CPU Performance Scaling

The Concept of CPU Performance Scaling

The majority of modern processors are capable of operating in a number of different clock frequency and voltage configurations, often referred to as Operating Performance Points or P-states (in ACPI terminology). As a rule, the higher the clock frequency and the higher the voltage, the more instructions can be retired by the CPU over a unit of time, but also the higher the clock frequency and the higher the voltage, the more energy is consumed over a unit of time (or the more power is drawn) by the CPU in the given P-state. Therefore there is a natural tradeoff between the CPU capacity (the number of instructions that can be executed over a unit of time) and the power drawn by the CPU.

In some situations it is desirable or even necessary to run the program as fast as possible and then there is no reason to use any P-states different from the highest one (i.e. the highest-performance frequency/voltage configuration available). In some other cases, however, it may not be necessary to execute instructions so quickly and maintaining the highest available CPU capacity for a relatively long time without utilizing it entirely may be regarded as wasteful. It also may not be physically possible to maintain maximum CPU capacity for too long for thermal or power supply capacity reasons or similar. To cover those cases, there are hardware interfaces allowing CPUs to be switched between different frequency/voltage configurations or (in the ACPI terminology) to be put into different P-states.

Typically, they are used along with algorithms to estimate the required CPU capacity, so as to decide which P-states to put the CPUs into. Of course, since the utilization of the system generally changes over time, that has to be done repeatedly on a regular basis. The activity by which this happens is referred to as CPU performance scaling or CPU frequency scaling (because it involves adjusting the CPU clock frequency).

CPU Performance Scaling in Linux

The Linux kernel supports CPU performance scaling by means of the `CPUFreq` (CPU Frequency scaling) subsystem that consists of three layers of code: the core, scaling governors and scaling drivers.

The `CPUFreq` core provides the common code infrastructure and user space interfaces for all platforms that support CPU performance scaling. It defines the basic framework in which the other components operate.

Scaling governors implement algorithms to estimate the required CPU capacity. As a rule, each governor implements one, possibly parametrized, scaling algorithm.

Scaling drivers talk to the hardware. They provide scaling governors with information on the available P-states (or P-state ranges in some cases) and access platform-specific hardware interfaces to change CPU P-states as requested by scaling governors.

In principle, all available scaling governors can be used with every scaling driver. That design is based on the observation that the information used by performance scaling algorithms for P-state selection can be represented in a platform-independent form in the majority of cases, so it should be possible to use the same performance scaling algorithm implemented in exactly the same way regardless of which scaling driver is used. Consequently, the same set of scaling governors should be suitable for every supported platform.

However, that observation may not hold for performance scaling algorithms based on information provided by the hardware itself, for example through feedback registers, as that information is typically specific to the hardware interface it comes from and may not be easily represented in an abstract, platform-independent way. For this reason, `CPUFreq` allows scaling drivers to bypass the governor layer and implement their own performance scaling algorithms. That is done by the *intel_pstate* scaling driver.

`CPUFreq` Policy Objects

In some cases the hardware interface for P-state control is shared by multiple CPUs. That is, for example, the same register (or set of registers) is used to control the P-state of multiple CPUs at the same time and writing to it affects all of those CPUs simultaneously.

Sets of CPUs sharing hardware P-state control interfaces are represented by `CPUFreq` as `struct cpufreq_policy` objects. For consistency, `struct cpufreq_policy` is also used when there is only one CPU in the given set.

The `CPUFreq` core maintains a pointer to a `struct cpufreq_policy` object for every CPU in the system, including CPUs that are currently offline. If multiple CPUs share the same hardware P-state control interface, all of the pointers corresponding to them point to the same `struct cpufreq_policy` object.

`CPUFreq` uses `struct cpufreq_policy` as its basic data type and the design of its user space interface is based on the policy concept.

CPU Initialization

First of all, a scaling driver has to be registered for `CPUFreq` to work. It is only possible to register one scaling driver at a time, so the scaling driver is expected to be able to handle all CPUs in the system.

The scaling driver may be registered before or after CPU registration. If CPUs are registered earlier, the driver core invokes the `CPUFreq` core to take a note of all of the already registered CPUs during the registration of the scaling driver. In turn, if any CPUs are registered after the registration of the scaling driver, the `CPUFreq` core will be invoked to take note of them at their registration time.

