Power Management on Embedded Linux Systems

by Daniel Thompson | RISCstar Principal Engineer

Executive Summary

Power management is a fast-changing topic that has become increasingly relevant to embedded Linux systems. Modern battery technology means more and more systems can be freed from the power plug. That brings new requirements for managing power, since reducing the power consumed by the system allows devices to last longer between charges (or allows you to build it with smaller, lighter, and less expensive batteries).

Even when your embedded system is hooked to the grid, environmental certification programs often make energy efficiency a key part of purchasing decisions.

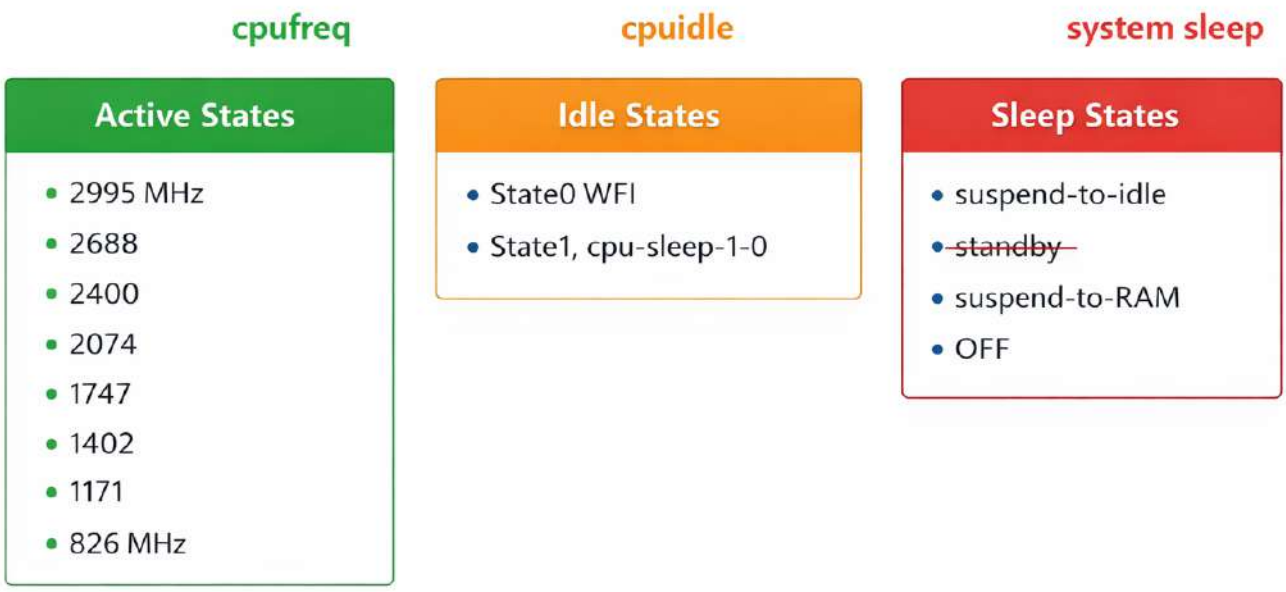
This paper looks at the most recent changes in cpufreq and cpuidle, and how to get the very best out of the hardware and software that drives your Arm or RISC-V-based embedded system.

Contents

The CPU Power State Spectrum

Modern devices provide many different power levels to support different modes of operation. These range from the highest performance state which, sadly, also has the greatest power consumption right down the lowest power state (off) which does nothing and consumes no power.

We can present these different power levels (power states) as a spectrum from highest to lowest power consumption. Here is a summary of the power states that are implemented on my own laptop:



In truth the diagram is simplified in a couple of ways. Firstly my laptop actually has 21 active states and drawing all of these would make the diagram rather cumbersome. To solve this I preserved the fastest and slowest operating frequencies but, for our mutual convenience, I have filtered out many of the states in the middle. Secondly, although my laptop is based around a pair of Arm big.LITTLE CPU clusters, I have included only one cluster in the above. This reflects the fact that heterogeneous CPU architectures, such as big.LITTLE, are less common in embedded systems than they are in laptops and mobile phones.

Take note that the diagram is ordered by power consumption but does not include a scale; the position doesn't tell us much about the actual power consumed in each state. We will come back to that later when we talk about DVFS.

Power State Definitions

The diagram separates the power states into three distinct categories: active, idle, and sleep. Each of these categories is tagged above with the Linux kernel subsystem that applies in each state. Before diving deeper let's take a quick look at what each power state means.

Active States

These represent periods where there is work for the CPU to do. Put in slightly more technical jargon, an active state means there are runnable tasks queued on the kernel scheduler. Some programs spend a lot of time waiting for input from the user or from slow peripherals such as a disk drive. Tasks that are waiting for input are blocked and are not runnable. For example, if a browser is showing a web page (and not playing music and adverts in the background), then there's not much to do and none of its tasks are runnable. However, as soon as the user scrolls the page, the browser will become active again (some of its tasks will be made runnable) so that it can update what's displayed.

When running in an active state, we can influence the power efficiency by changing the clock frequency of the CPU. The impact this can have on battery life is profound. My laptop provides 21 operating frequencies to allow for finely tuned decisions. While higher clock speeds deliver increased performance, they are less efficient and will drain the battery much faster than doing the same calculations at a slower operating frequency. We must therefore trade off how quickly we can deliver results to the user against how much of their battery we have to use to meet their expectations.

Idle States

When the kernel scheduler has no runnable tasks for the CPU it will transition into a low-power idle state. Idle states conserve energy but allow a rapid return to an active state as soon as an interrupt or wake-up event gives the CPU work to do. Most systems implement multiple idle states, and although there can be any number of them, two is common and there are seldom more than four.

Multiple idle states allow for different depths-of-idle. Shallow idle states consume more energy than deeper idle states. Multiple idle states exist because there is a time and energy cost to entering and exiting idle states. Shallow idle states are quicker to enter and need less energy to enter or exit. If we enter a deep idle state and the CPU is immediately woken up, we will probably spend more energy entering and exiting that state than we would have saved during the idle time. Having multiple idle states allows the kernel to estimate how long it will sleep for and to select an appropriate depth-of-idle to avoid wasting energy this way.

