Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
Matthew Wilson, *rPath*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
Eugene Teo, *Red Hat, Inc.*
Kyle McMartin, *Red Hat, Inc.*
Jake Edge, *LWN.net*
Robyn Bergeron
Dave Boutcher, *IBM*
Mats Wichmann, *Intel*

Energy-aware task and interrupt management in Linux

Vaidyanathan Srinivasan, Gautham R Shenoy,
Srivatsa Vaddagiri, Dipankar Sarma
IBM Linux Technology Center
{svaidy, vatsa}@linux.vnet.ibm.com, {ego, dipankar}@in.ibm.com

Venkatesh Pallipadi
Intel Open Source Technology Center
venkatesh.pallipadi@intel.com

Abstract

As multi-core and SMP systems become more generally available, energy management needs in Linux™ have also become more complex. Energy management in Linux was primarily designed for interactive systems where relatively simple inactivity based strategies worked effectively for most cases. Modern enterprise class hardware needs a more complex power management strategy to save energy with the least impact on the performance of enterprise workloads. Traditionally, the Linux kernel for servers has been optimized for throughput and not power efficiency.

This paper discusses the behaviour of the current task management subsystem (scheduler and loadbalancer) on a multi-core SMP system and its effectiveness in saving energy consumption under several situations (idle, moderate load). It then describes several techniques such as timer migration, task wakeup biasing and related heuristics for reducing energy consumption. The paper also looks at possible methods to mitigate interrupts for energy savings during different workloads and concludes by discussing some results of these new strategies.

1 Introduction

Traditionally, operating systems designers have focussed on optimizing for performance. The key design goals have been to make maximum usage of resources to get the most out of the underlying systems. On multiprocessor and multi-core systems, this approach led to using all CPU resources in parallel as much as possible. This approach to system design had to be re-evaluated when battery operated low-power devices became important. Various system technologies like DVFS (Dynamic voltage and frequency scaling) and exploitation of them in operating systems led to significant improvement in power consumption [1][4]. Various operating system techniques were adopted to manage tasks with a goal of reducing power consumption [17][6][18]. With the advent of multiprocessor systems, additional techniques have been used to do CPU power management [15]. With the cost of energy going up in recent years, the need for energy efficiency has been acutely felt across the entire spectrum of systems—small handheld computers to large multi-processor servers in datacenters. Due to increased computation density of modern enterprise servers, the thermal limits of the design is beginning to constrain the integration and performance. Power management in enterprise servers primarily help data centers improve their computation density by getting more computation done without increasing power consumption. The objective of power management in laptops and other battery powered devices has been primarily to extend the battery life, while on enterprise server and datacenters, power management forms the building blocks to provide higher level services like power trending and power capping. Thermal management, which is an interesting side effect of power management, and power capping are of great interest to enterprise customers. Fundamentally, enterprise customer would like to control parameters that have been previously considered passive and hence ignored.

This paper investigates power management in two areas to simplify the discussion, namely

[1]With additional contributions from **Suresh B. Siddha**, *Intel Open Source Technology Center*, suresh.b.siddha@intel.com

1. Idle system power management
2. Power management in under-utilized (or non-idle) systems

An idle system is one where no useful work is done by the system with respect to its workload and applications. Such a system could be waiting for inputs from user or requests on the network. On this kind of system, it is usually the system house keeping jobs that are active.

On a non-idle system, the system is actively running the workload or application but the overall system capacity is under-utilised. This provides scope to perform several run-time power management strategies such as frequency and voltage scaling. In the Linux kernel, the ondemand governor [11] does the job of selecting the correct CPU capacity or frequency that would match the current workload.

The new SPECPower benchmark [2] tries to characterise performance-per-watt under various system loads.

Avoiding the periodic scheduler tick in an idle system with the tickless kernel feature significantly helps to save power in an idle system, while process scheduler tweaks are needed to save power in an non-idle system. The next section explains the problem space and existing solutions in detail.

2 Scheduler Overview

In a multi-processor system, an important goal for a power-aware operating system is to consolidate *all* activity (like execution of tasks, interrupt handling etc) on fewer CPUs so that remaining CPUs can become idle and enter low-power states. That implies a constant tug between providing good throughput for applications and providing good energy savings.

We now provide a brief overview of Linux CPU scheduler, how it is currently meeting the needs of a power-aware operating system and some potential enhancements to make it more energy-concious.

