

Processor Power Management features and Process Scheduler: Do we need to tie them together?

Venkatesh Pallipadi

venkatesh.pallipadi@intel.com

Suresh B Siddha

suresh.b.siddha@intel.com

Intel Open Source Technology Center

Abstract

Power savings is a key focus area in today's microprocessors, with almost all latest microprocessors providing wide variety of power saving features. Processor P-state is the capability of running the processor at different voltage and/or frequency levels. Processor C-state is the processor capability to go into various low power idle states (with varying wakeup latency). Linux kernel policies like cpufreqondemand governor and cpuidle-menu governor make effective use of these processor power management features, giving power savings to the end user. Linux kernel scheduler also has power management related switches, which lets the administrator to switch between power v/s performance scheduling policy on platforms with multi-core and hyper-threading processors.

This paper looks at various inter-relations between Linux power management features and process scheduler. In particular, it covers various issues and mechanisms for incorporating power management related information in process scheduler. Paper focuses on merits demerits of different solutions and challenges involved. Paper will also look into how the cur-

rent power v/s performance scheduler switch can be made automatic.

1 Introduction

Processor power management has been an area that is getting a lot of attention in recent years. That has resulted in wide variety of processor power management features like processor P-states and C-states. Linux kernel has drivers and driver infrastructure to support these features.

Basic support for such processor power management features is a nice starting point. But, such support overlooks the fact that many of those features can be inter-twined with different kernel components. Specifically, P-states and C-states are inter-related and also coupled with process scheduler and processor features like Hyper-threading, Multi-core etc.

This paper takes a look at such inter-dependencies, changes and optimizations in Linux kernel to make overall system performance/power efficient. The paper starts with some background information in section 2. Then looks at the ways to introduce automatic

power and performance switches that adapt to the system conditions and fine tune existing solutions in section 3, followed by highlighting the inter-dependencies among the components and way to address them in section 4. Paper concludes in section 5.

2 Background

2.1 P-states, cpufreq and ondemand governor

Processor P-state is the capability of processor to switch its operating voltage and/or frequency at run time. This capability allows the processor to provide different performance levels based on the current requirements of the system. The main benefit of the feature being the reduction in the processor power consumed at lower voltage-frequency states [6].

cpufreq is the generic infrastructure in Linux kernel to handle processors with P-state capability[3] [4].

ondemand governor is a kernel driver that manages the processor frequency/voltage dynamically, based on current processor utilization [7].

2.2 C-states, cpuidle and menu governor

Processor C-state is the capability of processor to support multiple idle states; states in which processor does not retire instructions. Such idle states are characterized by the amount of power consumed while in that state and the latency to enter/exit that state (and may also vary in amount of content preserved in the processor across such a state entry and exit).

cpuidle is a currently in development infrastructure, to support processor idle states in a

generic way. Menu governor is a cpuidle policy manager that determines the optimal idle state that the processor will use dynamically [8].

2.3 Other related processor features

Hyper-threading Technology is a processor feature that provides the support for multiple logical threads of execution on a single processor core. Threads aim at increasing the utilization of core level resources. Hyper-threading Technology introduces some key interactions across power, performance and optimal scheduling that are detailed in later sections.

Another processor feature that has impact on power, performance and scheduling is Intel[®] Dynamic Acceleration [1]. This is a feature where in a processor can provide more frequency than advertised, provided there is enough thermal power headroom and the system has the need for this increased frequency.

2.4 Process scheduler and power v/s performance switch

Linux kernel process scheduler has /sys/fs switches to switch between performance and power scheduling policies. These switches, for hyper-threading and multi-core domains, impacts the process load balancing in lightly loaded cases (where number of active processes are less than the number of available logical CPUs). In performance mode, load balancer tries to keep each processor package busy by distributing the processes across packages while certain logical processors in the packages may be idle. This allows processes to get greater amount of resources, thus providing better performance. In power saving mode, load balancer tries to keep all logical processors in a package busy, before allocating processes on another processor package. This lets

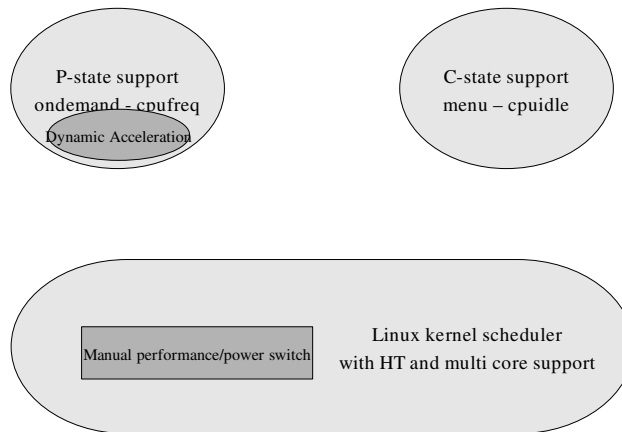


Figure 1: Current state of Power Management and Process scheduler

entire packages to be idle, there by reducing the power consumed [5].