Sleep States

Sleep states differ from idle states because they force the kernel to stop scheduling work, even if there are runnable tasks ready to do. Whilst in the sleep state the system will ignore all input except for a very narrowly chosen set of wake-up events, such as pressing a button or opening the device's lid.

Sleep states are very intuitive for laptops. Even if your laptop is busy compiling the latest and greatest version of the kernel, when you shut the lid, you expect it to stop what it is doing so that it doesn't get too hot when you shove it in your backpack.

Sleep states do not exist in all systems or, in some cases, they do exist but are not very useful. Mobile phones and many embedded systems are designed to reside in a very deep idle state even when the user perceives them to be "off." For example, although mobile phones can be turned fully off, they are not useful in this state and many users don't know how to turn the device fully off. Devices like this are designed with very deep idle states together with extensive driver power management and careful sandboxing of applications. That allows them to spend a relatively long time in the deepest idle state, eliminating the need for sleep states.

We will not talk much more about the sleep states in this series since we're going to focus on the states related to the cpufreq and cpuidle subsystems in the Linux kernel.

Dynamic Voltage and Frequency Scaling (DVFS)

What is Dynamic Voltage and Frequency Scaling?

DVFS is the tool we have to manage power consumption when we are operating in the active states.

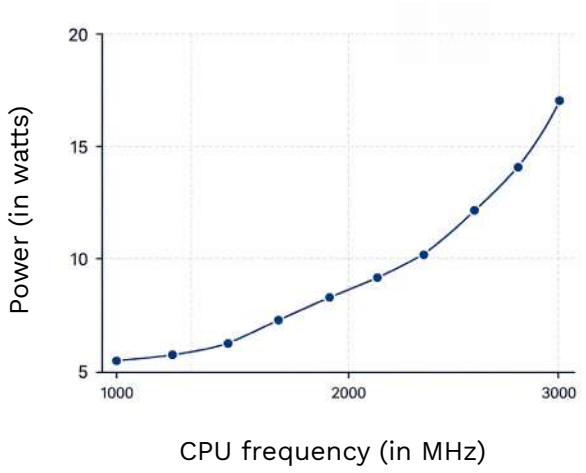
DVFS is a form of performance scaling that allows us to select between high-performance/high-power and lower-performance with more efficient power states. As the name suggests, it is not enough to simply select the frequency of a CPU. When we change frequency, we are also forced to dynamically scale the voltage to ensure the CPU can run correctly at the selected frequency. At their highest frequencies, CPUs need to run at high voltages to make sure all those electrons get to where they are needed before the next clock edge. At lower frequencies, however, we can use a lower voltage, thereby reducing power consumption.

Operating Performance Points (OPPs) Explained

DVFS systems typically offer a variety of "operating points" (sometimes called Operating Performance Points or OPPs), meaning the combination of supply voltage and clock frequency that work together safely and efficiently.

There are equations you could look up that approximately model how power is affected by changing the voltage and frequency. However, we can also simply look at a graph of real-world data. Once again, I have gathered information from my laptop, which shows the power draw from the battery (vertical axis, in watts) versus the clock frequency (horizontal axis, in MHz).

The power consumption includes all load on the battery. In addition to the CPU load, there are plenty of fixed loads such as the LCD backlight and DRAM refresh. This gives us an unusually high base load of around four or five watts, plus whatever the CPUs are consuming.



Based on the above graph, we can estimate that the battery will last three times longer under light loads, such as media playback, compared to heavy compute-intensive loads. The effect of compute-intensive activities can be even more pronounced on embedded devices, since they typically have much lower base loads.

DVFS and Embedded System Power Efficiency

For battery-powered systems, power-efficient use of the CPU means getting it to do the most calculations possible before the battery runs out. For equipment plugged in, it means lower electricity usage. The chosen OPP has a profound effect on power efficiency. As you can see on the graph, power consumption has a non-linear relationship with frequency, mostly due to the effects of voltage scaling. Doubling the frequency will more than double the power usage of the CPU. In contrast, performance will scale more or less linearly with frequency. Knowing this can help you read the graph and reason about embedded system power efficiency.

One especially interesting comparison is the difference between running the CPU at 80 percent of its capacity (2.4GHz) versus 100 percent of its capacity (~3GHz). The power consumed rockets from 10 watts to 17 watts. That means my 50-Watt-Hour battery will survive five hours at 80 percent but less than three hours at 100 percent. If you play with the math, that translates to being able to do 33 percent more calculations (or compile 33 percent more code) before the battery dies when operating at 80 percent capacity. Of course, we have to accept it will take longer to give us the answer in exchange for that.

DVFS allows us to make a choice. Should we minimize the time until we complete our calculations, or should we minimize the energy cost of doing the calculations to allow the battery to last longer? DVFS doesn't help us make the choice, but it does give us the flexibility to choose the best operating point for the circumstances.

Master DVFS for Maximum Battery Life

Understanding Dynamic Voltage and Frequency Scaling is crucial for optimizing your embedded system's power efficiency. The right implementation can extend battery life by 3x or more while maintaining performance where it matters.

Idle State Management

Core and Cluster Idling

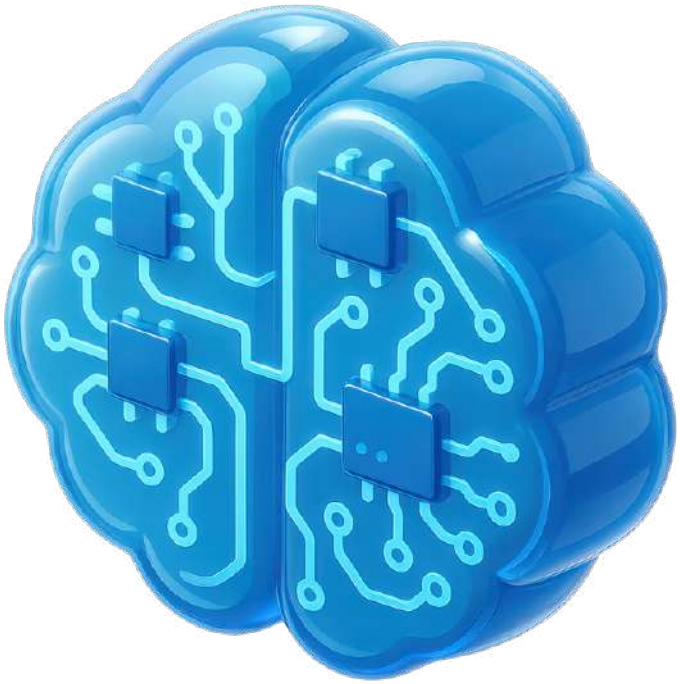
Idle states allow parts of the CPU to be de-clocked and/or powered off to minimize the power they consume. These features, selectively disabling clock or power to parts of the circuit, are sometimes referred to as clock- or power-gating.

