NO_HZ: Reducing Scheduling-Clock Ticks

This document describes Kconfig options and boot parameters that can reduce the number of scheduling-clock interrupts, thereby improving energy efficiency and reducing OS jitter. Reducing OS jitter is important for some types of computationally intensive high-performance computing (HPC) applications and for real-time applications.

There are three main ways of managing scheduling-clock interrupts (also known as "scheduling-clock ticks" or simply "ticks"):

1. Never omit scheduling-clock ticks (CONFIG_HZ_PERIODIC=y or CONFIG_NO_HZ=n for older kernels). You normally will -not- want to choose this option.
2. Omit scheduling-clock ticks on idle CPUs (CONFIG_NO_HZ_IDLE=y or CONFIG_NO_HZ=y for older kernels). This is the most common approach, and should be the default.
3. Omit scheduling-clock ticks on CPUs that are either idle or that have only one runnable task (CONFIG_NO_HZ_FULL=y). Unless you are running realtime applications or certain types of HPC workloads, you will normally -not- want this option.

These three cases are described in the following three sections, followed by a third section on RCU-specific considerations, a fourth section discussing testing, and a fifth and final section listing known issues.

Never Omit Scheduling-Clock Ticks

Very old versions of Linux from the 1990s and the very early 2000s are incapable of omitting scheduling-clock ticks. It turns out that there are some situations where this old-school approach is still the right approach, for example, in heavy workloads with lots of tasks that use short bursts of CPU, where there are very frequent idle periods, but where these idle periods are also quite short (tens or hundreds of microseconds). For these types of workloads, scheduling clock interrupts will normally be delivered any way because there will frequently be multiple runnable tasks per CPU. In these cases, attempting to turn off the scheduling clock interrupt will have no effect other than increasing the overhead of switching to and from idle and transitioning between user and kernel execution.

This mode of operation can be selected using CONFIG_HZ_PERIODIC=y (or CONFIG_NO_HZ=n for older kernels).

However, if you are instead running a light workload with long idle periods, failing to omit scheduling-clock interrupts will result in excessive power consumption. This is especially bad on battery-powered devices, where it results in extremely short battery lifetimes. If you are running light workloads, you should therefore read the following section.

In addition, if you are running either a real-time workload or an HPC workload with short iterations, the scheduling-clock interrupts can degrade your applications performance. If this describes your workload, you should read the following two sections.

Omit Scheduling-Clock Ticks For Idle CPUs

If a CPU is idle, there is little point in sending it a scheduling-clock interrupt. After all, the primary purpose of a scheduling-clock interrupt is to force a busy CPU to shift its attention among multiple duties, and an idle CPU has no duties to shift its attention among.

An idle CPU that is not receiving scheduling-clock interrupts is said to be "dyntick-idle", "in dyntick-idle mode", "in nohz mode", or "running tickless". The remainder of this document will use "dyntick-idle mode".

The CONFIG_NO_HZ_IDLE=y Kconfig option causes the kernel to avoid sending scheduling-clock interrupts to idle CPUs, which is critically important both to battery-powered devices and to highly virtualized mainframes. A battery-powered device running a CONFIG_HZ_PERIODIC=y kernel would drain its battery very quickly, easily 2-3 times as fast as would the same device running a CONFIG_NO_HZ_IDLE=y kernel. A mainframe running 1,500 OS instances might find that half of its CPU time was consumed by unnecessary scheduling-clock interrupts. In these situations, there is strong motivation to avoid sending scheduling-clock interrupts to idle CPUs. That said, dyntick-idle mode is not free:

1. It increases the number of instructions executed on the path to and from the idle loop.
2. On many architectures, dyntick-idle mode also increases the number of expensive clock-reprogramming operations.

Therefore, systems with aggressive real-time response constraints often run CONFIG_HZ_PERIODIC=y kernels (or CONFIG_NO_HZ=n for older kernels) in order to avoid degrading from-idle transition latencies.

There is also a boot parameter "nohz=" that can be used to disable dyntick-idle mode in CONFIG_NO_HZ_IDLE=y kernels by specifying "nohz=off". By default, CONFIG_NO_HZ_IDLE=y kernels boot with "nohz=on", enabling dyntick-idle mode.

Omit Scheduling-Clock Ticks For CPUs With Only One Runnable Task

If a CPU has only one runnable task, there is little point in sending it a scheduling-clock interrupt because there is no other task to switch to. Note that omitting scheduling-clock ticks for CPUs with only one runnable task implies also omitting them for idle CPUs.