- **CFS scheduler**
  From the primitive v2.4 scheduler, to the scalable O(1) scheduler in v2.5, to the more recent scalable and responsive CFS scheduler, the Linux cpu scheduler has significantly changed several times. The most recent rewrite, termed CFS (Completely Fair Scheduler), was authored by Ingo Molnar [8] and has been adopted since v2.6.23 (July '07). It was mainly written to address several interactivity woes that the Linux community complained about. Its salient highlights are:
  - More modular scheduler core by introducing scheduler classes
  - Time indexed rb-tree as runqueue for SCHED_OTHER tasks
  - Excellent interactivity for desktop users by doing away with concept of fixed timeslice
  - Scheduler tunables
  - Group scheduler – divide bandwidth fairly between task-groups first and then between tasks in the group
- **Power awaress existing in current Linux scheduler**

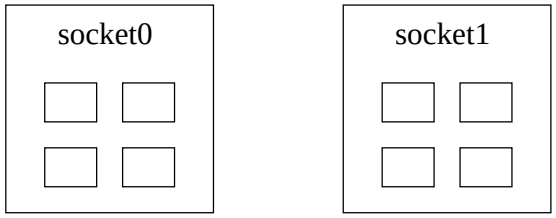

Figure 1: Two socket quad-core system

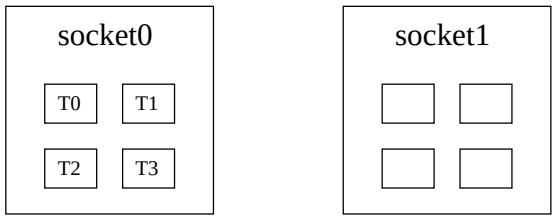

Figure 2: Good task distribution from power perspective

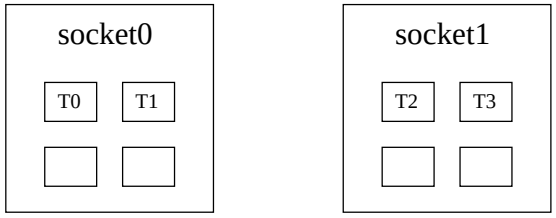

Figure 3: Bad task distribution from power perspective

Linux CPU scheduler has options, both compile

time and runtime [10], through which it can be directed to consolidate tasks across fewer CPUs rather than spreading them apart on all available CPUs for improved throughput. This lets more CPUs to become idle and thus enter low-power states when system is not heavily loaded. Additionally, the Linux scheduler is aware of the underlying multi-core and NUMA topology. This makes it possible for the scheduler to further optimize power-savings. For example, consider a multi-core system as shown in Figure 1. The system has two sockets, each of which can accommodate a quad-core chip. Typically in such systems, the granularity at which frequency/voltage can be varied is at each chip or package level [5]. In other words, the frequency/voltage cannot be different for different cores resident in the same chip. In such a scenario, the CPU scheduler is required to recognize such *power* domains and work towards not only consolidating tasks on just fewer cpus but also on fewer *power* domains (in this case, the chips). As an example consider that the system in Figure 1 had just four tasks. Then it is better for a power-aware CPU scheduler to consolidate these 4 tasks on the 4 cpus in same chip (as in Figure 2) rather than on *any* 4 arbitrary cpus (as in Figure 3). Linux CPU scheduler has the capability to do this chip-wise consolidation of tasks when required.

- **Areas for improving power-awareness in scheduler**

  Consolidating tasks on fewer cpus and chips relies on accurate cpu load (number of tasks on a cpu) calculation. Since cpu load is sampled periodically, it is possible that short lived tasks (ex: daemon that run periodically for short intervals of time) don't show up as cpu load, which can result in failure to consolidate on fewer cpus/chips. This is discussed in detail in the Section 3.5. Typically the total CPU time utilised by the daemons in an idle system will be less than 1% but the distribution of this jobs across all CPUs influence the CPU's low power sleep time thereby affecting the power consumption at idle. Section 3 describes the idle system in detail.

  In addition, CPUs that are in their low-power states can be interrupted prematurely by task wakeup code, which attempts to schedule waking tasks on the same cpu where they last slept.

3 Idle system power management

Apart from the applications or the workload, there are a host of system daemons, device drivers, and interrupt processing that happen in an idle system. If the operating system and hardware can be optimised to significantly reduce these house keeping tasks, then an idle system can sleep for longer duration leading to power savings. There are significant activities even on a tickless idle [12][7] system that reduce the duration of a CPU's low power sleep time leading to an increase in the energy consumption at idle.

The objective of idle system power management in an enterprise server is to consolidate daemon tasks and interrupts to fewer CPUs or packages in an idle system. Typically an enterprise server would have more than one CPU in SMP configuration. Multi-core processors have helped increase the number of cores in an enterprise system. Dual socket server can have dual core processor modules thus forming a 4-way SMP configuration. Optionally processor threading feature can be enabled which would further increase the number of logical CPUs to eight. In this sample configuration, (assume no threading) we have 4 logical CPUs in two physical packages.

Under low system utilisation or idle, if all the housekeeping work can be consolidated to one package, then the other package can continue to be in low power sleep state, thus saving power. In general the SMP scheduler will try to spread the workload across different package for better throughput. This is the main design point in power savings and performance trade off.