In any case, the `CPUFreq` core is invoked to take note of any logical CPU it has not seen so far as soon as it is ready to handle that CPU. [Note that the logical CPU may be a physical single-core processor, or a single core in a multicore processor, or a hardware thread in a physical processor or processor core. In what follows "CPU" always means "logical CPU" unless explicitly stated otherwise and the word "processor" is used to refer to the physical part possibly including multiple logical CPUs.]

Once invoked, the `CPUFreq` core checks if the policy pointer is already set for the given CPU and if so, it skips the policy object creation. Otherwise, a new policy object is created and initialized, which involves the creation of a new policy directory in `sysfs`, and the policy pointer corresponding to the given CPU is set to the new policy object's address in memory.

Next, the scaling driver's `->init()` callback is invoked with the policy pointer of the new CPU passed to it as the argument. That callback is expected to initialize the performance scaling hardware interface for the given CPU (or, more precisely, for the set of CPUs sharing the hardware interface it belongs to, represented by its policy object) and, if the policy object it has been called for is new, to set parameters of the policy, like the minimum and maximum frequencies supported by the hardware, the table of available frequencies (if the set of supported P-states is not a continuous range), and the mask of CPUs that belong to the same policy (including both online and offline CPUs). That mask is then used by the core to populate the policy pointers for all of the CPUs in it.

The next major initialization step for a new policy object is to attach a scaling governor to it (to begin with, that is the default scaling governor determined by the kernel configuration, but it may be changed later via `sysfs`). First, a pointer to the new policy object is passed to the governor's `->init()` callback which is expected to initialize all of the data structures necessary to handle the given policy and, possibly, to add a governor `sysfs` interface to it. Next, the governor is started by invoking its `->start()` callback.

That callback it expected to register per-CPU utilization update callbacks for all of the online CPUs belonging to the given policy with the CPU scheduler. The utilization update callbacks will be invoked by the CPU scheduler on important events, like task enqueue and dequeue, on every iteration of the scheduler tick or generally whenever the CPU utilization may change (from the scheduler's perspective). They are expected to carry out computations needed to determine the P-state to use for the given policy going forward and to invoke the scaling driver to make changes to the hardware in accordance with the P-state selection. The scaling driver may be invoked directly from scheduler context or asynchronously, via a kernel thread or workqueue, depending on the configuration and capabilities of the scaling driver and the governor.

Similar steps are taken for policy objects that are not new, but were “inactive” previously, meaning that all of the CPUs belonging to them were offline. The only practical difference in that case is that the `CPUFreq` core will attempt to use the scaling governor previously used with the policy that became “inactive” (and is re-initialized now) instead of the default governor.

In turn, if a previously offline CPU is being brought back online, but some other CPUs sharing the policy object with it are online already, there is no need to re-initialize the policy object at all. In that case, it only is necessary to restart the scaling governor so that it can take the new online CPU into account. That is achieved by invoking the governor’s `->stop` and `->start()` callbacks, in this order, for the entire policy.

As mentioned before, the *intel_pstate* scaling driver bypasses the scaling governor layer of `CPUFreq` and provides its own P-state selection algorithms. Consequently, if *intel_pstate* is used, scaling governors are not attached to new policy objects. Instead, the driver’s `->setpolicy()` callback is invoked to register per-CPU utilization update callbacks for each policy. These callbacks are invoked by the CPU scheduler in the same way as for scaling governors, but in the *intel_pstate* case they both determine the P-state to use and change the hardware configuration accordingly in one go from scheduler context.

The policy objects created during CPU initialization and other data structures associated with them are torn down when the scaling driver is unregistered (which happens when the kernel module containing it is unloaded, for example) or when the last CPU belonging to the given policy in unregistered.

Policy Interface in `sysfs`

During the initialization of the kernel, the `CPUFreq` core creates a `sysfs` directory (kobject) called `cpufreq` under `/sys/devices/system/cpu/`.

That directory contains a `policyX` subdirectory (where `X` represents an integer number) for every policy object maintained by the `CPUFreq` core. Each `policyX` directory is pointed to by `cpufreq` symbolic links under `/sys/devices/system/cpu/cpuY/` (where `Y` represents an integer that may be different from the one represented by `X`) for all of the CPUs associated with (or belonging to) the given policy. The `policyX` directories in `/sys/devices/system/cpu/cpufreq` each contain policy-specific attributes (files) to control `CPUFreq` behavior for the corresponding policy objects (that is, for all of the CPUs associated with them).

