CPU frequency and voltage scaling code in the Linux(TM) kernel

Linux CPUFreq

CPUFreq Governors

- information for users and developers -

Dominik Brodowski <linux@brodo.de>
some additions and corrections by Nico Golde <nico@ngolde.de>
Rafael J. Wysocki <rafael.j.wysocki@intel.com>
Viresh Kumar <viresh.kumar@linaro.org>

Clock scaling allows you to change the clock speed of the CPUs on the fly. This is a nice method to save battery power, because the lower the clock speed, the less power the CPU consumes.

Contents:
----

1. What Is A CPUFreq Governor?

==============================

Most cpufreq drivers (except the intel_pstate and longrun) or even most cpu frequency scaling algorithms only allow the CPU frequency to be set to predefined fixed values. In order to offer dynamic frequency scaling, the cpufreq core must be able to tell these drivers of a "target frequency". So these specific drivers will be transformed to offer a "->target/target_index/fast_switch()" call instead of the

"->setpolicy()" call. For set_policy drivers, all stays the same, though.

How to decide what frequency within the CPUfreq policy should be used? That's done using "cpufreq governors".

Basically, it's the following flow graph:

```
CPU can be set to switch independently        |             CPU can only be set
      within specific "limits"                    |       to specific frequencies

                                 "CPUfreq policy"
                        consists of frequency limits (policy->{min,max})
                             and CPUfreq governor to be used
                                    /                          \
                                   /                            \
                              /                       the cpufreq governor decides
                             /                                  (dynamically or statically)
                            /                                   what target_freq to set within
                           /                                    the limits of policy->{min,max}
                          /                                          \
                         /                                                \
            Using the ->setpolicy call,                   Using the ->target/target_index/fast_switch
call,
                the limits and the                                the frequency closest
                 "policy" is set.                                   to target_freq is set.
                                                                        It is assured that it
                                                                        is within policy->{min,max}
```

2. Governors In the Linux Kernel
================================

2.1 Performance

The CPUfreq governor "performance" sets the CPU statically to the highest frequency within the borders of scaling_min_freq and scaling_max_freq.

2.2 Powersave

The CPUfreq governor "powersave" sets the CPU statically to the lowest frequency within the borders of scaling_min_freq and scaling_max_freq.

2.3 Userspace

The CPUfreq governor "userspace" allows the user, or any userspace program running with UID "root", to set the CPU to a specific frequency by making a sysfs file "scaling_setspeed" available in the CPU-device directory.

2.4 Ondemand

The CPUfreq governor "ondemand" sets the CPU frequency depending on the current system load. Load estimation is triggered by the scheduler through the update_util_data->func hook; when triggered, cpufreq checks the CPU-usage statistics over the last period and the governor sets the CPU accordingly. The CPU must have the capability to switch the frequency very quickly.

Sysfs files:

* sampling_rate:

Measured in uS (10^-6 seconds), this is how often you want the kernel to look at the CPU usage and to make decisions on what to do about the frequency. Typically this is set to values of around '10000' or more. It's default value is (cmp. with users-guide.txt): transition_latency * 1000. Be aware that transition latency is in ns and sampling_rate is in us, so you get the same sysfs value by default. Sampling rate should always get adjusted considering the transition latency to set the sampling rate 750 times as high as the transition latency in the bash (as said, 1000 is default), do:

```
$ echo `$(($(cat cpuinfo_transition_latency) * 750 / 1000)) > ondemand/sampling_rate
```

* sampling_rate_min:

The sampling rate is limited by the HW transition latency:
transition_latency * 100

Or by kernel restrictions:

```
- If CONFIG_NO_HZ_COMMON is set, the limit is 10ms fixed.
- If CONFIG_NO_HZ_COMMON is not set or nohz=off boot parameter is
  used, the limits depend on the CONFIG_HZ option:
  HZ=1000: min=20000us  (20ms)
  HZ=250:  min=80000us  (80ms)
  HZ=100:  min=200000us (200ms)
```

The highest value of kernel and HW latency restrictions is shown and used as the minimum sampling rate.

* up_threshold:

This defines what the average CPU usage between the samplings of 'sampling_rate' needs to be for the kernel to make a decision on whether it should increase the frequency. For example when it is set to its default value of '95' it means that between the checking intervals the CPU needs to be on average more than 95% in use to then decide that the CPU frequency needs to be increased.

* ignore_nice_load:

This parameter takes a value of '0' or '1'. When set to '0' (its default), all processes are counted towards the 'cpu utilisation' value. When set to '1', the processes that are run with a 'nice' value will not count (and thus be ignored) in the overall usage calculation. This is useful if you are running a CPU intensive calculation on your laptop that you do not care how long it takes to complete as you can 'nice' it and prevent it from taking part in the deciding process of whether to increase your CPU frequency.