The most shallow idle state on both Arm and RISC-V architecture can be entered by executing a wait-for-interrupt (WFI) instruction. This causes the CPU to enter a low-power state until the next interrupt. This idle state is usually "free" to enter. It does not conserve as much power as deeper idle states but is nearly instantaneous to enter and exit, meaning there are essentially no circumstances (except hardware bugs) where it should not be one of the idle states. WFI only affects the current core, so if the core shares resources (such as an L2 cache) with other cores then the shared resources may remain active even when all cores are in WFI loops.

A deeper idle state that exists on systems where the cores are combined into clusters is the cluster sleep. Cluster sleeps allow shared resources within the cluster to also enter low power states. There is usually a greater cost to enter the cluster sleep state; often it involves sending a request to a dedicated micro-controller, which has to orchestrate the sleep/wake-up sequence and requires time to perform this task. Cluster sleep is an example of a "voted" sleep, in that it requires all cores within the cluster to vote to enter the cluster sleep, otherwise the cores must remain in a shallower idle-state until they all vote the same way.

Some SoCs implement additional idle states beyond the two offered by my laptop. This was particularly common in older CPU designs where clock-gating could happen independently of power-gating. For example, if a CPU is implemented such that the WFI instruction only causes clock-gating, then some system designers introduced an additional idle state between WFI and cluster-idle. The intermediate state allowed a specific core to be power-gated without voting for a cluster sleep. This approach is less common today because most advanced CPU designs implement power-gating internally as part of the WFI implementation.

The largest number of idle states I ever recall seeing was four although that was long enough ago I can't remember what the hardware was. Based on this logic, the laptop I diagrammed above must be very advanced because it only implements two idle states!



Dynamic CPU Power Management with cpufreq and schedutil

cpufreq generally adopts the mantra “work when there is work to be done”. In other words, when there is a “demand” on the system then the kernel will do its best to give these tasks the CPU time they require to perform the task. However whenever the demand is less than 100% of the maximum CPU capacity cpufreq will attempt to opportunistically save energy by slowing down some or all of the cores.

To simplify its design cpufreq combines hardware specific cpufreq drivers with generic cpufreq governors. The split allows mechanism and policy to be separated:

Mechanism

cpufreq drivers provide the mechanism to switch between different DVFS operating points and are typically partially- or fully-customized to a specific system-on-chip (SoC). They are also responsible for describing the available operating points allowing the governor to make decisions.

Different drivers invoke DVFS changes in different ways. Some directly manipulate the registers of power control hardware whilst others work by sending messages to co-processors or supervisor mode firmware requesting a change of power state. The thing typical cpufreq drivers (outside the x86 eco-system, see below) have in common is that they do what they are told and are not responsible for making decisions about what operating point to adopt.

Policy

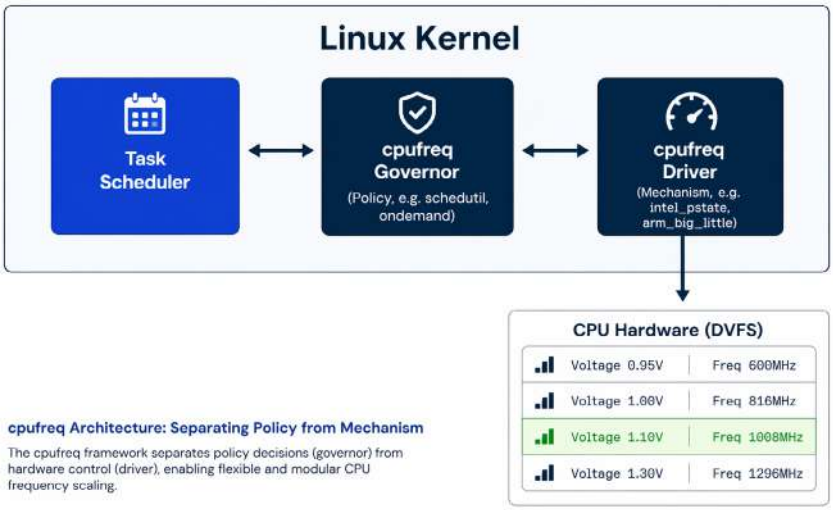
cpufreq governors provide the policy and are responsible for choosing the correct operating point from a list provided by the cpufreq driver.

Some governors are very simple. For example the performance and powersave governors will, respectively, run the CPU as fast or slow as possible. However the more flexible governors, such as ondemand, conservative, and schedutil seek to estimate the current CPU load and will adjust the CPU frequency to match. For example, during media playback, the CPU wakes up to decode the audio and video but then goes to sleep until the video hardware is ready to accept the next frame. This may require only 50% of CPU capacity so the ideal governor would select an operating point that is approximately half of the maximum frequency.

schedutil stands apart for its more complex governor and uses data captured by the scheduler to make decisions.

On Intel systems it is common for a cpufreq governor to be integrated into the cpufreq driver. On modern Intel systems the selection of power state is usually made by hardware. Similar designs can also exist on Arm and RISC-V platforms but they are unusual. Hardware managed designs benefit from low-latency feedback on current processor demand that is provided by special purpose monitoring hardware. Having said that hardware must rely on low-latency feedback; it does not benefit from the insight into the future provided by the scheduler (and utilized by schedutil to improve decision making).

How does the schedutil governor decide what frequency to run at?



cpufreq Architecture: Separating Policy from Mechanism

The cpufreq framework separates policy decisions (governor) from hardware control (driver), enabling flexible and modular CPU frequency scaling.

Historically cpufreq governors used relatively crude observations of the system and then applied heuristics to determine what frequency to run at.