If the number of tasks to run is less than the number of cores, spreading the tasks to one core on each physical package will provide better throughput (assuming the tasks do not share data), while consolidating them on one physical package is better for power savings.

In the following subsections we shall discuss the challenges involved in consolidating daemon jobs in an idle system to one physical package in a dual package system.

3.1 Process, timers, and interrupts at idle

The daemon process that are running in an idle system can be easily identified using *ps* or *top* commands.

Other than the processes that use CPU time, there could be interrupts from IO devices like ethernet and harddisk that wakeup CPUs and consume power. Timers programmed by applications and device drivers are actually interrupts that wakeup CPUs when the timer expires.

CPU utilisation and interrupt rate can give a good idea of the idleness of the system. All these events are required for normal system operation, however in an idle system, these events contribute to reduced sleep time of the CPU.

In a typical distro installation,[1] CPU utilisation from *top* and process wakeup rate from *powertop* at idle are shown in Table 3.1.

The number of interrupts observed during the 15-second duration is listed in Table 1.

IRQ	CPU0	CPU1	CPU2	CPU3	Description
17	21				ata_piix
214		61			eth0
LOC	66	55	95	71	Local timer interrupts
TLB		1		1	TLB Shootdowns

Table 1: /proc/interrupts diff for 15 seconds

In order to improve CPU sleep time, idle polling activities should be reduced and moved to asynchronous notification. USB inherently needs to have time based polling loops. USB auto suspend will work as long as there are no devices connected, but if the USB port is being connected even to an idle keyboard, the polling loops are needed.

Since the introduction of powertop utility, the behaviour of user space applications and drivers have significantly improved and moved away from unnecessary polling. On an enterprise hardware with multiple CPU packages, the timers and interrupts that cannot be reduced can be consolidated to one CPU package. This provides new opportunity for power savings by allowing parts of the system to be more idle. The objective of idle system power management is to significantly increase the idle time for some of the CPU packages in the system.

For our discussion lets assume *idle time* is the duration over which a CPU is in tickless idle state. During this time, no task is scheduled on the CPU. Thus, the CPU

[1] Fedora 9 beta was used in this experiment.

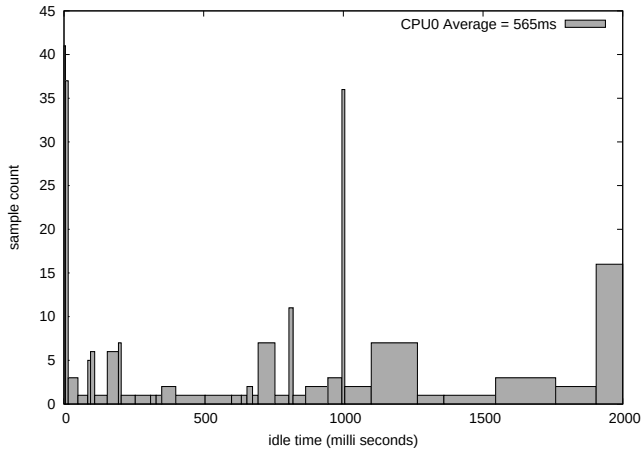

Figure 4: Fedora 9 distro on 4 CPU system

can potentially go to low power sleep state and save power. However in this state, the CPU can receive interrupts. The interrupts can be from an IO device or a programmed timer. Hence the *idle time* duration can further be fragmented by interrupts and timers. Lets call the time interval between such interrupts where the CPU can really sleep in low power mode as *sleep time*. Tickless kernels that turn off periodic timer interrupts have a significantly long *idle time*. However only the uninterrupted *sleep time* contributes to power savings. Hence, to characterise various scenarios, we extract two parameters, namely the CPU *idle time* and *sleep time*. *Sleep time* can be obtained by *idle time* divided by the number of interrupts and timers received during the interval.

Idle time distribution can be obtained by instrumenting `tick_nohz_stop_sched_tick()` and `tick_nohz_restart_sched_tick()` code [13]. Figure 4 plots the histogram of idle time obtained on one of the CPU in a typical distro. Basically the idle time values for 120 seconds in an idle system has been converted to a histogram for easy visualisation. The x-axis is the idle time and y-axis is the sample count observed during 120 seconds. This gives an idea of expected sleep time for a given CPU in a multi-cpu system. Actually this experiment was done on a two socket dual core system and such histograms are available for each of the four CPUs. The distribution is similar in other CPUs and thus we will not discuss the histogram for all four CPUs. As observed in the histogram, the maximum idle time was 2 seconds while most of the samples are concentrated at less than 10ms. There is a pattern of 1s idle time as well.

