Cpuidle from user space

Madhu Palmur, Zhichao Li, and Erez Zadok
FSL Technical Report FSL-13-05

Abstract

In this paper we present a user space cpuidle governor. In addition to providing a user space interface to pick idle states for individual cores in a multicore system, the governor also ensures that each core stays in the specified idle state forever. In other words, the cores do not wake up from a specified idle state unless specified by the user. This gives a user complete control over a core's idle states without worrying about any kind of wake-ups.

A user space governor can be very useful in scenarios where every workload is run with a customized cpuidle power saving algorithm. Coding different algorithms and dynamically switching between them is fairly simpler in user space when compared to kernel space. From our evaluation results, we have concluded that this technique does not hurt power consumption savings or performance benefits in any way.

1 Introduction

Cpuidle is a module in the Linux kernel which is responsible for running some power saving routines on a core when the core does not have any task in its run queue [3]. The power saving routines try to put the core into a low power state or an *idle* state. Processors in the market today offer various idle states with different power consumption savings and wakeup latency overheads.

All machines with i386 and x86_64 architecture follow an open standard for power management called ACPI which stands for Advanced Configuration and Power Interface. In ACPI terminology, the idle states are called C-states. Linux code for platform specific ACPI functionalities can be found in the directory drivers/acpi [4]. Among many other things, this driver is mainly responsible for implementing hardware specific instructions to make the actual transition of a core into and out of a specified idle state.

The idle states come with a trade-off. Every idle state has an enter and exit latency associated with it. These latencies are the time required for a core to enter the corresponding idle state and time required for a core to exit from the corresponding idle state respectively. The trade-off is, as and when the idle states get deeper, the power consumption savings increase but the enter-exit latencies increase as well. For example, An Intel Xeon 5,600 series processor which consumes 80W of power when it is up, consumes 40W in C1 state, 33W in C3 state and, only 12W in C6 state. Also, On an x86 machine, C1, C3, and C6's exit latencies are 3, 20 and 200 microseconds, respectively [1]. High enter-exit latencies can have negative effects on the performance of the system. Hence, it is essential for the cpuidle module to pick the most appropriate idle state at any given time.

2 Background

As mentioned before, the Linux kernel has the cpuidle module which implements the entire infrastructure of picking an appropriate idle state, transitioning cores into those idle states and waking them up when required. The cpuidle module has the following components:

- The sysfs interface
- The cpuidle governors
- The core cpuidle logic
- The platform specific driver functions

The sysfs interface exposes several read only and read write interfaces. Some of the read-only interfaces include `current_driver`, `current_governor_ro` for the governor algorithm and driver type that has been selected. For every core read only information like power consumed while in a particular idle state, latency to exit out of a particular idle state is also exposed. The sysfs has some writable interfaces like `current_driver` and `current_governor` so that current driver and governor selections can be changed dynamically.

Cpuidle governors implement the main algorithm which core has to enter which idle state at any given time. Two of the traditional governors in the Linux kernel are the menu and the ladder governors. The ladder governor initially starts off with the lowest possible idle state for a core and increments the idle state if the core can afford the latency overheads of the new idle state. The menu governor improvises on this algorithm by not starting off with the lowest possible idle state but by picking the best idle state a core can afford to transition into at any given point of time. The menu governor considers metrics like energy break even point, average load on the core into consideration, performance

requirements, latency requirements, etc. while picking the best possible idle state for a core.

The core cpuidle logic is called from the idle thread for every core. It is responsible for enabling the idle infrastructure for all online cores. Once enabled, it makes a call to the selected governor algorithm to get the next desired idle state and then makes a call to the selected driver function to actually transition the core into the specified idle state.

The platform specific idle functions are hooked into the main cpuidle logic. They implement architecture dependent functionalities like transitioning the cores into a selected idle state, waking up the core when certain conditions are met, etc.

3 Related work

The Ladder governor was the first governor algorithm written for the cpuidle module. The algorithm always starts off with the lowest possible idle state. When the processor is idle again for the next time, it checks for the QOS requirements of the system and jumps into the next higher idle state. When an idle state no longer meet the QOS requirements of the system, the ladder governor jumps to the immediate lower idle state. This has proven to work well with tick-based kernels. However, this step-wise approach does not work well with tickless kernels because the kernel can be idle for a really long time without the periodic tick and not get a chance to step into a deeper idle state whenever it goes idle.

The Menu governor was written as an improvement over the ladder governor algorithm. The Menu governor takes into account three factors before deciding a C-state: energy break even point, performance impact, and latency tolerance. The Menu governor implements a naïve prediction mechanism to speculate how long a core is going to be in a particular idle state so that C state entry and exit energy cost breaks even with the energy that is saved when the core actually resides in a particular idle state. The Menu governor follows a heuristic of picking a C-state that has the least impact if the system is busy. However, the Menu governor is said to not perform well with short lived tasks.