Some of those attributes are generic. They are created by the `CPUFreq` core and their behavior generally does not depend on what scaling driver is in use and what scaling governor is attached to the given policy. Some scaling drivers also add driver-specific attributes to the policy directories in `sysfs` to control policy-specific aspects of driver behavior.

The generic attributes under `/sys/devices/system/cpu/cpufreq/policyX/` are the following:

`affected_cpus`
: List of online CPUs belonging to this policy (i.e. sharing the hardware performance scaling interface represented by the `policyX` policy object).

`bios_limit`
: If the platform firmware (BIOS) tells the OS to apply an upper limit to CPU frequencies, that limit will be reported through this attribute (if present).

: The existence of the limit may be a result of some (often unintentional) BIOS settings, restrictions coming from a service processor or another BIOS/HW-based mechanisms.

: This does not cover ACPI thermal limitations which can be discovered through a generic thermal driver.

: This attribute is not present if the scaling driver in use does not support it.

`cpuinfo_max_freq`
: Maximum possible operating frequency the CPUs belonging to this policy can run at (in kHz).

`cpuinfo_min_freq`
: Minimum possible operating frequency the CPUs belonging to this policy can run at (in kHz).

`cpuinfo_transition_latency`
: The time it takes to switch the CPUs belonging to this policy from one P-state to another, in nanoseconds.

  If unknown or if known to be so high that the scaling driver does not work with the ondemand governor, -1 (`CPUFREQ_ETERNAL`) will be returned by reads from this attribute.

`related_cpus`
: List of all (online and offline) CPUs belonging to this policy.

`scaling_available_governors`
: List of `CPUFreq` scaling governors present in the kernel that can be attached to this policy or (if the *intel_pstate* scaling driver is in use) list of scaling algorithms provided by the driver that can be applied to this policy.

  [Note that some governors are modular and it may be necessary to load a kernel module for the governor held by it to become available and be listed by this attribute.]

`scaling_cur_freq`
: Current frequency of all of the CPUs belonging to this policy (in kHz).

  For the majority of scaling drivers, this is the frequency of the last P-state requested by the driver from the hardware using the scaling interface provided by it, which may or may not reflect the frequency the CPU is actually running at (due to hardware design and other limitations).

  Some scaling drivers (e.g. *intel_pstate*) attempt to provide information more precisely reflecting the current CPU frequency through this attribute, but that still may not be the exact current CPU frequency as seen by the hardware at the moment.

`scaling_driver`
: The scaling driver currently in use.

`scaling_governor`
: The scaling governor currently attached to this policy or (if the *intel_pstate* scaling driver is in use) the scaling algorithm provided by the driver that is currently applied to this policy.

  This attribute is read-write and writing to it will cause a new scaling governor to be attached to this policy or a new scaling algorithm provided by the scaling driver to be applied to it (in the *intel_pstate* case), as indicated by the string written to this attribute (which must be one of the names listed by the `scaling_available_governors` attribute described above).

`scaling_max_freq`
: Maximum frequency the CPUs belonging to this policy are allowed to be running at (in kHz).

  This attribute is read-write and writing a string representing an integer to it will cause a new limit to be set (it must not be lower than the value of the `scaling_min_freq` attribute).

`scaling_min_freq`
: Minimum frequency the CPUs belonging to this policy are allowed to be running at (in kHz).

  This attribute is read-write and writing a string representing a non-negative integer to it will cause a new limit to be set (it must not be higher than the value of the `scaling_max_freq` attribute).

`scaling_setspeed`
: This attribute is functional only if the userspace scaling governor is attached to the given policy.

  It returns the last frequency requested by the governor (in kHz) or can be written to in order to set a new frequency for the policy.

Generic Scaling Governors

`CPUFreq` provides generic scaling governors that can be used with all scaling drivers. As stated before, each of them implements a single, possibly parametrized, performance scaling algorithm.

Scaling governors are attached to policy objects and different policy objects can be handled by different scaling governors at the same time (although that may lead to suboptimal results in some cases).

The scaling governor for a given policy object can be changed at any time with the help of the `scaling_governor` policy attribute in `sysfs`.

Some governors expose `sysfs` attributes to control or fine-tune the scaling algorithms implemented by them. Those attributes, referred to as governor tunables, can be either global (system-wide) or per-policy, depending on the scaling driver in use. If the driver requires governor tunables to be per-policy, they are located in a subdirectory of each policy directory. Otherwise, they are located in a subdirectory under `/sys/devices/system/cpu/cpufreq/`. In either case the name of the subdirectory containing the governor tunables is the name of the governor providing them.