The CONFIG_NO_HZ_FULL=y Kconfig option causes the kernel to avoid sending scheduling-clock interrupts to CPUs with a single runnable task, and such CPUs are said to be "adaptive-ticks CPUs". This is important for applications with aggressive real-time response constraints because it allows them to improve their worst-case response times by the maximum duration of a scheduling-clock interrupt. It is also important for computationally intensive short-iteration workloads: If any CPU is delayed during a given

iteration, all the other CPUs will be forced to wait idle while the delayed CPU finishes. Thus, the delay is multiplied by one less than the number of CPUs. In these situations, there is again strong motivation to avoid sending scheduling-clock interrupts.

By default, no CPU will be an adaptive-ticks CPU. The "nohz_full=" boot parameter specifies the adaptive-ticks CPUs. For example, "nohz_full=1,6-8" says that CPUs 1, 6, 7, and 8 are to be adaptive-ticks CPUs. Note that you are prohibited from marking all of the CPUs as adaptive-tick CPUs: At least one non-adaptive-tick CPU must remain online to handle timekeeping tasks in order to ensure that system calls like gettimeofday() returns accurate values on adaptive-tick CPUs. (This is not an issue for CONFIG_NO_HZ_IDLE=y because there are no running user processes to observe slight drifts in clock rate.) Therefore, the boot CPU is prohibited from entering adaptive-ticks mode. Specifying a "nohz_full=" mask that includes the boot CPU will result in a boot-time error message, and the boot CPU will be removed from the mask. Note that this means that your system must have at least two CPUs in order for CONFIG_NO_HZ_FULL=y to do anything for you.

Finally, adaptive-ticks CPUs must have their RCU callbacks offloaded. This is covered in the "RCU IMPLICATIONS" section below.

Normally, a CPU remains in adaptive-ticks mode as long as possible. In particular, transitioning to kernel mode does not automatically change the mode. Instead, the CPU will exit adaptive-ticks mode only if needed, for example, if that CPU enqueues an RCU callback.

Just as with dyntick-idle mode, the benefits of adaptive-tick mode do not come for free:

1. CONFIG_NO_HZ_FULL selects CONFIG_NO_HZ_COMMON, so you cannot run adaptive ticks without also running dyntick idle. This dependency extends down into the implementation, so that all of the costs of CONFIG_NO_HZ_IDLE are also incurred by CONFIG_NO_HZ_FULL.
2. The user/kernel transitions are slightly more expensive due to the need to inform kernel subsystems (such as RCU) about the change in mode.
3. POSIX CPU timers prevent CPUs from entering adaptive-tick mode. Real-time applications needing to take actions based on CPU time consumption need to use other means of doing so.
4. If there are more perf events pending than the hardware can accommodate, they are normally round-robined so as to collect all of them over time. Adaptive-tick mode may prevent this round-robining from happening. This will likely be fixed by preventing CPUs with large numbers of perf events pending from entering adaptive-tick mode.
5. Scheduler statistics for adaptive-tick CPUs may be computed slightly differently than those for non-adaptive-tick CPUs. This might in turn perturb load-balancing of real-time tasks.

Although improvements are expected over time, adaptive ticks is quite useful for many types of real-time and compute-intensive applications. However, the drawbacks listed above mean that adaptive ticks should not (yet) be enabled by default.

RCU Implications

There are situations in which idle CPUs cannot be permitted to enter either dyntick-idle mode or adaptive-tick mode, the most common being when that CPU has RCU callbacks pending.

Avoid this by offloading RCU callback processing to "rcuo" kthreads using the CONFIG_RCU_NOCB_CPU=y Kconfig option. The specific CPUs to offload may be selected using The "rcu_nocbs=" kernel boot parameter, which takes a comma-separated list of CPUs and CPU ranges, for example, "1,3-5" selects CPUs 1, 3, 4, and 5. Note that CPUs specified by the "nohz_full" kernel boot parameter are also offloaded.

The offloaded CPUs will never queue RCU callbacks, and therefore RCU never prevents offloaded CPUs from entering either dyntick-idle mode or adaptive-tick mode. That said, note that it is up to userspace to pin the "rcuo" kthreads to specific CPUs if desired. Otherwise, the scheduler will decide where to run them, which might or might not be where you want them to run.