4 Motivation

The cpuidle governor algorithms currently run from the kernel space. The present governor algorithms are generic, they do not offer any kind of customizations depending on the workload. Not just that, implementing a new customized algorithm for every type of workload in the kernel space is hard. Every time a new governor algorithm is written, a new patch has to be written and applied to the kernel, the kernel config file has to be changed to run the new governor algorithm, and the kernel has to be recompiled and installed again. This process is time consuming. Also, writing kernel code is any day harder than writing programs in user space. Building a tool which exports all the idle state decision making infrastructure to user space can hence turn out to be useful.

The main goals of this project are:

1. Build a kernel tool that exports the whole idle state decision making infrastructure to the user space.
2. The tool should make sure that users can specify core specific idle state value. It should also make sure, that a core actually stays in the specified idle state until the user changes it.
3. Run a workload and prove that even though the idle state decisions are made in the user space, the performance benefits and power consumption savings are at least as compared to the existing kernel space governor algorithms.

5 Design

The goal is to build a system which gives the user complete control over the idle states of the cores. The traditional cpuidle governors have their own algorithm for picking the next c-state a core should transition into. Initially, we wrote a custom governor which does nothing but look at the value which indicates what c-state the user is requesting for and set the next c-state value for that core to this value. We called that custom governor the “Noop” governor.

The current design of the cpuidle module is such that even though the core gets transitioned into a c-state, it is woken up as soon as there are tasks or IRQs that have to be scheduled on the core. To give a clearer picture, the current flow of the idle thread is something as follows: [2]

1. The idle thread gets scheduled when there are no other tasks present in the run queue of the core.
2. The idle thread runs an infinite loop. It calls the cpuidle module inside this loop.
3. The cpuidle module in turn calls its governor to select the next c-state the core should transition into.
4. The cpuidle module then transitions the core into that c-state.
5. The idle thread yields itself when there are other tasks present in the run queue of the core.

Given this, it is obvious that even though we move a core into user requested c-state using our custom governor, it is woken up time and again to run tasks and IRQs assigned to it. This means that there can be a situation

wherein even though the user requested a deeper c-state for a particular core, it can move back to C0 temporarily to empty its run queue. This is well against our goal of giving the user complete control over the idle states of cores. Hence, just using the noop governor will not meet our desired goals.

It is not so simple to halt a core completely without waking it up for a long time. This is because the Linux scheduler can pick any online core to run a particular task or an IRQ. Also, each core has been destined to service some specific set of IRQs. The core should also keep emptying its run queue as and when the ready tasks arrive. Not waking up the core can make the system completely unresponsive.

We utilized the CPU hot-plugging infrastructure [5] to achieve our goal of keeping a core in the user requested c-state and never waking it up. The CPU hot-plug infrastructure can be used to make a core completely invisible to the OS so that the scheduler can never try to schedule any task on that core. The CPU hot-plugging infrastructure essentially migrates off all tasks, interrupts, timers, and tasklets from the victim core to the other online cores. The CPU hot-plugging code makes use of the monitor, MWAIT instruction pair to halt the cores. By picking the appropriate argument to be passed to the MWAIT instruction, the core can be moved to any desire idle state.

We invoke this infrastructure whenever a user requests a c-state greater than C0 for a core. We still keep a custom governor running for the rest of the cores which are in C0 state. These cores are in C0 state either because the user has selected C0 state for them or user never requested for a different c-state for these cores. The governor just selects C0 state for these cores for which the user has selected no other state other than C0.

6 Implementation

The implementation can be broken down to three parts. The three parts are:

1. Change the sysfs code to accept core specific user input
2. Hook the cpu hot-plug infrastructure to the sysfs store operation
3. Write a custom kernel which ensures that the core in C0 state always stay in C0 state until the user requests a different one

We have added a new file under the sysfs directory to take the c-state value as input from the user: `/sys/devices/system/cpu/cpuX/cpuidle/c_state`. The new file `c_state` is present under the cpuidle directory of every core. To remember the current c_state for every core, we have added a new integer field called `c_state` in the structure `cpuidle_device`.