Figure 5 is a histogram of sleep time. Sleep time is much smaller than the idle time and inversely proportional to

Utilisation from top:

```
Cpu(s):  0.0%us,  0.1%sy,  0.0%ni, 99.9%id,  0.0%wa,  0.0%hi,  0.0%si, 0.0%st
```

Output of powertop -d:

```
PowerTOP 1.8     (C) 2007 Intel Corporation

Collecting data for 15 seconds
< Detailed C-state information is only available on Mobile CPUs (laptops) >
P-states (frequencies)
  2.40 Ghz     0.0%
  2.13 Ghz     0.0%
  1.87 Ghz     0.0%
  1.60 Ghz   100.0%
Wakeups-from-idle per second : 10.8     interval: 15.0s
Top causes for wakeups:
  28.7% (  4.0)   <kernel module> : usb_hcd_poll_rh_status (rh_timer_func)
  27.3% (  3.8)       <interrupt> : eth0
  10.0% (  1.4)       <interrupt> : ata_piix
   7.2% (  1.0)                ip : bnx2_open (bnx2_timer)
```

the interrupt rate. The maximum sleep time was 400ms while the typical sleep time was less than 10ms.

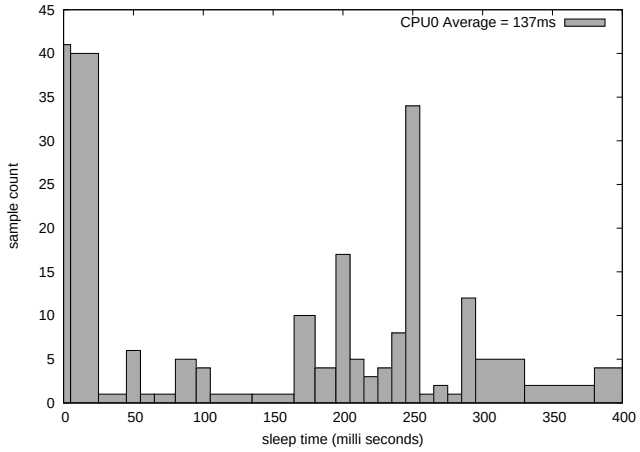

Figure 5: Fedora 9 distro on 4 CPU system

In order to analyse the effect of various kernel tunables and scheduler changes we need to derive a metric for comparison. The obvious and simplest metric is the average idle time and average sleep time on each CPU. Basically the weighted average of samples obtained from the histogram for each CPU in the system gives the average idle and sleep time values. The approximation in assuming average value per CPU over long duration is that even marginal change in average value could significantly affect the power savings. The longer the CPU is in sleep state, the more power is saved. Small number of long sleep intervals is better than large number of small sleep intervals. Hence to improve the accuracy of the evaluation model and its correlation with real power consumption, a weight factor may be needed for each sleep duration corresponding to the processor's deep sleep state transition latency. Based on typical processor sleep state transition latencies and power values, we can perhaps assume that a sleep duration of more than 100ms is good enough for the CPU to transition into deep sleep state. We have omitted the power penalty for transition into various sleep states as well. The average value has been marked in the histogram.

3.2 Multi core scheduler heuristics

One of the first tunables in the kernel to tweak in an multi core, multi socket system is `/sys/devices/system/cpu/sched_mc_power_savings`. When the multi-core power saving mode [3] is enabled, the scheduler's load balancer is biased to keep workload on single physical package. This has significant impact when the number of tasks running in the system is less than the number of cores. Figure 6 plots the total idle time for each CPU for a 120 second observation interval. The system topology was a two socket dual core, with CPU0 and CPU1 sharing a package and CPU2 and CPU3 sharing the other package. Power savings can be improved if CPU0-1 are idle or CPU2-3 are idle allowing the other package to go to low power sleep state. Ebizzy is a simple cpu intensive benchmarking tool. It was simple to use and demonstrate

the effect of `sched_mc_power_savings`. Figure 7 shows that the idle time on first package has improved by consolidating the workload on CPU2 and CPU3. Ebizzy[2] was run with two threads for a duration of 120 seconds. There is a power savings of 5.4% by enabling `sched_mc_power_savings` for such cpu intensive tasks where the number of threads are less than the total number of cores available in the system.

