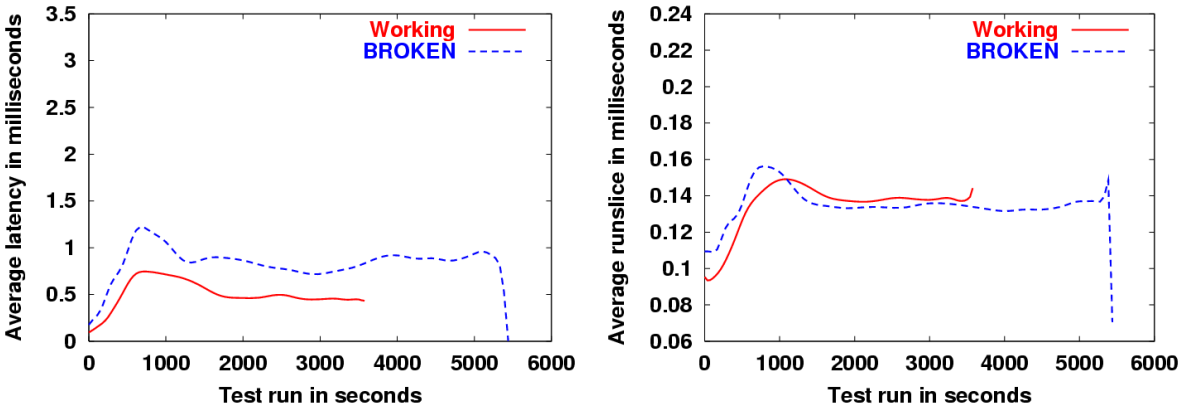

Figure 2: Latency and runslice duration

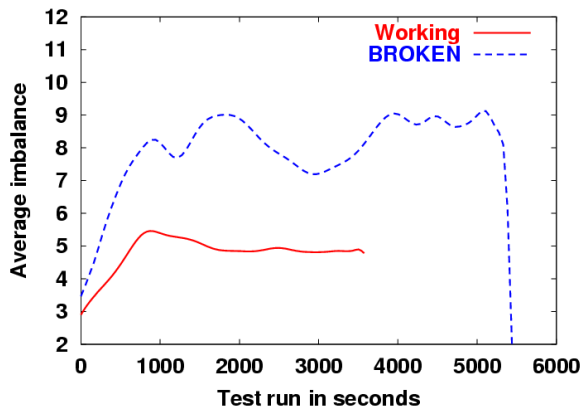

Figure 3: Average load imbalances

may benefit from statistics describing how often they are retuned.

There is also some evidence that NUMA machines may benefit from device, task, or memory affinitization strategies which try to keep data from crossing NUMA node boundaries. Scheduler statistics can be used to reliably demonstrate whether these strategies are being effective.

Lastly, the data provided by scheduler statistics probably ought to be moved out of /proc eventually, as there is an ongoing effort to return /proc to its original task of just listing processes.

Scheduler statistics provide a quantifiable means of measuring scheduler changes. Much as disk statistics can be used to a variety of ends – measuring disk utilization, throughput rates, and transfer rates, for example – scheduler statistics can help with analysis of a variety of situations. The latest revisions go to lengths to avoid creating "Heisenbugs," or bugs which disappear when you try to examine them closely. Perhaps best of all, developers need not rely on mice and windowing systems to measure their test results. Latency numbers, in particular, provide a key way of measuring scheduler success, and runslice figures can help characterize the load that tests create so that the best set of tests can be chosen to test a particular feature or system. Cursor wiggles and audible skips can be set aside until they are needed again.

Disclaimer

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

IBM is registered trademark of International Business Machines Corporation in the United States and/or other countries worldwide.

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

Appendix A

Table 1 is a sample of what `/proc/schedstat` might look like for a 2-proc machine. The actual format and number of counters will vary between different versions. For purposes of this example, the last three lines are artificially folded for readability, but in actual output, each would be one long line.

This is a brief description of each of the 23 counters for version 4 output. Applications can check the `version` field to make sure they look for and correctly interpret the counters. Note that all counters may wrap back to zero, and applications using these counters should be prepared to deal with that. Since all counters start at zero at boot time, the most useful way to use them is to get periodic snapshots of the counters, then subtract one set from a previously obtained one to obtain the delta. All counters are per-processor.

```
version 4
timestamp 4295814751
cpu0 8909 9103 612 11869 264585 9821 392921 1065335 406 140662 1206403 62905
1192940 0 13440 13469 0 0 0 0 82278 1497607 264615
cpu1 5138 5328 577 8126 265205 6270 402877 943453 1005 149999 1094457 77670
1074828 0 13469 13440 0 0 0 0 200998 448842 265175
totals 14047 14431 1189 19995 529790 16091 795798 2008788 1411 290661 2300860
140575 2267768 0 26909 26909 0 0 0 0
```

Table 1: Sample output from `/proc/schedstat`

1. in `sched_yield()`, number of times both the active and the expired queue were empty
2. in `sched_yield()`, number of times just the active queue was empty
3. in `sched_yield()`, number of times just the expired queue was empty
4. in `sched_yield()`, number of times `sched_yield()` was called
5. in `schedule()`, number of times the active queue had at least one other task on it
6. in `schedule()`, number of times we switched to the expired queue and reused it
7. number of times `schedule()` was called
8. number of times `load_balance()` was called at an idle tick
9. number of times `load_balance()` was called at a busy tick
10. number of times `load_balance()` was called from `schedule()`
11. number of times `load_balance()` was called
12. sum of imbalances discovered (if any) with each call to `load_balance()`
13. number of times `load_balance()` was called when we did not find a "busiest" queue
14. number of times `load_balance()` was called from `balance_node()`
15. number of times `pull_task()` moved a task to this cpu
16. number of times `pull_task()` stole a task from this cpu
17. number of times `pull_task()` moved a task to this cpu from another node (requires CONFIG_NUMA)
18. number of times `pull_task()` stole a task from this cpu for another node (requires CONFIG_NUMA)
19. number of times `balance_node()` was called
20. number of times `balance_node()` was called at an idle tick
21. sum of all time spent running by tasks (in ms)
22. sum of all time spent waiting by tasks (in ms)
23. number of tasks (not necessarily unique) given to the processor

The last three make it possible to find the average latency on a particular runqueue or, if taken from the `totals` fields, the overall system. Given two points in time, A and B, (22B - 22A)/(23B - 23A) will give you the average time tasks had to wait after being scheduled to run but before actually running.

`/proc/<pid>/stat`

This version of the patch also changes the `stat` output of *individual tasks* to include the same latency and runslice information described above. Three new fields, corresponding to the last three fields described above, are added to the end of the per-task `stat` file, but apply only for that task rather than a whole processor.