We use the cpu hot-plug API cpu_down() to bring a core down to any c-state. In the store function of the sysfs c_state file described in the previous section, we make a call to this cpu_down() function when the c_state requested is anything greater than C0. Internally, this cpu_down() API makes a call to an architecture-specific function which calls monitor, MWAIT instructions pair to move the core into a particular idle state. In this function, we add an extra piece of code which checks `c_state` field of the cpuidle_device structure and passes appropriate argument to the `mwait()` call. This argument determines which idle state the core will enter into. Also, we use the CPU hot-plug API, cpu_up() to bring the core back up to C0 state in case the user requests a C0 state for a core which was not already in C0 state. A point to note is that to move a core from one c-state to another c-state, it should first be brought online by using cpu_up() API. Figure 1 explains this.

The third part of the implementation is to keep a custom governor which will not interfere with any of the core's idle states. Note that the cores for which the idle state selected is something other than C0 will not even reach this part of the code. This is because we halt that core by running cpu_down() immediately after the user requests for a c-state change. So, we do not give a chance for the governor to even run on that core after the c-state change. The cores for which either the user has not requested for any particular c_state or the user has specifically requested for C0 state, we need to keep them in C0 state. The custom governor always selects C0 state for such cores.

7 Evaluation

We measure the following two major metrics. The user space governor should not perform worse than the kernel space governor w.r.t., the following metrics at any cost.

1. Performance
2. Power consumption savings

We have evaluated our tool with a completely CPU bound workload. The workload forks some specified number of processes which continuously perform random mathematical floating point computations. We wrote a simple bash script that writes the value of the desired C-state into the c-state file of the sysfs interface. The bash script runs the workload by specifying a specified number of processes to be forked and keeps those many number of cores active. It puts the rest of the cores to the deepest possible C-state. We optimistically picked the deepest possible C-state to save more energy with the

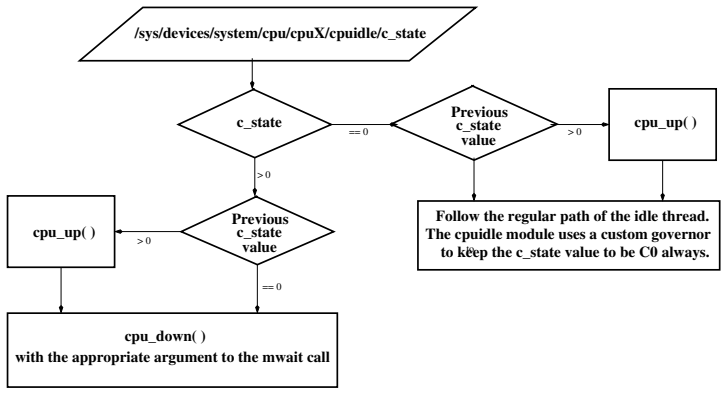

Figure 1: Implementation

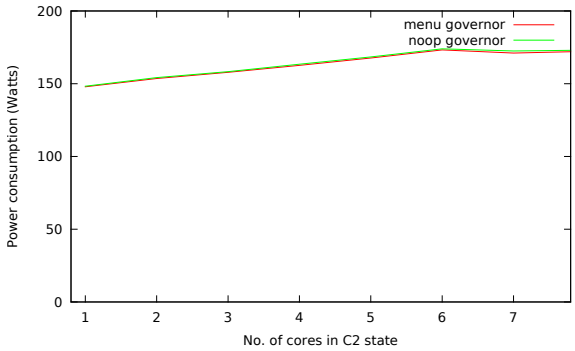

Figure 2: Comparing Power Consumption savings of Noop governor with Menu governor for CPU bound workload

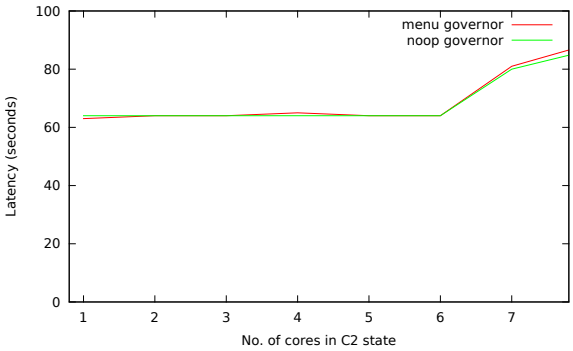

Figure 3: Performance comparison of Noop governor with Menu governor for CPU bound workload

intention of lowering the C-state if performance took a hit.

All experiments are carried out on a server containing Intel Xeon 5,650 processor. There are 6 cores in the system in total (with hyper threading disabled). This system supports three C states: C0, C1 and C2. In all our experiments we have used the C2 state whenever we want to move a core into an idle state.

Figure 2 shows the power consumption savings of our user space governor when compared with the menu governor. We called our user space governor the noop governor because the governor does not implement any power saving algorithm on its own but relies on the user to do it. As it can be seen, the power consumption savings of the noop governor are almost similar to menu governor. The result is pretty intuitive because the workload is completely CPU bound and keeps the CPU almost 100% busy all the time. Since the cores that are not put in any idle state in our case all remain active even when the menu governor is run and the cores put in idle state remain in the idle state in both the cases, menu governor does not save any extra energy.