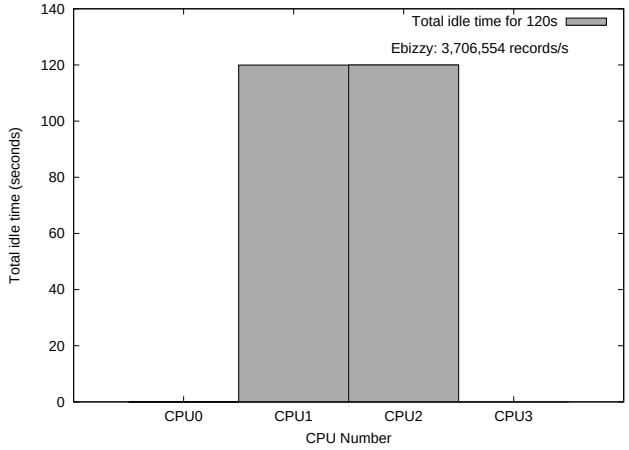

Figure 6: ebizzy with sched_mc_power_savings=0

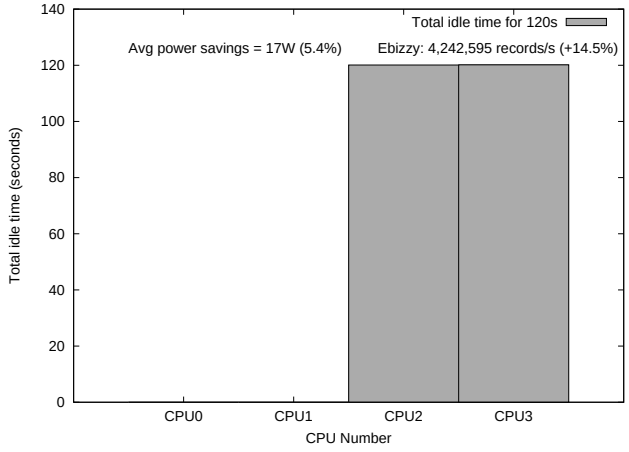

Figure 7: ebizzy with sched_mc_power_savings=1

However, the tunable will bias the scheduler loadbalancer and will not explicitly move tasks to different package. The impact is that short running daemon jobs that wakeup on various CPUs in the system will finish execution before the loadbalancer is invoked or a CPU load imbalance is detected. Hence they will continue to wakeup idle CPUs in the system.

The ineffectiveness of `sched_mc_power_savings` for short running jobs can be observed in case of kernel compilation (kernbench) workload. Figure 8 shows the idle time for `make-j2` on the same box used in the ebizzy experiment. When `sched_mc_power_savings` is enabled as shown in Figure 9, there is not much variation in the total idle time across all CPUs and hence there is no influence in the power value. The effectiveness of the heuristics is workload dependent. Kernel compilation consist of large number of short running jobs and high rate of process creation and exit as compared to pure CPU burn type workload. The variation in characteristics is mainly due to the mix of IO operations.

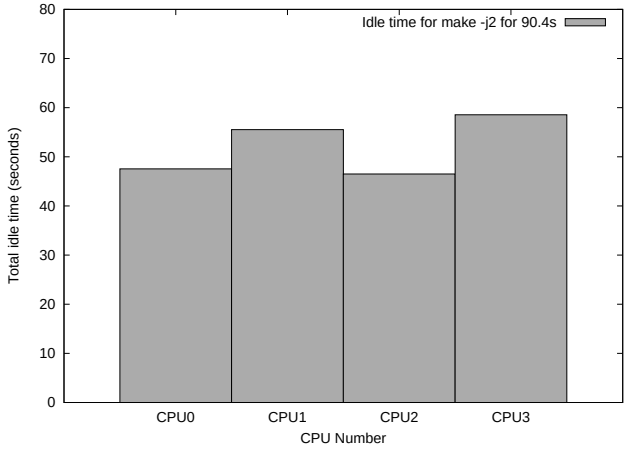

Figure 8: make -j2 with sched_mc_power_savings=0

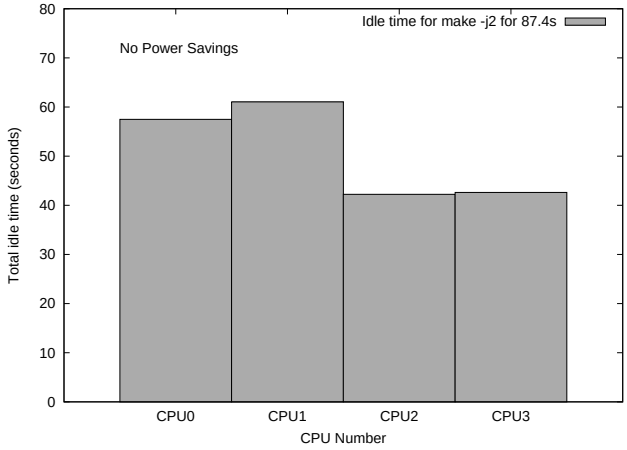

Figure 9: make -j2 with sched_mc_power_savings=1

3.3 Interrupt Migration

Interrupts in the system can be routed to one or many CPU cores in an SMP system. Default kernel routing of interrupts is to broadcast to all available CPU if that is supported by the hardware or else the interrupts stay

[2] ebizzy -t 2 -s 4096 -S 120

with the boot-up CPU or logical CPU0. There are in-kernel interrupt balancing thread and user space solutions available. The user space solution to manage interrupt routing in an SMP system is the *irqbalance daemon* [16] which is already included by most distros. The latest version 0.55 of the daemon includes power management feature where interrupts will be consolidated to CPU0 at very low interrupt rate. Once the system activity increases, the interrupt rate will increase and then the daemon will re-calculate the load and distribute the interrupts among different CPUs appropriately. The irqbalance daemon takes into account the system topology, threads, cores and packages while making the interrupt routing decision. The class of interrupt is also considered since cost of migrating ethernet interrupts is higher than that of storage interrupts.