For example, both the ondemand and conservative governors both work by observing the idle time on each CPU since they were last invoked. This allows them to speed up busy CPUs and slow down quiet ones. Both offer a variety of controls to tune their heuristics to get the desired result. These include things like how frequently they trigger and what thresholds should be met before enacting a change. There are two well-known issues with this approach:

1. It is difficult for them to handle tasks with bursty CPU demand. They simply assume that what happened since they were last invoked will continue to happen. This does not work well on systems with frequently changing loads.

2. They are difficult to tune because there is an indirect relationship between the desired behaviour and the tuneables available. Seesaw tuning is especially common, where one end goes up (one use-case improves) but the other end goes down (a different use-case regresses).

This situation changed with the introduction of the schedutil governor. As the name hints, this governor uses the scheduler's CPU utilization measurements to make decisions. These measurements are gathered on a per-task basis and provide a better basis to predict future load than the simplistic aggregate view of utilization available to the ondemand and conservative governors.

The scheduler knows which task is assigned to which CPU and maintains a CPU utilization estimate. This allows schedutil to sum the utilization of all scheduled tasks to give an estimated utilization for the CPU. From there a simple linear formula is used to select the operating point. In the presence of varying loads, the per-task estimated loads are typically more stable than the simplistic whole system view which cannot model bursts of activity effectively.

For example, let's think about a system that is:

Playing a movie

This system will be partially loaded with lots of idle time and will maintain a similar level of CPU usage for the whole movie. The load estimation for each task involved in playback will reflect the fact that these tasks do not require all the available CPU.

Browsing the web

When the user reads the page the browser will typically sleep with a CPU load close to zero waiting for user input. When the user clicks on something it will wake up and need lots of resources to run Javascript before it goes back to sleep. The load estimation for the browser tasks will reflect their bursty nature (e.g. when runnable they need lots of CPU).

schedutil not only copes gracefully with both the above cases individually but it is also able to handle both concurrently!

There are, of course, many different ways to estimate load. We can't go into every detail here except to note that the quality of these estimates is clearly important and that there have been several different approaches tried, both in the kernel tree and out-of-tree over the years, all of which lead to the current design.

Now that we understand how schedutil makes decisions, let's look at how to tune these systems to reduce power consumption in real-world applications.

Tuning cpufreq and schedutil for Lower Power Consumption

One of the key innovations offered by schedutil was the removal of heuristics. Heuristics typically take shortcuts and do not completely model the problem to be solved. Tuneables often go hand-in-hand with heuristics as a way to bias a heuristic to better match a particular use-case. schedutil replaces heuristics with a full model of all runnable tasks within the system together with the estimated utilization of each task. As a result schedutil differs from some of its predecessors and the governor itself doesn't provide any tuneables to bias its decisions.

None of this means that schedutil cannot be tuned. Instead there are two combinable strategies can be adopted:

1. Limit the choices schedutil can make by making certain operating points unavailable (this approach also works with other governors)

2. Use knowledge of the system to interfere with the estimated utilization.

Cull inefficient operating points

It is possible for the clock hardware in some designs to offer operating points that are not useful because a faster OPP is more efficient. In some cases it is better to run at a faster speed, complete the calculations quicker and then enter an idle state that causes power-gating.

In most Arm and RISC-V systems the table of operating points is found in the devicetree along with information to estimate the energy cost of each state. Normally the SoC vendor will have culled inefficient operating points for you but if you have an early release (or you work for a SoC vendor) there is no downside to removing inefficient operating points from the device tree if you find any.

Why does hardware provide inefficient operating points in the first place?

Energy usage can be approximated as the sum of static and dynamic leakage. Static leakage is the current consumed whenever a circuit is powered and exists because insulation layers within the silicon chip are not perfect. It is voltage dependent but is independent of clock speed. In contrast, dynamic leakage is the energy lost when charge (and the 0s and 1s that charge represents) is moved around the chip, this is both voltage and frequency dependent.

As manufacturing processes shrink the relative costs of static and dynamic leakage have changed. Twenty years ago static leakage was often considered to be negligible but today it cannot be ignored when making power management decisions. Static leakage explains why some low frequency operating points are less efficient than a higher frequency neighbour. If there is a lot of static leakage the best way to reduce power is by completing calculations quickly and entering an idle state that gates the power and therefore reduces that static leakage costs.

These carefully balanced factors can make it difficult to accurately predict energy usage so hardware is sometimes designed to be flexible on the assumption that inefficient operating points, if they exist, will never be used.