2.5 Existing kernel solution

The P-state management, C-state management and power/performance policies in scheduler support in current kernel (2.6.22) [2] are independent of each other. They are done in a stand alone way by separate part of code in Linux kernel and they do not interact with each other.

This paper highlights the interdependencies and interactions across these different features and we look at various ways of tying these features together. The goal is to optimize the power performance under diverse workload conditions with minimal user interaction.

3 Fine-tuning switches

For any policy or optimization to be fully useful to the end user, it has to be auto-tunable. Following section looks at oppurtunities to introduce automatic switches in power management and scheduler area.

3.1 Automatic scheduler power performance switch

As hinted in section 2.4, there is a tunable which lets administrator to pick among the performance and power setting in scheduler. With that option, administrator can switch between:

performance mode - Where tasks are distributed equally across the processor packages first, before other cores/threads in the package gets tasks to run. This lets tasks to maximize the utilization of resources in a package, there by getting the high performance

powersave mode - Where tasks are distributed among the cores/threads of a package first, before they are distributed to another packages. This lets entire packages to be idle conserving power while one package makes full use of its cores/threads. Note that there may be some performance penalty in this mode as cores/threads share package resources.

Note that the current tunables are global, system wide settings.

Is there a way to get best of both worlds without actually involving the system administrator?

First challenge is to make this auto tunable. And the second challenge is to efficiently incorporate the auto tunable knowledge into the process scheduler by incorporating the performance Vs power mode selection at each resource sharing (perf-domain) or power sharing domain.

To address the first challenge, one needs to know the shared resource usage for individual tasks. Based on individual task usage and the available shared resources per domain (typically per package), need for performance policy with in that domain can be determined. In today's platforms, one need to rely on performance monitoring counters to get an estimate of the resource usages and there is no easy way for software to come to conclusion that the shared resources are getting contended, from that information. Also, performance monitoring counters are mostly architecture specific and mostly varies from processor generation to generation. Quite a bit of hardware and software research is going on in this area. In the absence of precise information, we can explore some heuristics to characterize the task as resource intensive. For example, we can rely on task's RSS to characterize it as memory intensive (and hence cache intensive) or not. Similarly, we can treat highest priority tasks as resource intensive and implement performance policy for the domain where the task is running. One can also use the individual CPU's utilization and use performance policy for a CPU's domain, if that particular CPU is 100% busy. While the heuristics are not entirely accurate, if the process scheduler infrastructure is in place, one can simply replace the heuristic with a better algorithm when one surfaces.

Second and equally interesting, more important compared to the first challenge, is the efficient

scheduler implementation that takes the decision from the first step and enforce them. First step mentioned above can trigger the resource contention issue that is happening on a particular domain. If the system is lightly loaded, in addition to regular CPU load balance, periodic idle load balancer can also look at the shared resource usages on different domains and can minimize the resource contention by making the leastly loaded (from shared resource perspective) domain, pull the resource intensive load from the contended domain. Or for power savings, the leastly loaded domain can pull the load from other leastly loaded domains to minimize the number of power-domains carrying load.

This is an area the authors are actively exploring currently.

3.2 C-state governor dependency on real time process scheduling

Linux kernel today has an interface in place for drivers to limit the idle latencies (`set_acceptable_latency()` and friends). This interface allows drivers to limit the C-state the kernel will try to use while the limit is set.

One limitation of this interface is that it is system level. Even if there is one audio/video playback process that announces a low latency, all logical CPUs in the system cannot go to deep C-state.

Addressing this limitation can be done using combination of different approaches below:

- **RT Process** The latency limit control can be made per process and automatically set to the processors that have user level RT tasks assigned to them. Latency limit needs to be set when RT task is sleeping as well, to address the wakeup latency.

- `timers` The latency limit control per process and use that while setting any timer on behalf of that process. Once this information is included in timer structure, latency limit can be enforced while going to idle just before the expiration of particular timers.
- `interrupts` Hardest problem to solve is enforcing latency requirements for interrupts. Potentially, there can be a IRQ sharing, where one process wants to have latency enforced on and other process do not care about the latency on interrupts it needs from that IRQ. Kernel drivers need to keep track of process that may utilize the interrupt and enforce latency on processors that are supposed to take interrupt on that particular IRQ, using the lowest latency time requested among all the processes.

At the time of writing this paper all the above options are being explored and incremental patches to address this is expected soon.