Why does interrupt routing matter for power savings? Any CPU in its low power sleep state can be woken-up by an interrupt. Interrupts are critical for system operation and they cannot be avoided. If the interrupts are distributed to all CPUs, then even at low interrupt rate (idle system), more CPUs in the system need to wake-up for a short duration and process the interrupt. Effectively the low power sleep time of the CPU is reduced. In the case of SMP system where many CPUs are available, the interrupts can be routed to one of the CPU package or core. This leads to only one CPU package in the system to paying the power penalty for wake-up while rest of the CPU packages in the system can continue to be in low power sleep state for longer time.

Interrupt routing can be modified by writing the bitmask corresponding to the destination CPU to `/proc/irq/<irq_nr>/smp_affinity`.

`echo2>/proc/irq/214/smp_affinity` would route eth0 interrupt to CPU1 in the experimental setup described in Section 3.1, Table 1.

The user space irqbalance daemon controlls interrupt routing by writing appropriate bitmask to `/proc/irq/<irq_nr>/smp_affinity`. More than one bit can be set in the bitmask which enables the hardware to distribute the interrupt to the subset of CPUs if such interrupt broadcasting is supported by the platform and chipset.

Interrupt migration helps to improve CPU sleep time on some of the CPUs in an SMP system, but timers queued by the device drivers and applications are not affected since timers are triggered when the CPU receives an interrupt from the per-cpu tick-device. Timers queued on various CPUs in an SMP system significantly contribute to CPU wake-up from deep sleep state.

3.4 Timer Migration

3.4.1 Timers—an Introduction

Device drivers and other subsystems keep a sense of time in the kernel by means of timers. The kernel provides APIs such as `add_timer()`, `mod_timer()`, `add_timer_on()` that allow the subsystems to add or modify a timer to expire sometime in the future. With the introduction of High Resolution timer infrastructure, users can now opt for timers with finer granularity should they need it. The APIs present in the Linux kernel for the High-Resolution timers are `hrtimer_start()` and `hrtimer_forward()`.

When a timer expires, the timer subsystem will call the associated handler function which will perform the required task.

At the time this paper was written, the Non-High-Resolution timers in the Linux kernel can be classified into two types:

1. **Non-Deferrable Timers**: These are normal timers which expire when a specified amount of time elapses
2. **Deferrable Timers**: These are timers, which on a busy system behave the same way as a normal timer. But on an idle system they can be ignored while determining the next timer event. Thus they will expire when the next non-deferrable timer on the idle CPU expires. [9]

Most of these timers are initialized and queued for the first time from the task context. However, the handler function gets called from the softirq context. The requeuing of a timer can occur from the softirq context or the task context. It is easy to observe that currently, the timers which get requeued from the softirq context will be pinned to the CPU where they had been first initialized. However, those timers which are queued from the task context can migrate as the tasks queueing them get migrated. Thus the timer distribution on an SMP system is currently dependent on which CPU did the timer initialization happen and the load balancing.

3.4.2 Effect of timers on Idle CPU

On an idle CPU which is in a NO_HZ state, we program the timer hardware to interrupt the idle cpu to coincide with the nearest non-deferrable timer expiry time. Thus if there are device drivers which had initialized timers on a CPU which is now mostly idle, we would nevertheless have to wake up the idle cpu to service this timer.

This highlights the importance of consolidation of timers onto a fewer number of CPUs. Migration of these timers in a idle system is possible. As of now, the timers are migrated during CPU *offline* operation.

However CPU hotplug for the sake of idle system power management is too heavy. We will need to implement light weight timer migration framework that can be invoked in a idle system for power management purposes.

Typical distribution of timers in a distro[3] at idle in a 120 second duration is detailed in Tables 2 and 3. These results were obtained by instrumenting the timer code `__next_timer_interrupt` in `kernel/timer.c` and post processing the trace data [14].

As mentioned earlier, timers can re-queue themself in task context or softirq context. Table 2 details the list of timers that were queued in softirq context. They are generally stuck to the same CPU until forcefully moved or the application or device driver removes the timer. Table 3 details the timers that were queued in task context. These timers will generally be queued on the CPU where the corresponding task has run. The idea behind this data is to assess the percentage of timers that can be consolidated by just moving or biasing the tasks. From the data it can be observed that almost half of the timers are from task context and they can be moved by moving the task which is much easier than migrating the timer. Migrating the timer may need notification and opportunity for the task to cancel the timer all together.