Testing

So you enable all the OS-jitter features described in this document, but do not see any change in your workload's behavior. Is this because your workload isn't affected that much by OS jitter, or is it because something else is in the way? This section helps answer this question by providing a simple OS-jitter test suite, which is available on branch master of the following git archive:

git://git.kernel.org/pub/scm/linux/kernel/git/frederic/dynticks-testing.git

Clone this archive and follow the instructions in the README file. This test procedure will produce a trace that will allow you to evaluate whether or not you have succeeded in removing OS jitter from your system. If this trace shows that you have removed OS jitter as much as is possible, then you can conclude that your workload is not all that sensitive to OS jitter.

Note: this test requires that your system have at least two CPUs. We do not currently have a good way to remove OS jitter from single-CPU systems.

Known Issues

- Dyntick-idle slows transitions to and from idle slightly. In practice, this has not been a problem except for the most aggressive real-time workloads, which have the option of disabling dyntick-idle mode, an option that most of them take. However, some workloads will no doubt want to use adaptive ticks to eliminate scheduling-clock interrupt latencies. Here are some options for

these workloads:

a. Use PMQOS from userspace to inform the kernel of your latency requirements (preferred).
b. On x86 systems, use the "idle=mwait" boot parameter.

c. On x86 systems, use the "intel_idle.max_cstate=" to limit ` the maximum C-state depth.

d. On x86 systems, use the "idle=poll" boot parameter. However, please note that use of this parameter can cause your CPU to overheat, which may cause thermal throttling to degrade your latencies -- and that this degradation can be even worse than that of dyntick-idle. Furthermore, this parameter effectively disables Turbo Mode on Intel CPUs, which can significantly reduce maximum performance.

- Adaptive-ticks slows user/kernel transitions slightly. This is not expected to be a problem for computationally intensive workloads, which have few such transitions. Careful benchmarking will be required to determine whether or not other workloads are significantly affected by this effect.
- Adaptive-ticks does not do anything unless there is only one runnable task for a given CPU, even though there are a number of other situations where the scheduling-clock tick is not needed. To give but one example, consider a CPU that has one runnable high-priority SCHED_FIFO task and an arbitrary number of low-priority SCHED_OTHER tasks. In this case, the CPU is required to run the SCHED_FIFO task until it either blocks or some other higher-priority task awakens on (or is assigned to) this CPU, so there is no point in sending a scheduling-clock interrupt to this CPU. However, the current implementation nevertheless sends scheduling-clock interrupts to CPUs having a single runnable SCHED_FIFO task and multiple runnable SCHED_OTHER tasks, even though these interrupts are unnecessary.

  And even when there are multiple runnable tasks on a given CPU, there is little point in interrupting that CPU until the current running task's timeslice expires, which is almost always way longer than the time of the next scheduling-clock interrupt.

  Better handling of these sorts of situations is future work.
- A reboot is required to reconfigure both adaptive idle and RCU callback offloading. Runtime reconfiguration could be provided if needed, however, due to the complexity of reconfiguring RCU at runtime, there would need to be an earthshakingly good reason. Especially given that you have the straightforward option of simply offloading RCU callbacks from all CPUs and pinning them where you want them whenever you want them pinned.
- Additional configuration is required to deal with other sources of OS jitter, including interrupts and system-utility tasks and processes. This configuration normally involves binding interrupts and tasks to particular CPUs.
- Some sources of OS jitter can currently be eliminated only by constraining the workload. For example, the only way to eliminate OS jitter due to global TLB shootdowns is to avoid the unmapping operations (such as kernel module unload operations) that result in these shootdowns. For another example, page faults and TLB misses can be reduced (and in some cases eliminated) by using huge pages and by constraining the amount of memory used by the application. Pre-faulting the working set can also be helpful, especially when combined with the mlock() and mlockall() system calls.
- Unless all CPUs are idle, at least one CPU must keep the scheduling-clock interrupt going in order to support accurate timekeeping.
- If there might potentially be some adaptive-ticks CPUs, there will be at least one CPU keeping the scheduling-clock interrupt going, even if all CPUs are otherwise idle.

  Better handling of this situation is ongoing work.
- Some process-handling operations still require the occasional scheduling-clock tick. These operations include calculating CPU load, maintaining sched average, computing CFS entity vruntime, computing avenrun, and carrying out load balancing. They are currently accommodated by scheduling-clock tick every second or so. On-going work will eliminate the need even for these infrequent scheduling-clock ticks.