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

L i n u x   C P U F r e q

C P U F r e q   G o v e r n o r s

- information for users and developers -

Dominik Brodowski <linux@brodo.de>
some additions and corrections by Nico Golde <nico@ngolde.de>

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 (in fact, all except one, longrun) or even most cpu frequency scaling algorithms only offer the CPU to be set to one frequency. 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" call instead of the existing "->setpolicy" call. For "longrun", all stays the same, though.

How to decide what frequency within the CPUfreq policy should be used? That's done using "cpufreq governors". Two are already in this patch -- they're the already existing "powersave" and "performance" which set the frequency statically to the lowest or highest frequency, respectively. At least two more such governors will be ready for addition in the near future, but likely many more as there are various different theories and models about dynamic frequency scaling around. Using such a generic interface as cpufreq offers to scaling governors, these can be tested extensively, and the best one can be selected for each specific use.

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 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 depending on the

current usage. To do this the CPU must have the capability to
switch the frequency very quickly. There are a number of sysfs file
accessible parameters:

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
```

show_sampling_rate_min:
The sampling rate is limited by the HW transition latency:
transition_latency * 100
Or by kernel restrictions:
If CONFIG_NO_HZ is set, the limit is 10ms fixed.
If CONFIG_NO_HZ is not set or no_hz=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.

show_sampling_rate_max: THIS INTERFACE IS DEPRECATED, DON'T USE IT.

up_threshold: 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.

2.5 Conservative

The CPUfreq governor "conservative", much like the "ondemand"
governor, sets the CPU 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 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.

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->governor -        The governor callback function
governor->owner -           .THIS_MODULE for the governor module (if
                            appropriate)
```

The governor->governor callback is called with the current (or to-be-set)
cpufreq_policy struct for that CPU, and an unsigned int event. The
following events are currently defined:

```
CPUFREQ_GOV_START:   This governor shall start its duty for the CPU
                     policy->cpu
CPUFREQ_GOV_STOP:    This governor shall end its duty for the CPU
                     policy->cpu
CPUFREQ_GOV_LIMITS:  The limits for CPU policy->cpu have changed to
                     policy->min and policy->max.
```

If you need other "events" externally of your driver, _only_ use the
cpufreq_governor_l(unsigned int cpu, unsigned int event) call to the
CPUfreq core to ensure proper locking.

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 still is in a direct code path of a call to governor->governor, the per-CPU cpufreq lock 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 lock the cpufreq per-CPU lock before the command is passed to the cpufreq processor driver.