* sampling_down_factor:

This parameter controls the rate at which the kernel makes a decision on when to decrease the frequency while running at top speed. When set to 1 (the default) decisions to reevaluate load are made at the same interval regardless of current clock speed. But when set to greater than 1 (e.g. 100) it acts as a multiplier for the scheduling interval for reevaluating load when the CPU is at its top speed due to high load. This improves performance by reducing the overhead of load evaluation and helping the CPU stay at its top speed when truly busy, rather than shifting back and forth in speed. This tunable has no effect on behavior at lower speeds/lower CPU loads.

* powersave_bias:

This parameter takes a value between 0 to 1000. It defines the percentage (times 10) value of the target frequency that will be shaved off of the target. For example, when set to 100 -- 10%, when ondemand governor would have targeted 1000 MHz, it will target 1000 MHz - (10% of 1000 MHz) = 900 MHz instead. This is set to 0 (disabled) by default.

When AMD frequency sensitivity powersave bias driver -- drivers/cpufreq/amd_freq_sensitivity.c is loaded, this parameter defines the workload frequency sensitivity threshold in which a lower frequency is chosen instead of ondemand governor's original target. The frequency sensitivity is a hardware reported (on AMD Family 16h Processors and above) value between 0 to 100% that tells software how the performance of the workload running on a CPU will change when frequency changes. A workload with sensitivity of 0% (memory/IO-bound) will not perform any better on higher core frequency, whereas a workload with sensitivity of 100% (CPU-bound) will perform better higher the frequency. When the driver is loaded, this is set to 400 by default -- for CPUs running workloads with sensitivity value below 40%, a lower frequency is chosen. Unloading the driver or writing 0 will disable this feature.

2.5 Conservative

The CPUfreq governor "conservative", much like the "ondemand" governor, sets the CPU frequency depending on the current usage. It differs in behaviour in that it gracefully increases and decreases the CPU speed rather than jumping to max speed the moment there is any load on the CPU. This behaviour is more suitable in a battery powered environment. The governor is tweaked in the same manner as the "ondemand" governor through sysfs with the addition of:

* freq_step:

This describes what percentage steps the cpu freq should be increased and decreased smoothly by. By default the cpu frequency will increase in 5% chunks of your maximum cpu frequency. You can change this value to anywhere between 0 and 100 where '0' will effectively lock your CPU at a speed regardless of its load whilst '100' will, in theory, make it behave identically to the "ondemand" governor.

* down_threshold:

Same as the 'up_threshold' found for the "ondemand" governor but for the opposite direction. For example when set to its default value of '20' it means that if the CPU usage needs to be below 20% between samples to have the frequency decreased.

* sampling_down_factor:

Similar functionality as in "ondemand" governor. But in "conservative", it controls the rate at which the kernel makes a decision on when to decrease the frequency while running in any speed. Load for frequency increase is still evaluated every sampling rate.

2.6 Schedutil

The "schedutil" governor aims at better integration with the Linux kernel scheduler. Load estimation is achieved through the scheduler's Per-Entity Load Tracking (PELT) mechanism, which also provides information about the recent load [1]. This governor currently does load based DVFS only for tasks managed by CFS. RT and DL scheduler tasks are always run at the highest frequency. Unlike all the other governors, the code is located under the kernel/sched/ directory.

Sysfs files:

* rate_limit_us:

  This contains a value in microseconds. The governor waits for rate_limit_us time before reevaluating the load again, after it has evaluated the load once.

For an in-depth comparison with the other governors refer to [2].

3. The Governor Interface in the CPUfreq Core

A new governor must register itself with the CPUfreq core using "cpufreq_register_governor". The struct cpufreq_governor, which has to be passed to that function, must contain the following values:

governor->name - A unique name for this governor.
governor->owner - .THIS_MODULE for the governor module (if appropriate).

plus a set of hooks to the functions implementing the governor's logic.

The CPUfreq governor may call the CPU processor driver using one of these two functions:

```
int cpufreq_driver_target(struct cpufreq_policy *policy,
                          unsigned int target_freq,
                          unsigned int relation);

int __cpufreq_driver_target(struct cpufreq_policy *policy,
                            unsigned int target_freq,
                            unsigned int relation);
```

target_freq must be within policy->min and policy->max, of course. What's the difference between these two functions? When your governor is

in a direct code path of a call to governor callbacks, like
governor->start(), the policy->rwsem is still held in the cpufreq core,
and there's no need to lock it again (in fact, this would cause a
deadlock). So use __cpufreq_driver_target only in these cases. In all
other cases (for example, when there's a "daemonized" function that
wakes up every second), use cpufreq_driver_target to take policy->rwsem
before the command is passed to the cpufreq driver.

4. References
=============

[1] Per-entity load tracking: https://lwn.net/Articles/531853/
[2] Improvements in CPU frequency management: https://lwn.net/Articles/682391/