

## [PATCH -v2 0/7] New RT Task Balancing -v2

---

*Source:* <http://linux.derkeiler.com/Mailing-Lists/Kernel/2007-10/msg07901.html>

---

- *From:* Steven Rostedt <rostedt@xxxxxxxxxxxx>
  - *Date:* Mon, 22 Oct 2007 22:59:00 -0400
- 

[  
Changes since V1:  
Updated to git tree 55b70a0300b873c0ec7ea6e33752af56f41250ce

Various clean ups suggested by Gregory Haskins, Dmitry Adamushko,  
and Peter Zijlstra.

Biggest change was recommended by Ingo Molnar. This is the use of cpusets  
for keeping track of RT overloaded CPUS. When CONFIG\_CPUSETS is not  
defined, we have a single global rt\_overload\_mask that keeps track  
of the runqueues with more than one RT task queued. When CONFIG\_CPUSETS  
is configured, that bitmask is stored in the cpusets.

Note that in the case of overlapping cpusets it is possible to have  
inconsistent data between the bitmask and actual RT overloaded runqueues.  
The worst that can happen is that a task doesn't get moved quickly  
over to a runnable CPU. But this is a price we pay to keep from  
dirtying caches for large number of CPU boxes. If this does happen  
it gets cleaned up rather quickly since there are checks for  
RT overload bits being set when they shouldn't be.

For most systems this is not an issue since a single cpuset is used.  
]

Currently in mainline the balancing of multiple RT threads is quite broken.  
That is to say that a high priority thread that is scheduled on a CPU  
with a higher priority thread, may need to unnecessarily wait while it  
can easily run on another CPU that's running a lower priority thread.

Balancing (or migrating) tasks in general is an art. Lots of considerations  
must be taken into account. Cache lines, NUMA and more. This is true  
with general processes which expect high throughput and migration can  
be done in batch. But when it comes to RT tasks, we really need to  
put them off to a CPU that they can run on as soon as possible. Even  
if it means a bit of cache line flushing.

Right now an RT task can wait several milliseconds before it gets scheduled  
to run. And perhaps even longer. The migration thread is not fast enough  
to take care of RT tasks.

To demonstrate this, I wrote a simple test.

<http://rostedt.homelinux.com/rt/rt-migrate-test.c>

(gcc -o rt-migrate-test rt-migrate-test.c -lpthread)

This test expects a parameter to pass in the number of threads to create. If you add the '-c' option (check) it will terminate if the test fails one of the iterations. If you add this, pass in +1 threads.

For example, on a 4 way box, I used

```
rt-migrate-test -c 5
```

What this test does is to create the number of threads specified (in this case 5). Each thread is set as an RT FIFO task starting at a specified prio (default 2) and each thread being one priority higher. So with this example the 5 threads created are at priorities 2, 3, 4, 5, and 6.

The parent thread sets its priority to one higher than the highest of the children (this example 7). It uses pthread\_barrier\_wait to synchronize the threads. Then it takes a time stamp and starts all the threads.

The threads when woken up take a time stamp and compares it to the parent thread to see how long it took to be awoken. It then runs for an interval (20ms default) in a busy loop. The busy loop ends when it reaches the interval delta from the start time stamp. So if it is preempted, it may not actually run for the full interval. This is expected behavior of the test.

The numbers recorded are the delta from the thread's time stamp from the parent time stamp. The number of iterations it ran the busy loop for, and the delta from a thread time stamp taken at the end of the loop to the parent time stamp.

Sometimes a lower priority task might wake up before a higher priority, but this is OK, as long as the higher priority process gets the CPU when it is awoken.

At the end of the test, the iteration data is printed to stdout. If a higher priority task had to wait for a lower one to finish running, then this is considered a failure. Here's an example of the output from a run against git commit 4fa4d23fa20de67df919030c1216295664866ad7.

```
1: 36 33 20041 39 33
len: 20036 20033 40041 20039 20033
loops: 167789 167693 227167 167829 167814
```

On iteration 1 (starts at 0) the third task started at 20ms after the parent woke it up. We can see here that the first two tasks ran to completion before the higher priority task was even able to start. That is a

20ms latency for the higher priority task!!!