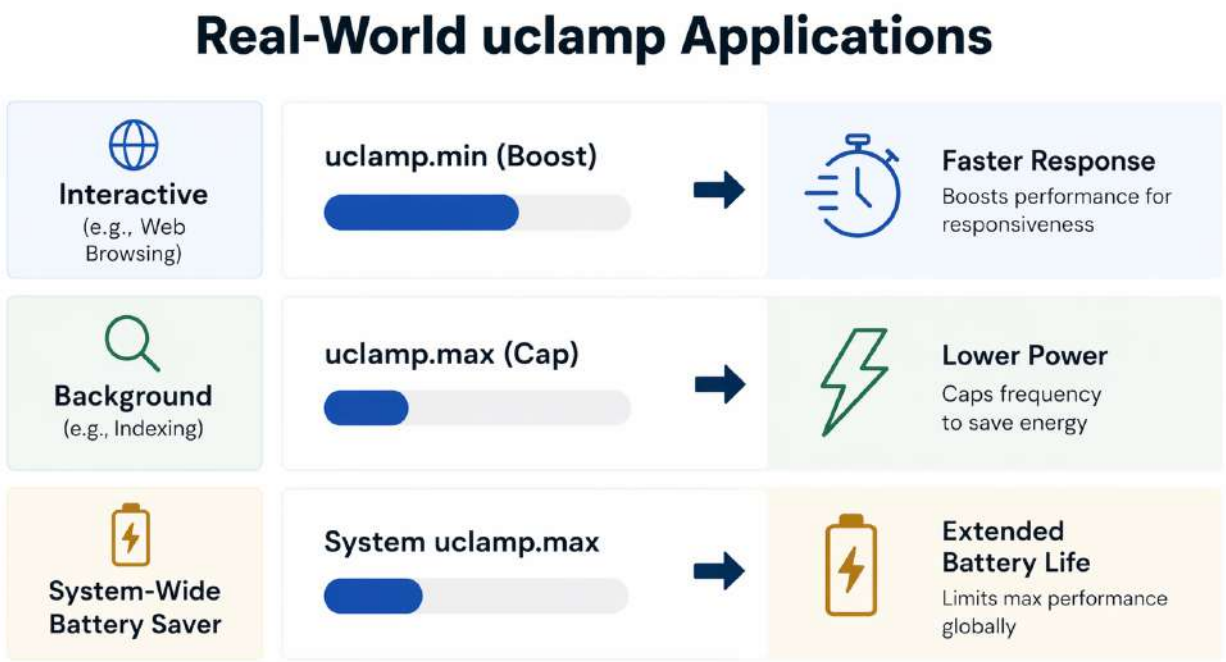
Frequency Clamps

Frequency clamps provide upper and lower limits for the CPU clock frequency. Upper limits prevent the system from adopting the fastest (and least energy efficient) operating points. The upper limit can be set using scaling_max_freq values in sysfs. On devices that have multiple CPU clusters with independent DVFS selection there will be multiple instances of this file in sysfs. Take a look at /sys/devices/system/cpu/cpufreq on a running system to see what policies your system provides.

Once set, schedutil will no longer use higher frequencies (nor would any other cpufreq governor). Frequency clamping, which is a form of deliberate underclocking, reduces the maximum performance of the whole system. It therefore only makes sense to apply this tuning when this reduced performance is acceptable.

Frequency clamping is a great way to implement “battery saver” modes on phones and laptops. It can also be used in modal systems where the current operating mode is known not to require all available CPU resources.

In some cases it may even be appropriate to enable frequency clamping permanently! Permanent frequency clamps are appropriate for battery operated embedded systems where the CPU is overspecified for the mission. This typically occurs when a SoC with a powerful CPU is selected for a particular application because the on-chip peripherals are necessary for the use case. That means the system may not need a powerful CPU but it got one “for free” along with the rest of the SoC.



Utilization Clamps

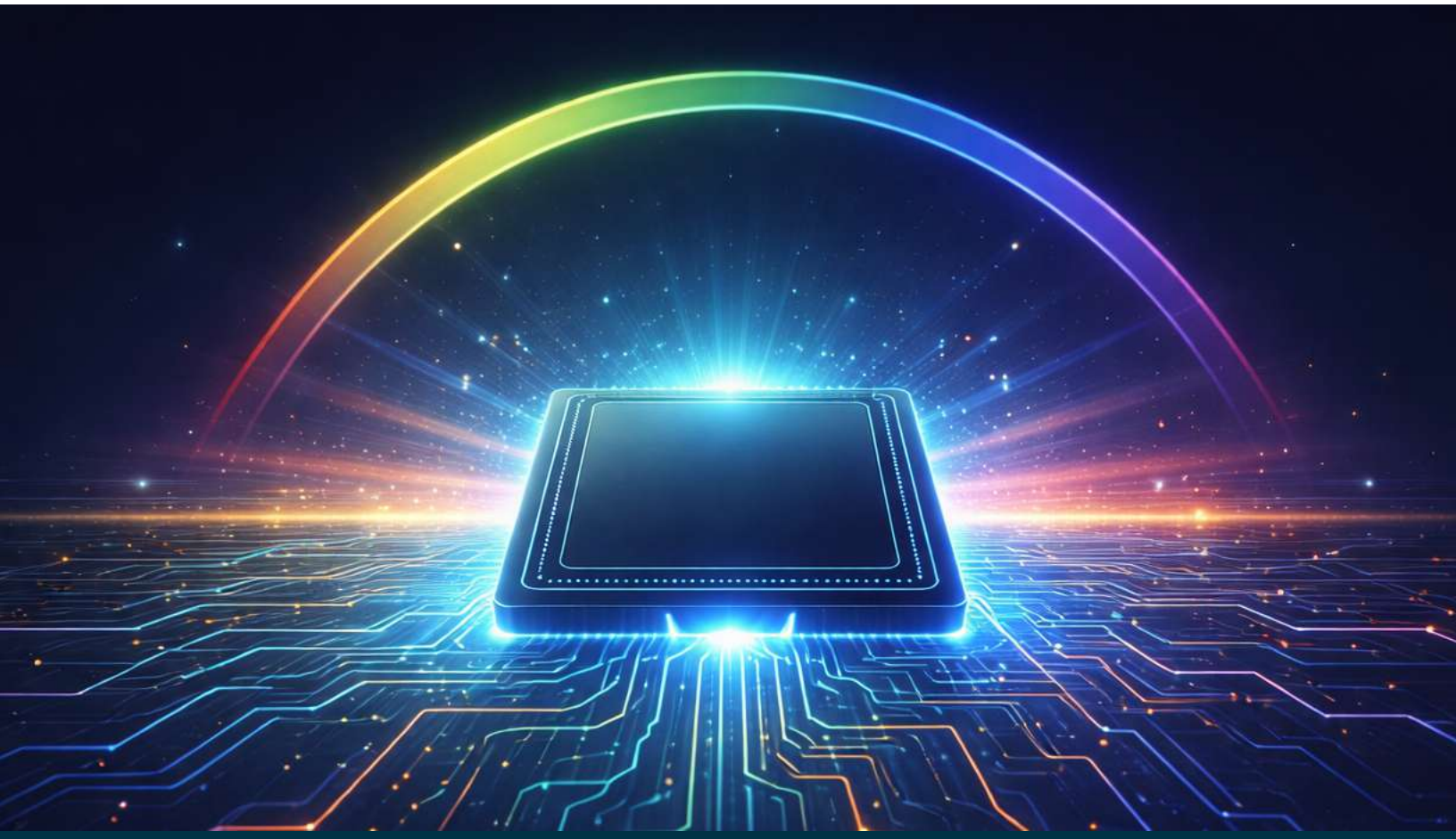
Utilization clamps (uclamp) allow the userspace to force a lower or upper bound on the estimated utilization of a task or of group of tasks. uclamps are used to bias schedutil decisions either to improve performance or to make it less likely schedutil will select the highest (and least energy efficient) operating points.