`performance`

When attached to a policy object, this governor causes the highest frequency, within the `scaling_max_freq` policy limit, to be requested for that policy.

The request is made once at that time the governor for the policy is set to `performance` and whenever the `scaling_max_freq` or `scaling_min_freq` policy limits change after that.

`powersave`

When attached to a policy object, this governor causes the lowest frequency, within the `scaling_min_freq` policy limit, to be requested for that policy.

The request is made once at that time the governor for the policy is set to `powersave` and whenever the `scaling_max_freq` or `scaling_min_freq` policy limits change after that.

`userspace`

This governor does not do anything by itself. Instead, it allows user space to set the CPU frequency for the policy it is attached to by writing to the `scaling_setspeed` attribute of that policy.

`schedutil`

This governor uses CPU utilization data available from the CPU scheduler. It generally is regarded as a part of the CPU scheduler, so it can access the scheduler's internal data structures directly.

It runs entirely in scheduler context, although in some cases it may need to invoke the scaling driver asynchronously when it decides that the CPU frequency should be changed for a given policy (that depends on whether or not the driver is capable of changing the CPU frequency from scheduler context).

The actions of this governor for a particular CPU depend on the scheduling class invoking its utilization update callback for that CPU. If it is invoked by the RT or deadline scheduling classes, the governor will increase the frequency to the allowed maximum (that is, the `scaling_max_freq` policy limit). In turn, if it is invoked by the CFS scheduling class, the governor will use the Per-Entity Load Tracking (PELT) metric for the root control group of the given CPU as the CPU utilization estimate (see the Per-entity load tracking LWN.net article for a description of the PELT mechanism). Then, the new CPU frequency to apply is computed in accordance with the formula

> f = 1.25 * `f_0` * `util` / `max`

where `util` is the PELT number, `max` is the theoretical maximum of `util`, and `f_0` is either the maximum possible CPU frequency for the given policy (if the PELT number is frequency-invariant), or the current CPU frequency (otherwise).

This governor also employs a mechanism allowing it to temporarily bump up the CPU frequency for tasks that have been waiting on I/O most recently, called "IO-wait boosting". That happens when the ***`SCHED_CPUFREQ_IOWAIT`*** flag is passed by the scheduler to the governor callback which causes the frequency to go up to the allowed maximum immediately and then draw back to the value returned by the above formula over time.

This governor exposes only one tunable:

`rate_limit_us`

> Minimum time (in microseconds) that has to pass between two consecutive runs of governor computations (default: 1000 times the scaling driver's transition latency).
>
> The purpose of this tunable is to reduce the scheduler context overhead of the governor which might be excessive without it.

This governor generally is regarded as a replacement for the older ondemand and conservative governors (described below), as it is simpler and more tightly integrated with the CPU scheduler, its overhead in terms of CPU context switches and similar is less significant, and it uses the scheduler's own CPU utilization metric, so in principle its decisions should not contradict the decisions made by the other parts of the scheduler.

`ondemand`

This governor uses CPU load as a CPU frequency selection metric.

In order to estimate the current CPU load, it measures the time elapsed between consecutive invocations of its worker routine and computes the fraction of that time in which the given CPU was not idle. The ratio of the non-idle (active) time to the total CPU time is taken as an estimate of the load.

If this governor is attached to a policy shared by multiple CPUs, the load is estimated for all of them and the greatest result is taken as the load estimate for the entire policy.

The worker routine of this governor has to run in process context, so it is invoked asynchronously (via a workqueue) and CPU P-states are updated from there if necessary. As a result, the scheduler context overhead from this governor is minimum, but it causes additional CPU context switches to happen relatively often and the CPU P-state updates triggered by it can be relatively irregular. Also, it affects its own CPU load metric by running code that reduces the CPU idle time (even though the CPU idle time is only reduced very slightly by it).

It generally selects CPU frequencies proportional to the estimated load, so that the value of the `cpuinfo_max_freq` policy attribute corresponds to the load of 1 (or 100%), and the value of the `cpuinfo_min_freq` policy attribute corresponds to the load of 0, unless when the load exceeds a (configurable) speedup threshold, in which case it will go straight for the highest frequency it is allowed to use (the `scaling_max_freq` policy limit).