4 Resolving interdependencies

4.1 P-state governor dependency on C-state governor

Ondemand governor today tries to run each processor at slowest possible frequency that can keep the tasks running on that processor happy. When there is a task that keeps processor at say 70% busy (at highest freq), ondemand will try to find a P-state that is slightly higher than 70% so that the processor can run at low freq/low voltage for most of the time. But, this policy may not be the right one in all conditions, especially when there is a deep C-state that can save lot more power at idle. In such scenarios,

kernel can potentially conserve more power by running highest frequency while processor is busy and later going into a deep C-state state. This policy, also referred as race-to-idle policy, should take the current C-state latency requirement into consideration (otherwise processor may not really enter the deep C-state), and do the race-to-idle only when it expects it to be beneficial for overall power.

To understand the scenario, consider a numerical example. Table 1, row 1, gives the power numbers from an actual system. Power consumed when the system is fully idle and power consumed while running 100% CPU load on one core at all supported frequencies. Row 2 is not the actual measured number. But, it is a hypothetical example which is closely related to Row 1, but with much less idle power. Row 2 is only used to show the effect of power numbers on policy selection.

Now consider system running at various loads, with power consumption according to table 1. Energy consumption can be calculated using the power numbers in table, at different loads by taking into account the amount of time spent in idle and the amount of time spent running at different frequencies (assuming the load scales directly with frequency). Figure 2 shows a chart with energy consumption in y-axis at various loads in x-axis (relative to highest frequency). As seen in the figure, ondemand shows lower energy consumption than race-to-idle with base power numbers. But, with lower idle power numbers, race-to-idle performs better especially at lower load.

As can be seen from the data, different policies can be better on different situations. It depends on system power numbers at different frequencies and at idle. Further, on a same system, different policies may be better depending on the load.

	Idle Power	CPU @ Max	CPU @ 0.8 Max	CPU @ 0.6 * Max	CPU @ 0.4 * Max
Measured System power	13.7	27.3	22.7	19.4	17
Hypothetical System with lower idle power	8.7	27.3	22.7	19.4	17

Table 1: Power numbers for idle and different frequencies (in Watts)

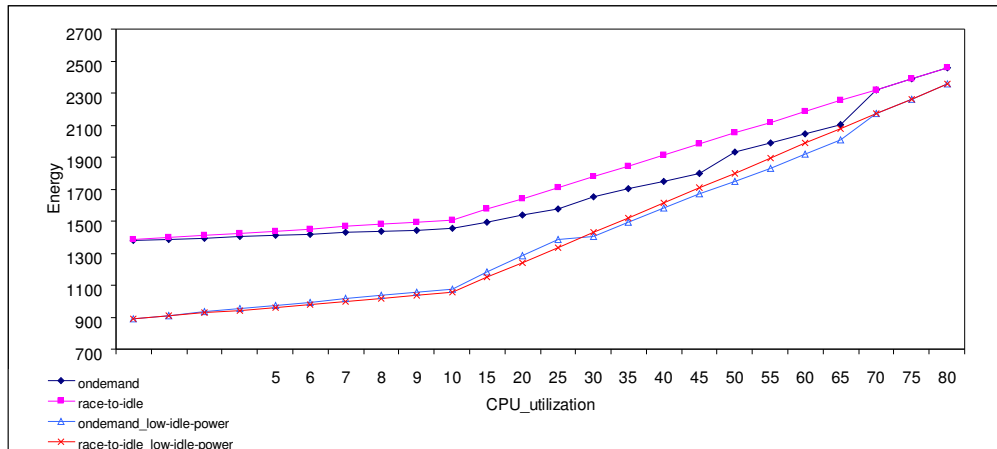


Figure 2: Energy consumption at different loads with DBS and race-to-idle

4.1.1 Implementation details/challenges

First step here is to implement a simple race-to-idle policy that a user can switch to instead of ondemand. More important challenge here is to identify dynamically which policy will be better under what conditions and using that policy transparent to the user. Better still is to have one single governor that can switch to different modes depending on the conditions. This is an area that needs further investigations and analysis.

4.2 P-state governor dependency on scheduler statistics

Ondemand governor currently uses the processor statistics from kstats which is of jiffy granularity. This granularity is not very accurate in terms of idle times and ondemand adds a big

guard band across this statistics to prevent any wrong frequency decisions. Tracking idle time statistics in each processor in more fine granularity (like microseconds), ondemand can get rid of guard band and be more aggressive in choosing the low frequencies there by increasing the power advantages.

Adding this micro-accounting has become simpler now with tickless and CFS related changes in area. Only additional change needed is to account for interrupt activity on an idle CPU correctly. The patch to do this and use this information in ondemand governor is in flight and should hit lkml very soon.