Lower bounds

Lower bounds are useful for performance boosting. In particular, it can be used to boost the apparent utilization of interactive tasks. When a task clamps at the lower bound then schedutil adopts a higher operating point than it would otherwise. This imposes a greater energy cost but will also improve the time-to-completion which makes interactive applications more enjoyable to use.

Upper bounds

Upper bounds are great for tasks that require lots of CPU bandwidth but no one really cares how long it takes to complete. These tasks are typically CPU-expensive non-interactive tasks, for example regenerating an index, processing photos, housekeeping, etc. Such tasks can be achieved at a lower energy cost by adopting a more efficient operating point whilst still allowing other tasks to transiently appear and demand the full capability of the CPU.



Both bounds are expressed as a ratio on the scale of 0 to 1024 (where 1024 is the maximum capacity of the fastest CPU in the system).

For simple systems uclamps can be supplied on a per-task basis (CONFIG_UCLAMP_TASK). For example, consider a system that is largely interactive but there are a small number of CPU intensive background tasks causing excessive power draw. Applying an upper bound to these CPU intensive tasks prevents them from causing the CPU to adopt the highest (and least energy efficient) operating points.

For more complex systems a grouped approach (CONFIG_UCLAMP_TASK_GROUP) may be easier to manage. Linux systems allow tasks to be collected hierarchically into control groups (cgroups). When a process spawns new tasks they will belong to the same parent cgroup allowing more complex multi-threaded applications to be managed collectively.

For example, on a system managed by systemd each service is separated into its own control group allowing services that consume excessive power to be deprioritized. On our example system the control group for the CUPS printer management service is represented as a directory within sysfs. Sysfs presents each control as a file, meaning to set the utilization clamp for the upper bound we write the desired utilization to /sys/fs/cgroup/system.slice/cups.service/cpu.uclamp.max . Setting the upper bound will hierarchically affect all CUPS tasks (including any rasterizers that are spawned by printer drivers to render content) and can be used to hold the CPU in a more energy efficient operating point during printing.

At the most granular level, utilization clamps can be applied across the entire system. The upper bound, found in /proc/sys/kernel/sched_util_clamp_max provides an effect similar to frequency clamping. In contrast to frequency clamping, which uses real hardware units, utilization clamps allow the limit to be expressed as a ratio rather than being set in hardware specific units. For example, on all hardware 768 is 75% of the maximum capacity. This makes utilization clamps less direct and likely less precise but allows generic operating systems (such as the one on your laptop) to implement portable battery saver modes without specific knowledge of the underlying hardware. For embedded systems either approach is valid and could be influenced by how many different hardware devices your software targets.

With the three different granularities utilization clamps provide a powerful set of tools to allow userspace to describe to the kernel the performance the currently active use-cases require. Upper bounds allow userspace to reduce power consumption for use-cases they don't need the CPU to run at its highest operating points, whilst lower bounds can be used to guarantee performance for sensitive workloads allowing aggressive power management strategies to be pursued system-wide.

Idle CPU power management: cpuidle

Twenty years ago, it was easy for an operating system kernel to go idle: when there were no tasks to run, “the idle loop” would be scheduled. Early idle loops were basically empty infinite loops that did nothing while waiting for the next interrupt to happen. This saved power simply by avoiding running instructions that needed power hungry components such as the cache or FPU!

Over time, changing technology has allowed multiple additional hardware mechanisms to reduce power to be introduced. With these new options available, today the idle loop is responsible for choosing and deploying the “best” way to go idle.

We discussed why modern systems provide multiple idle states in the introduction to this series. As a brief reminder, entering and returning from an idle state has a cost and that cost can be measured both in time and in energy. Typically the shallowest idle state is “nearly free” to enter/exit whilst deeper idle states have increasingly higher costs to enter and exit. If the system enters a deep idle state and wakes up soon after sleeping then energy will have been wasted because the energy cost to enter the deep idle state is greater than the energy saved whilst residing in that state.

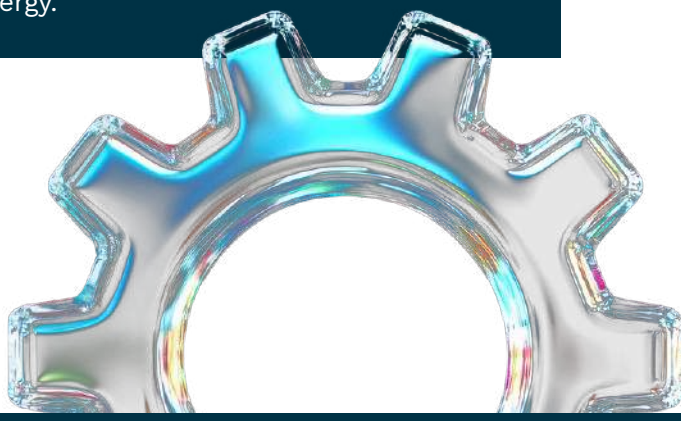
cpuidle is the kernel sub-system that governs idle state transitions. Like cpufreq, which was introduced previously, mechanism is separated from policy, through the use of drivers and governors.

cpuidle drivers provide the mechanism needed to enter and exit idle states. They enumerate the available idle states to the governor. As part of that they describe to the governor the energy saving properties of each state. Finally drivers are able to enter idle states at the request of the governor.