Figure 3 shows the latencies of our user space governor when compared to menu governor. The term latency here refers to the total time taken by the system while it waits for the specified number of forked processes to complete their mathematical computations and exit. As it can be seen from the figure, the latency results of the noop governor is comparable with that of menu governor's. Since the number of online cores is always equal to the number of processes forked, the result is pretty straight forward.

We also ran our user space governor with a workload which has intermittent I/Os. The workload just does Linux kernel compilation by spawning a specified number of threads. The command `make -jX` spawns X number of threads to finish the kernel compilation. We used the same bash script which starts the workload with a specified number of threads and keeps exactly those many number of cores active and the rest are put into the deepest possible idle state. What we observed was the latency graph of menu and noop governor were almost similar but the power consumption graph was drastically different.

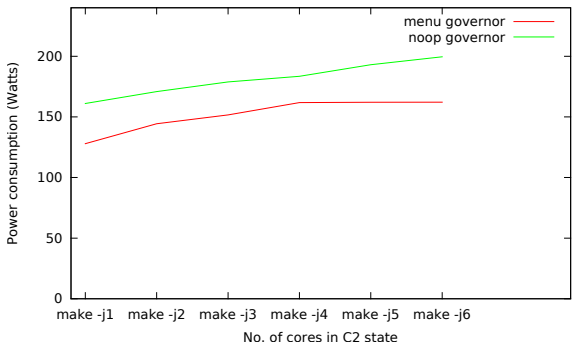

Figure 4: Comparing Power Consumption savings of Noop governor with Menu governor for a workload running "make -jX"

Figure 4 shows the power consumption savings of our user space governor when compared with the menu governor. As it can be seen from the graph, we perform worse than the menu governor in all cases. The reason for this is, the workload has intermittent I/Os. Consider an example where we have spawned just one thread and we have kept just one core active to run this thread and put the rest of them into idle states. In this cases, when the thread performs its intermittent I/Os, the core has no work to do and runs its idle thread. This is again a great window of opportunity to save power. We unfortunately keep our core active all the time without putting it into any idle state when compared to the menu governor. Also, the change in the power consumption savings is quite a lot when compared to menu governor because of the following reason. When the one core that is active goes to an idle state due to an I/O, effectively all cores are in an idle state. We observed that maximum power consumption savings are incurred in a system when all the shared resources like L3 cache of the cores are switched off too. Shared resources can only be switched off when all the cores are in idle states. This limitation can be overcome by writing a more intelligent user space program that learns from the workload. The program should be able to study the workload initially, and start predicting as to when I/Os happen and take appropriate action on the cores.

8 Future Work

As mentioned in the previous section, even though we have the tool ready to run governor algorithms from user space and we did show that the user space governor can perform at least as good as the kernel space governor, we still do not have an algorithm ready for every type of I/O bound workloads. A simple HMM (Hidden Markov Model) type of model can be used to first learn about the I/O patterns from the given workload and then start predicting as to when I/O actually happen so that the window can be used to save power. When run from user space, we believe this should perform at least as good as the menu governor or even better because the menu governor presently does not use such a sophisticated prediction mechanism.

9 Conclusion

In conclusion we have built a very useful tool that exports the whole idle state decision making infrastructure to the user space. This should give a lot of flexibility to the users to use customized algorithms from the user space for every kind of workload as opposed to using a generalized algorithm from the kernel space. Needless to say, writing simple programs from the user space is much easier when compared to writing governor algorithms inside the kernel. Dynamic switching between algorithms and addition of new algorithms can be very easily done from the user space as compared to the kernel space. We have proved by running a completely CPU bound workload that running governor algorithms from user space can be at least as good as running it from the kernel space. We also point out the future work of writing sophisticated algorithms for I/O bound workloads by using a very good prediction mechanism to predict I/O patterns which has the potential of performing better than the existing kernel space algorithms.

References

[1] Intel Inc. Intel xeon processor 5600 series. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-5600-vol-2-datasheet.html`, 2011.

[2] Linux 3.2.9. `http://www.kernel.org`.

[3] S. Li and A. Belay. cpuidle — Do nothing, efficiently... In *Proceedings of the Linux Symposium*, volume 2, 2007.

[4] Acpi in linux. `http://acpi.sourceforge.net`.

[5] Zwane Mwaikambo, Ashok Raj, Rusty Russell, and Joel Schopp. Linux Kernel Hotplug CPU Support. *Proceedings of the Linux Symposium*, 2, July 2004.