This governor exposes the following tunables:

`sampling_rate`
: This is how often the governor's worker routine should run, in microseconds.

  Typically, it is set to values of the order of 10000 (10 ms). Its default value is equal to the value of `cpuinfo_transition_latency` for each policy this governor is attached to (but since the unit here is greater by 1000, this means that the time represented by `sampling_rate` is 1000 times greater than the transition latency by default).

  If this tunable is per-policy, the following shell command sets the time represented by it to be 750 times as high as the transition latency:

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

`min_sampling_rate`
: The minimum value of `sampling_rate`.

  Equal to 10000 (10 ms) if ***`CONFIG_NO_HZ_COMMON`*** and ***`tick_nohz_active`*** are both set or to 20 times the value of ***`jiffies`*** in microseconds otherwise.

`up_threshold`
: If the estimated CPU load is above this value (in percent), the governor will set the frequency to the maximum value allowed for the policy. Otherwise, the selected frequency will be proportional to the estimated CPU load.

`ignore_nice_load`
: If set to 1 (default 0), it will cause the CPU load estimation code to treat the CPU time spent on executing tasks with "nice" levels greater than 0 as CPU idle time.

  This may be useful if there are tasks in the system that should not be taken into account when deciding what frequency to run the CPUs at. Then, to make that happen it is sufficient to increase the "nice" level of those tasks above 0 and set this attribute to 1.

`sampling_down_factor`
: Temporary multiplier, between 1 (default) and 100 inclusive, to apply to the `sampling_rate` value if the CPU load goes above `up_threshold`.

  This causes the next execution of the governor's worker routine (after setting the frequency to the allowed maximum) to be delayed, so the frequency stays at the maximum level for a longer time.

  Frequency fluctuations in some bursty workloads may be avoided this way at the cost of additional energy spent on maintaining the maximum CPU capacity.

`powersave_bias`

Reduction factor to apply to the original frequency target of the governor (including the maximum value used when the `up_threshold` value is exceeded by the estimated CPU load) or sensitivity threshold for the AMD frequency sensitivity powersave bias driver (`drivers/cpufreq/amd_freq_sensitivity.c`), between 0 and 1000 inclusive.

If the AMD frequency sensitivity powersave bias driver is not loaded, the effective frequency to apply is given by

f * (1 - `powersave_bias` / 1000)

where f is the governor's original frequency target. The default value of this attribute is 0 in that case.

If the AMD frequency sensitivity powersave bias driver is loaded, the value of this attribute is 400 by default and it is used in a different way.

On Family 16h (and later) AMD processors there is a mechanism to get a measured workload sensitivity, between 0 and 100% inclusive, from the hardware. That value can be used to estimate how the performance of the workload running on a CPU will change in response to frequency changes.

The performance of a workload with the sensitivity of 0 (memory-bound or IO-bound) is not expected to increase at all as a result of increasing the CPU frequency, whereas workloads with the sensitivity of 100% (CPU-bound) are expected to perform much better if the CPU frequency is increased.

If the workload sensitivity is less than the threshold represented by the `powersave_bias` value, the sensitivity powersave bias driver will cause the governor to select a frequency lower than its original target, so as to avoid over-provisioning workloads that will not benefit from running at higher CPU frequencies.

`conservative`

This governor uses CPU load as a CPU frequency selection metric.

It estimates the CPU load in the same way as the ondemand governor described above, but the CPU frequency selection algorithm implemented by it is different.

Namely, it avoids changing the frequency significantly over short time intervals which may not be suitable for systems with limited power supply capacity (e.g. battery-powered). To achieve that, it changes the frequency in relatively small steps, one step at a time, up or down - depending on whether or not a (configurable) threshold has been exceeded by the estimated CPU load.

This governor exposes the following tunables:

`freq_step`

Frequency step in percent of the maximum frequency the governor is allowed to set (the `scaling_max_freq` policy limit), between 0 and 100 (5 by default).

This is how much the frequency is allowed to change in one go. Setting it to 0 will cause the default frequency step (5 percent) to be used and setting it to 100 effectively causes the governor to periodically switch the frequency between the `scaling_min_freq` and `scaling_max_freq` policy limits.

`down_threshold`

Threshold value (in percent, 20 by default) used to determine the frequency change direction.

If the estimated CPU load is greater than this value, the frequency will go up (by `freq_step`). If the load is less than this value (and the `sampling_down_factor` mechanism is not in effect), the frequency will go down. Otherwise, the frequency will not be changed.