4.3 Load balancing for HT processor and power savings

With current process load balancing for HT processors, on a relatively lightly loaded sys-

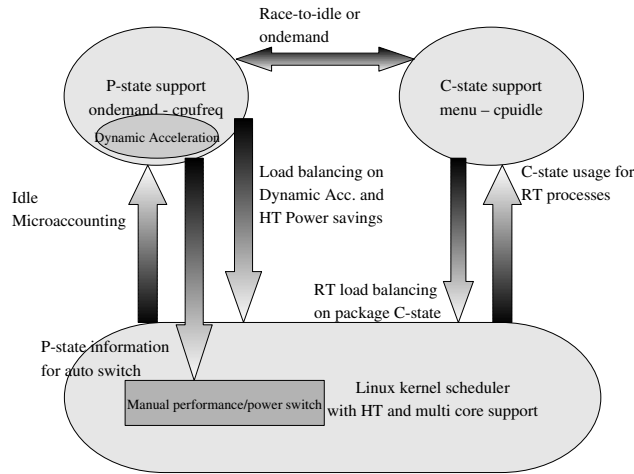


Figure 3: Power Management and Process scheduler dependencies

tem, threads are spread across cores or packages, before they are scheduled together on HT siblings. For example, if there are only two low priority threads active on a system, they will get scheduled on different cores or sockets (on multi-socket systems), with HT siblings being idle. With this kind of scheduling, as different cores are active, they cannot take the advantage of power savings fully. Better scheduling in the above case is to schedule both low priority threads on the same core on HT sibling CPUs. That will enable other complete cores or sockets to be idle saving more power.

4.4 Dynamic Acceleration dependency on process scheduling

Dynamic Acceleration provides an opportunistic performance boost provided there is enough power and thermal headroom to do so. The scheduler load balancing plays an important role in deciding the combination of various tasks that runs on a package, that can provide opportunities for such dynamic acceleration. One way to increase the opportunity of performance boost is to look at combinations of low priority and high priority tasks on different

cores of same package and when Dynamic Acceleration feature is available, scheduler can introduce certain "idle time" for low priority process so that the high priority process gets some performance boost.

There are more ways to take the benefit of Dynamic acceleration in a fair (CPU power proportional to process priority) manner and all of those methods require scheduler to be aware of real processor frequency and processor capabilities like Dynamic Acceleration. This area is being actively pursued at present, especially as it fits nicely into overall CFS architecture. Accounting the timeslice based on frequency rather than wall clock time will also help in case of frequency reduction due to P-state governor.

4.5 C-state dependency on real time process load balancing

Real time tasks disabling high latency C-states as described earlier section 3.2 helps them to keep good response time. But, if there are multiple processes, one assigned to each package, all packages may end up not utilizing deep C-states because of implicit dependency of deep

C-states across cores in a package. For example, on a Dual socket system, with each package having two cores, 2 RT tasks on different packages can make all four cores not enter deep C-state. But, if those two RT tasks are scheduled together on 2 cores of a single package then other package will be free to enter deep C-state saving power. So, there is a potential to factor in this aspect into load balancing of RT process scheduler. More analysis of interactions like this are being done.

5 Future Work

With various interactions and optimizations discussed in this paper, power management and scheduler looks more as in figure 3. This paper highlights issues and incremental changes to address these issues. Though the paper does not talk about any modular framework to handle these inter-dependencies, it may well be in order for future.

References

- [1] Intel centrino duo processor technology.
<http://www.intel.com/products/centrino/duo/description.htm>.
- [2] Linux 2.6.22.
<http://www.kernel.org>.
- [3] Linux kernel cpufreq subsystem.
<http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufreq.html>.
- [4] Dominik Brodowski. Current trend in linux kernel power management, linuxtag 2005. http://www.free-it.de/archiv/talks_2005/paper-11017/paper-11017.pdf.
- [5] Suresh B Siddha et al. Chip multi processing aware linux kernel scheduler, ols 2005. http://www.linuxsymposium.org/2005/linuxsymposium_procv2.pdf.
- [6] Venkatesh Pallipadi. Enhanced intel speedstep technology and demand-based switching on linux, intel software net. <http://www.intel.com/cd/ids/developer/asmo-na/eng/195910.htm>.
- [7] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor, ols 2006. http://www.linuxsymposium.org/2006/linuxsymposium_procv2.pdf.
- [8] Adam Belay Venkatesh Pallipadi and Shaohua Li. cpuidle - do nothing, efficiently ..., ols 2007. <http://ols2006.108.redhat.com/2007/Reprints/pallipadi-Reprint.pdf>.

This paper is (C) 2007 by Intel. Redistribution rights are granted per submission guidelines; all other rights are reserved.

* Other names and brands may be claimed as the property of others.