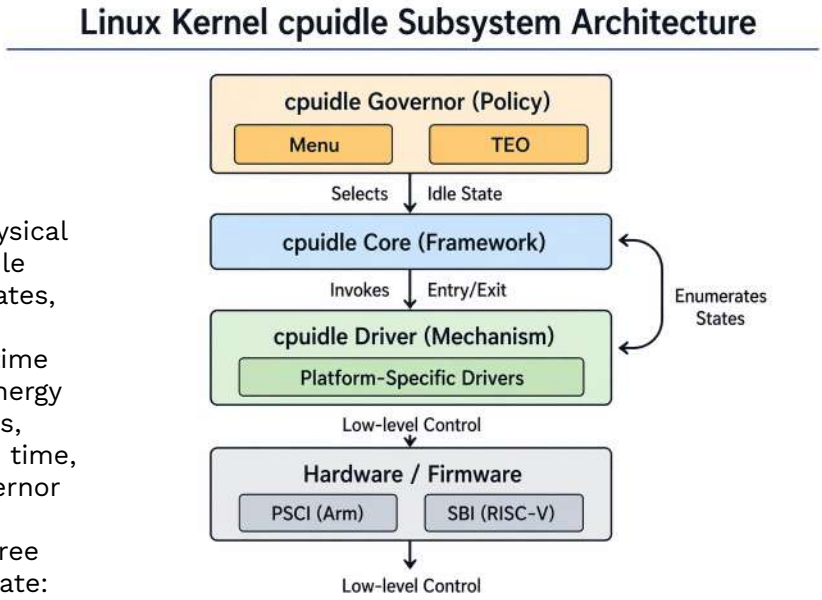
Drivers can be fully customized for the unique properties of each System-on-Chip (SoC). However CPU idle states are often well supported by low-level platform firmware and made available to the kernel using standard interfaces such as PSCI (Power State Coordination Interface) on Arm and SBI (Supervisor Binary Interface) on RISC-V. Thus whilst there is scope for per-SoC drivers, in reality these are becoming unusual. Most modern architectures, including both Arm and RISC-V, define standardized interfaces allowing a single Arm and single RISC-V driver to be shared by a wide range of SoC families.

cpuidle governors provide policy and are responsible for choosing the “best” idle state from among those available.

The governors use the information from the drivers, together with gathered data about historic or known-future events (such as timer wakeups) to make an “educated guess” about when the CPU will need to leave the idle state due to an interrupt. Based on the estimated wake up time it can select the idle state likely to save the greatest amount of energy.



How do cpuidle governors make decisions?



cpuidle governors receive data about the physical characteristics of the system from the cpuidle driver. This takes the form of a list of idle states, where each state is annotated with a target residency. Target residency is the minimum time a CPU must spend in an idle state to save energy compared to shallower states. In other words, although the target residency is measured in time, it actually provides data that allows the governor to compare the energy cost of entering and exiting the different idle states. There are three possibilities when a system leaves an idle state:

1

If a system is awakened before reaching the target residency time, then the system wasted energy by selecting an idle state that was too deep

2

If a system remains in an idle state for longer than the target residency time of a deeper idle state (if there is one) then the system wasted energy because the deeper idle state would have saved more energy

3

If a system was woken after reaching the target residency time for our selected state but before reaching the target residency of a deeper idle state then the choice was optimal

All cpuidle governors keep track of historic idle entry/exit timings, in fact they make them visible for each CPU at /sys/devices/system/cpu/cpu<N>/cpuidle so you can check them yourself!

The governors assume that the length of recent idle periods can be used to predict the future. In fact the ladder governor (used in systems with a regular scheduler tick) and haltpoll governors (specialized governor for virtual machines) work exclusively using historic data.

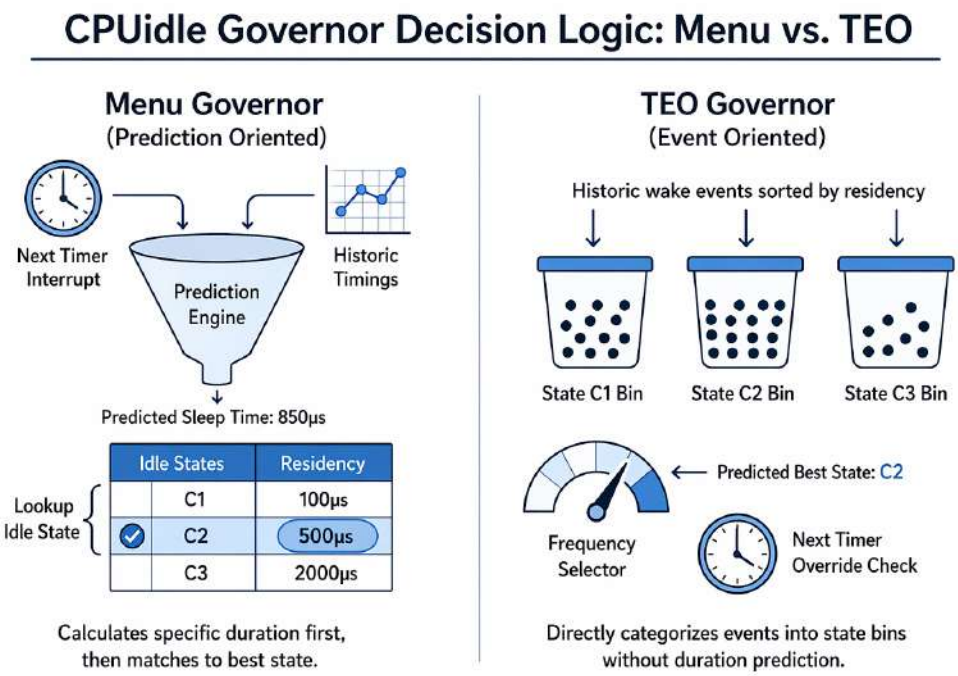
Other governors, such as menu and teo (timer-event oriented), receive a glimpse into the future, although that view is rather limited. In general, drivers don't report when they expect their interrupts will fire but there is one interrupt that can be predicted "perfectly": we always know when the next timer interrupt is due. Thus the upper bound for any idle period is the time to the next timer interrupt. For many use-cases, timer interrupts are significantly more frequent than any other. This means that, providing historic entry/exit tracking indicates we are running a use-case dominated by the timer interrupt, it is an excellent predictor of idle time.

The final factor at play in governance decisions is the energy cost of the decision making itself! All computation has an energy cost. It doesn't matter if the governor makes the best decision if it burns too much energy making the decision! This is especially true when the system is experiencing short idle periods. In that case making a fast decision to enter the shallowest idle state is extremely desirable!

The menu and teo governors both use the timer to inform decisions but adopt different strategies:

The menu governor seeks to generate a predicted sleep time by taking the next wake up time and applying a correction factor derived from recent history to adjust it. Once it has made a prediction it can look up the best idle state.