3.5 Workload Migration and Consolidation

As mentioned in Section 2, the load consolidation algorithm in the current scheduler relies on accurate cpu load, i.e., the number of tasks on a cpu as an input parameter. This value is updated by sampling periodically. Since, every task running on a system need not be CPU intensive, it is possible that techniques such as `sched_mc_power_savings` fail to capture the characteristics of such tasks when it comes to workload consolidation.

To prove this, consider an experiment where we have a *cpuset A*, which has a bash shell as a member, that runs `make-j2` of a Linux kernel. The experiment is run on a 2 socket dual core machine. The logical CPUs 0 and 1 are core siblings in the first socket and logical CPUs 2 and 3 are the core siblings in the other socket. We vary the number of CPUs allocated to the `cpuset` by writing different values to `cpuset.cpus` file. The time taken to complete the `make`, the avg power consumed (normalised value) during this interval, and the utilization of the individual CPUs in the system are recorded. During the experiment, `sched_mc_power_savings` is set to 1 and the `cpufreq` governor is set to `ondemand`. Table 4 details the result of this experiment.

Ideally, one would expect that running the job with only two cpus would yield the same results as running the job with all the four cpus with `sched_mc_power_savings` enabled. However, from the experiment, we observe that `sched_mc_power_savings` does not seem to have much effect when we run with all the four CPUs. There are only two active tasks in the system but they get distributed across all the CPUs.

In the experiment, the energy consumed for the case with only two CPUs is:

$$\begin{aligned} E_2 &= 71.751 \times 1.045x \\ &= 74.980xJ \end{aligned} \tag{1}$$

Energy consumed for the case with all the four CPUs is:

$$\begin{aligned} E_4 &= 85.185 \times 0.941x \\ &= 80.159xJ \end{aligned} \tag{2}$$

Thus, additional amount of energy spent would be:

$$\begin{aligned} E_{extra} &= E_4 - E_2 \\ &= 80.159x - 74.980x \\ &= 5.179xJ \end{aligned} \tag{3}$$

$$\begin{aligned} E_{extra}\% &= \frac{E_{extra}}{E_2} \times 100 \\ &= 6.91\% \end{aligned} \tag{4}$$

From Table 4 we can also observe that the time taken to complete the job is higher when all the 4 CPUs were

[3] Fedora 9 beta

Function_Name	CPU0	CPU1	CPU2	CPU3	Total
rh_timer_func			483		483
delayed_work_timer_fn	62	59	60	71	252
bnx2_timer		119			119
neigh_periodic_timer	30		60		90
dev_watchdog		48			48
process_timeout	32		1	5	38
wb_timer_fn	24				24
peer_check_expire	4				4
neigh_timer_handler			4		4
hangcheck_fire	3				3
commit_timeout	1			2	3
addrconf_verify			3		3
Total	156	226	611	78	1071

Table 2: Timer in SOFTIRQ context at idle for 120s

Function_Name	CPU0	CPU1	CPU2	CPU3	Total
hrtick	159	132	107	81	479
delayed_work_timer_fn	62	60	60	72	254
ide_timer_expiry	34	30	33	29	126
scsi_times_out	40	1	3	1	45
process_timeout	31		2	6	39
wb_timer_fn	24				24
blk_unplug_timeout	19	1	3	1	24
hrtimer_wakeup	2				2
commit_timeout	2				2
tcp_write_timer		1			1
it_real_fn	1				1
Total	374	225	208	190	997

Table 3: Timer in task context at idle for 120s

Experiment 'make -j2' of linux-2.6.25-rc7			
CPUs allocated	**Time taken**	**Power Consumed**	**% Utilization of the CPUs**
0	120.678 s	1.000x W	99, 02, 00, 00
0-1	71.751 s	1.045x W	83, 89, 01, 01
0-3	85.185 s	0.941x W	34, 33, 59, 57

Table 4: *'make -j2'* with varying number of cpus

used, when compared to the case where only 2 CPUs were used. Since the ondemand governor changes the processor frequency based on the processor utilization, when 2 threads were bouncing across all 4 processors, the system utilization was not high enough to increase the frequency to the maximum, and hence it took longer time. However, in the case of allocating just 2 processors, we note that the utilization is sufficiently high for the ondemand governor to run the job at the maximum frequency, thus finishing it faster.

Thus we observe that by allocating more processors than what is required, we're not only degrading the power savings, but also the performance in this case. Hence there is scope for power savings by improving the power aware task/workload consolidation in an under utilised system.

One of the possible solutions could be to consider the following parameters during scheduler load balancing or consolidation decision apart from just counting the number of waiting tasks:

- The nature of each task, whether it is CPU intensive or IO bound
- The overall utilization of the system.
- Any hints from the tasks themselves

Using some of these parameters, it is also possible to bias the wake up of a task onto a non-idle CPU, thereby avoiding waking up an idle CPU when the number of tasks on a particular runqueue is nonzero.