`sampling_down_factor`

Frequency decrease deferral factor, between 1 (default) and 10 inclusive.

It effectively causes the frequency to go down `sampling_down_factor` times slower than it ramps up.

Frequency Boost Support

Background

Some processors support a mechanism to raise the operating frequency of some cores in a multicore package temporarily (and above the sustainable frequency threshold for the whole package) under certain conditions, for example if the whole chip is not fully utilized and below its intended thermal or power budget.

Different names are used by different vendors to refer to this functionality. For Intel processors it is referred to as "Turbo Boost", AMD calls it "Turbo-Core" or (in technical documentation) "Core Performance Boost" and so on. As a rule, it also is implemented differently by different vendors. The simple term "frequency boost" is used here for brevity to refer to all of those implementations.

The frequency boost mechanism may be either hardware-based or software-based. If it is hardware-based (e.g. on x86), the decision to trigger the boosting is made by the hardware (although in general it requires the hardware to be put into a special state in which it can control the CPU frequency within certain limits). If it is software-based (e.g. on ARM), the scaling driver decides whether or not to trigger boosting and when to do that.

The `boost` File in `sysfs`

This file is located under `/sys/devices/system/cpu/cpufreq/` and controls the "boost" setting for the whole system. It is not present if the underlying scaling driver does not support the frequency boost mechanism (or supports it, but provides a driver-specific interface for controlling it, like *intel_pstate*).

If the value in this file is 1, the frequency boost mechanism is enabled. This means that either the hardware can be put into states in which it is able to trigger boosting (in the hardware-based case), or the software is allowed to trigger boosting (in the software-based case). It does not mean that boosting is actually in use at the moment on any CPUs in the system. It only means a permission to use the frequency boost mechanism (which still may never be used for other reasons).

If the value in this file is 0, the frequency boost mechanism is disabled and cannot be used at all.

The only values that can be written to this file are 0 and 1.

Rationale for Boost Control Knob

The frequency boost mechanism is generally intended to help to achieve optimum CPU performance on time scales below software resolution (e.g. below the scheduler tick interval) and it is demonstrably suitable for many workloads, but it may lead to problems in certain situations.

For this reason, many systems make it possible to disable the frequency boost mechanism in the platform firmware (BIOS) setup, but that requires the system to be restarted for the setting to be adjusted as desired, which may not be practical at least in some cases. For example:

1. Boosting means overclocking the processor, although under controlled conditions. Generally, the processor's energy consumption increases as a result of increasing its frequency and voltage, even temporarily. That may not be desirable on systems that switch to power sources of limited capacity, such as batteries, so the ability to disable the boost mechanism while the system is running may help there (but that depends on the workload too).
2. In some situations deterministic behavior is more important than performance or energy consumption (or both) and the ability to disable boosting while the system is running may be useful then.
3. To examine the impact of the frequency boost mechanism itself, it is useful to be able to run tests with and without boosting, preferably without restarting the system in the meantime.
4. Reproducible results are important when running benchmarks. Since the boosting functionality depends on the load of the whole package, single-thread performance may vary because of it which may lead to unreproducible results sometimes. That can be avoided by disabling the frequency boost mechanism before running benchmarks sensitive to that issue.

Legacy AMD `cpb` Knob

The AMD powernow-k8 scaling driver supports a `sysfs` knob very similar to the global `boost` one. It is used for disabling/enabling the "Core Performance Boost" feature of some AMD processors.

If present, that knob is located in every `CPUFreq` policy directory in `sysfs` (`/sys/devices/system/cpu/cpufreq/policyX/`) and is called `cpb`, which indicates a more fine grained control interface. The actual implementation, however, works on the system-wide basis and setting that knob for one policy causes the same value of it to be set for all of the other policies at the same time.

That knob is still supported on AMD processors that support its underlying hardware feature, but it may be configured out of the kernel (via the ***`CONFIG_X86_ACPI_CPUFREQ_CPB`*** configuration option) and the global `boost` knob is present regardless. Thus it is always possible use the `boost` knob instead of the `cpb` one which is highly recommended, as that is more consistent with what all of the other systems do (and the `cpb` knob may not be supported any more in the future).

The `cpb` knob is never present for any processors without the underlying hardware feature (e.g. all Intel ones), even if the ***`CONFIG_X86_ACPI_CPUFREQ_CPB`*** configuration option is set.