The teo governor quantizes the historic information into bins based on the target residency times for each idle state. The quantized data does not allow prediction of the idle time but, because each bin corresponds to a specific idle state, it is still able to predict which idle state will be best! It then uses the next wake up time to improve the choice by selecting a shallower mode if the timer will fire shortly. Additional detail about the menu and teo governors can be found in the Linux kernel documentation.



Reducing idle power consumption with cpuidle

All cpuidle governors share something in common with the schedutil governor we covered in earlier posts about cpufreq: the governors themselves do not offer any tuneable values to tweak their heuristics. As we saw before that doesn't mean there is nothing for us to tune, just that to conserve power (or improve performance) we have to look outside of the governor itself. Today we'll be discussing some of those options.

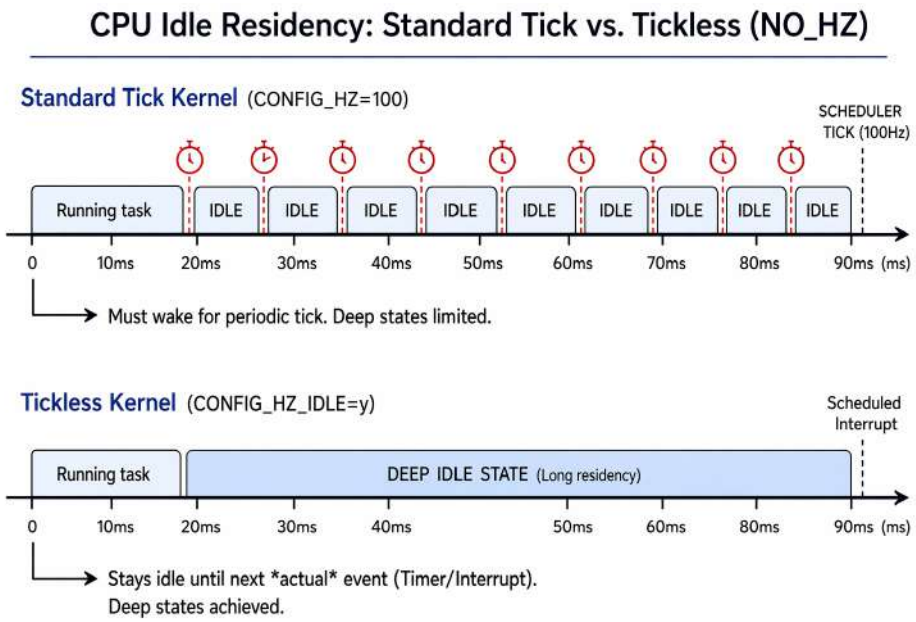
Going tickless

For many years Linux managed the flow of time by establishing a timer that fired 100 times per second and using this "scheduler tick" to swap processes and handle timer expiry in drivers. This tick is configurable and the scheduler tick can be set to 100, 250 or 1000Hz. Changing CONFIG_HZ can have profound effects on system behaviour and it is an interesting kernel tuneable when seeking to balance power consumption and interactivity (although which value results in the lowest power consumption varies with workload).

When CONFIG_HZ is set to the minimum value and the system is waking up 100 times a second, the system can never go idle for more than 10ms. Waking up this frequently can reduce the effectiveness of deep-idle states and should be avoided on systems that seek to conserve energy this way.

Tickless kernels either disable the scheduler tick when the system goes idle (CONFIG_NO_HZ_IDLE=y) or when there is only a single task running on the CPU (CONFIG_NO_HZ_FULL=y). Setting either option allows longer residency in idle states. The exact difference between these options is subtle and not in scope for this post. See NO_HZ: Reducing Scheduling-Clock Ticks if you are interested in more!

Finally we should note that there is little point in going tickless if there is a driver that wakes up 100 times a second to poll, for example, an SPI peripheral! If you have gone tickless you should also make sure that the code running on your system doesn't introduce unnecessary periodic ticks by polling for status. Note also that if polling cannot be avoided then it's still important to ensure the driver shuts the polling down when there are no clients.



Try a different cpuidle governor

If you have a tickless system there is a choice of two governors: menu and teo (timer-event oriented).

The menu governor is the default and works by predicting how long the system will be idle for. It monitors the historic wake up intervals and the due time of the next timer interrupt, filtering them to identify the "typical" wake up interval. It then uses that prediction to choose from the menu of choices offered by the cpuidle driver.

The teo governor uses the same sources of data but tracks the statistical data to directly predict the best idle state, without predicting exactly how long it expects the system to remain idle for. When the system wakes up frequently this allows it to avoid the (relatively expensive) peek at the timer queue reducing energy costs of its decision making.

The governor can be inspected and changed via sysfs: /sys/devices/system/cpu/cpuidle/current_governor.

Power Management Quality of Service (PM QoS) Requests

As mentioned in the introductory sections, entering/leaving idle state has a cost that can be measured in both time and in energy. The cpuidle governor usually makes decisions based on energy cost but there are situations where we have to consider the time cost as well. For example if we are in a deep idle state it can take a long time to get the CPU running again. What if an important interrupt arrives during a deep idle state and we don't respond fast enough? We don't want the idle system to cause us to miss real-time deadlines, such as refilling an audio buffer.

One way to solve this is to disable the deep idle state, but doing that globally will cause energy to be burned unnecessarily when not running time-sensitive applications.

A better way to address this is to ensure all drivers and userspace register their latency tolerance with the PM QoS framework. The latency tolerance is a value in microseconds that expresses how much additional latency due to CPU idling a driver or userspace process can tolerate before their performance is degraded. cpuidle chooses the lowest values among all drivers and processes and prevents the governor from adopting deep idle states if the entry/exit time is too long.

This is great for modal systems where entering deep idle states is useful for conserving power but it is not appropriate to enter deep idle states in all modes.

RISCSTAR™ SOLUTIONS

Optimize Your Embedded System's Power Management

Is your Arm or RISC-V platform reaching its full power efficiency potential? The complex interplay of active states, idle states, and DVFS can make a profound impact on your device's battery life and performance.

Speak with a RISCstar Power Management Expert

Our engineers specialize in optimizing power utilization for embedded Linux systems, helping you maximize battery life without sacrificing performance. Let us help you implement the perfect balance of power management features for your specific use case.

RISCstar.com

contactus@RISCstar.com