So people who think that their audio would lose most latencies by upping the priority, may be in for a surprise. Since some kernel threads (like the migration thread itself) may cause this latency.

To solve this issue, I've changed the RT task balancing from a passive method (migration thread) to an active method. This new method is to actively push or pull RT tasks when they are woken up or scheduled.

On wake up of a task if it is an RT task, and there's already an RT task of higher priority running on its runqueue, we initiate a `push_rt_tasks` algorithm. This algorithm looks at the highest non-running RT task and tries to find a CPU where it can run on. It only migrates the RT task if it finds a CPU (of lowest priority) where the RT task can run on and can preempt the currently running task on that CPU. We continue pushing RT tasks until we can't push anymore.

If a RT task fails to be migrated we stop the pushing. This is possible because we are always looking at the highest priority RT task on the run queue. And if it can't migrate, then most likely the lower RT tasks can not either.

There is one case that is not covered by this patch set. That is that when the highest priority non running RT task has its CPU affinity in such a way that it can not preempt any tasks on the CPUs running on CPUs of its affinity. But a lower priority task has a larger affinity to CPUs that it can run on. This is a case where the lower priority task will not be migrated to those CPUS (although those CPUs may pull that task). Currently this patch set ignores this scenario.

Another case where we push RT tasks is in the `finish_task_switch`. This is done since the running RT task can not be migrated while it is running. So if an RT task is preempted by a higher priority RT task, we can migrate the RT task being preempted at that moment.

We also actively pull RT tasks. Whenever a runqueue is about to lower its priority (schedule a lower priority task) a check is done to see if that runqueue can pull RT tasks to it to run instead. A search is made of all overloaded runqueues (runqueues with more than one RT task scheduled on it) and checked to see if they have an RT task that can run on the runqueue (affinity matches) and is of higher priority than the task the runqueue is about to schedule. The pull algorithm pulls all RT tasks that match this case.

With this patch set, I ran the `rt-migrate-test` over night in a while loop with the `-c` option (which terminates upon failure) and it passed over 6500 tests (each doing 50 wakeups each).

Here's an example of the output from the patched kernel. This is just to explain it a bit more.

[PATCH -v2 0/7] New RT Task Balancing -v2

1: 20060 61 55 56 61  
len: 40060 20061 20055 20056 20061  
loops: 227255 146050 168104 145965 168144

2: 40 46 31 35 42  
len: 20040 20046 20031 20035 20042  
loops: 28 167769 167781 167668 167804

The first iteration (really 2cd, since we start at zero), is a typical run. The lowest prio task didn't start executing until the other 4 tasks finished and gave up the CPU.

The second iteration seems at first like a failure. But this is actually fine. The lowest priority task just happen to schedule onto a CPU before the higher priority tasks were woken up. But as you can see from this example, the higher priority tasks still were able to get scheduled right away and in doing so preempted the lower priority task. This can be seen by the number of loops that the lower priority task was able to complete. Only 28. This is because the busy loop terminates when the time stamp reaches the time interval (20ms here) from the start time stamp. Since the lower priority task was able to sneak in and start, it's time stamp was low. So after it got preempted, and rescheduled, it was already past the run time interval so it simply ended the loop.

Finally, the CFS RT balancing had to be removed in order for this to work. Testing showed that the CFS RT balancing would actually pull RT tasks from runqueues already assigned to the proper runqueues, and again cause latencies. With this new approach, the CFS RT balancing is not needed, and I suggest that these patches replace the current CFS RT balancing.

Also, let me stress, that I made a great attempt to have this cause as little overhead (practically none) to the non RT cases. Most of these algorithms only take place when more than one RT task is scheduled on the same runqueue.

Special thanks goes to Gregory Haskins for his advice and back and forth on IRC with ideas. Although I didn't use his actual patches (his were against -rt) it did help me clean up some of my code. Also, thanks go to Ingo Molnar himself for taking some ideas from his RT balance code in the -rt patch.

Comments welcomed!

— Steve

—

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@xxxxxxxxxxxxxxxxx  
More majordomo info at <http://vger.kernel.org/majordomo-info.html>

[PATCH -v2 0/7] New RT Task Balancing -v2

Please read the FAQ at <http://www.tux.org/lkml/>