4 Sleep states

Coming to the core of the issue, why do we want the CPUs to be idle for long duration. Modern processors supports multiple idle states that vary from high power low latency idle states to low power high latency idle states.

With CPUs being idle for extended period, they can be put into low power high latency idle state, conserving significant power in the process. On the other end, frequently waking up CPUs cannot use deepest idle state, as if they do, they end up paying significant overhead due to higher transition latency in and out of the deepest idle state.

Other factors to keep note of with regard to idle CPUs are:

- Most of the current generation CPUs control the CPU voltage at the socket level. This means, if some cores in a socket are idle and other cores are busy, idle cores may not be at optimal power state due to the higher voltage on the socket leading to higher leakage power.
- Most of the current generation multi-core CPUs have some shared resources across all the cores, like last level cache. This shared resource will not be able to go to low power state unless all cores in the socket are idle.

This means it is important to keep as many cores and as many sockets in idle state as long as possible. That helps the CPUs to be at the most optimal power state.

The current Linux kernel has cpuidle governor that takes care of entering the right idle state based on the individual CPU activity and requirements. The scheduler power savings tunable takes care of keeping the entire socket idle in case of long running tasks. Newer versions of irqbalance take care of routing interrupts to one CPU while the system is relatively idle.

The things that are missing include:

- power-aware scheduling for short-running tasks
- smart routing of timers interrupts in the idle scenarios, and
- making the CPU latency requirements per CPU (instead of system-wide) so that processes and interrupts with critical latency requirements continue to have good response time in partially idle case, with other idle cores being in deepest idle state.

5 Conclusion

Every watt saved is a watt that doesn't have to be generated! Doing more computation with less power helps the environment and every power management feature makes the world greener. System power management helps to improve compute density and manage power as a resource by matching the power consumption to workload just as in an automobile.

The Linux kernel already takes advantage of various power management features available in the platform, however there is still scope for improvement as new platform features will become available in future.

6 Acknowledgments

We owe thanks to Ingo Molnar, Arjan van de Ven and members of linux-pm and lesswatts-discuss for their inputs on Linux kernel issues. We are also indebted to Patricia Gaughen, Vani S. Kulkarni, Sudarshan Rao, and Premalatha M. Nair for their support of this effort.

7 Legal Statement

References

[1] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power cmos digital design. *JSSC*, 27:473–484, 1992.

[2] Standard Performance Evaluation Corporation. `SPECpower_ssj2008`. `http://www.spec.org/power_ssj2008/`.

[3] Suresh B Siddha et al. Chip multi processing aware linux kernel scheduler. `http://www.linuxsymposium.org/2005/linuxsymposium_procv2.pdf`.

[4] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25, New York, NY, USA, 1995. ACM.

[5] Intel Technology Journal. Power and thermal management in the intel core duo processor. `http://download.intel.com/technology/itj/2006/volume10issue02/vol10_art03.pdf`.

[6] Jacob R. Lorch and Alan Jay Smith. Reducing processor power consumption by improving processor time management in a single-user operating system. In *MobiCom '96: Proceedings of the 2nd annual international conference on Mobile computing and networking*, pages 143–154, New York, NY, USA, 1996. ACM.

[7] LWN. Tickless, acpi and thermal management. `http://lwn.net/Articles/240253/`.

[8] Ingo Molnar. `CFS` scheduler. `http://kerneltrap.org/node/8059`.

[9] Venkatesh Pallipadi. Deferrable timers. `http://lwn.net/Articles/228143/`.

[10] Venkatesh Pallipadi and Suresh B Siddha. Processor power management fatures and process scheduler: Do we need to tie them together. `http://www.linuxconf.eu/2007/papers/Pallipadi.pdf`.

[11] Venkatesh Pallipadi and Alexiy Starikovsky. The ondemand governor - past, present and future. In *Proceedings of the Linux Symposium*, volume 2, Ottawa, ON, Canada, 2006. Linux Symposium.

[12] Suresh Siddha. Getting maximum mileage out of tickless. `http://ols.108.redhat.com/2007/Reprints/siddha-Reprint.pdf`.

[13] Vaidyanathan Srinivasan. Instrumenting idle time. `http://lkml.org/lkml/2008/1/8/243`.

[14] Vaidyanathan Srinivasan. Instrumenting timers at idle. `http://lkml.org/lkml/2008/3/2/109`.

[15] Jinwoo Suh, Dong-In Kang, and Stephen P. Crago. Dynamic power management of multiprocessor systems. *ipdps*, 2:0097b, 2002.

[16] Arjan van de Ven. Irqbalance - handholding your interrupts for power and performance. `http://irqbalance.org`.

[17] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 2, Berkeley, CA, USA, 1994. USENIX Association.

[18] Fan Zhang and Samuel T. Chanson. Power-aware processor scheduling under average delay constraints. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 202–212, Washington, DC, USA, 2005. IEEE Computer Society.