Powernightmares: The Challenge of Efficiently Using Sleep States on Multi-core Systems

Thomas Ilsche[1(✉)], Marcus Hähnel[2], Robert Schöne[1], Mario Bielert[1], and Daniel Hackenberg[1]

[1] Center for Information Services and High Performance Computing (ZIH), Technische Universität Dresden, 01062 Dresden, Germany
{thomas.ilsche,robert.schoene,mario.bielert, daniel.hackenberg}@tu-dresden.de
[2] Operating Systems Group, Technische Universität Dresden, 01062 Dresden, Germany
marcus.haehnel@tu-dresden.de

Abstract. Sleep states are an important and well-understood feature of modern server and desktop CPUs that enable significant power savings during idle and partial load scenarios. Making proper decisions about how to use this feature remains a major challenge for operating systems since it requires a trade-off between potential energy-savings and performance penalties for long and short phases of inactivity, respectively. In this paper we analyze the default behavior of the Linux kernel in this regard and identify weaknesses of certain default assumptions. We derive pathological patterns that trigger these weaknesses and lead to 'Powernightmares' during which power-saving sleep states are used insufficiently. Our analysis of a workstation and a large supercomputer reveals that these scenarios are relevant on real-life systems in default configuration. We present a methodology to analyze these effects in detail despite their inherent nature of being hardly observable. Finally, we present a concept to mitigate these problems and reclaim lost power saving opportunities.

Keywords: Linux · Sleep state · Energy efficiency
Power consumption

1 Introduction

As energy is one of the major cost factors in data-center operations, CPU developers are constantly pushing towards more aggressive techniques to scale power consumption with system load. One aspect to achieve this so called *power proportionality* is the reduction of power consumption during idle phases. These phases represent a major fraction of the run-time of desktop systems and are also noteworthy in the server domain. How deep a CPU sleeps determines how much power it consumes, but also how long it takes to wake up from its slumber.

D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 623–635, 2018.
https://doi.org/10.1007/978-3-319-75178-8_50

For example, depending on the depth of the sleep a typical Haswell server may consume 73 W in idle and wake up within 25 µs or consume 126 W and wake up within 2 µs [3]. The challenge for an operating system (OS) is to ensure that as much time as possible is spent sleeping as deeply as possible, while satisfying the latency requirements of the system. The job of the idle governor is to strike this balance. When investigating unexpected high power usage of one of our test systems, we found that sometimes the default governor in Linux does not let the system sleep as deeply as desirable for prolonged idle phases—it caused a *Powernightmare*. The same effect was found in our petascale production HPC system. We traced the cause of this inefficiency, and developed a mitigation that wakes the system from its nightmare and lets it sleep well again.

The remainder of this work is structured as follows: We explain the details of sleep states of modern CPUs and their use by the OS in Sect. 2, followed by a description of the problem and when it occurs in the wild in Sect. 3. We discuss possible solutions and describe and evaluate our mitigation approach in Sect. 4 before we conclude and give an outlook on future work in Sect. 5.

2 Background and Related Work

The ACPI standard [1] describes different power saving mechanisms. This includes P-states, which are implemented via voltage and frequency scaling (DVFS), T-states, which are implemented via clock modulation, and C-states that are typically implemented via clock gating [15, Sect. 5.2.1.1] and power gating [15, Sect. 5.3.2]. The four different C-states *C0* to *C3* are distinguished by ACPI. Higher C-state numbers refer to deeper sleep states with lower power consumption and longer wake-up latencies.

2.1 Hardware Perspective on C-States

Contemporary Intel server CPUs implement C-states per core and per package, referred to as *CC*-states and *PC*-states, respectively. Only the former can be directly influenced by the OS, while the latter are enabled by hardware under specific circumstances. The CC-state is the lowest of the selected C-states among all hardware threads on the core. Similarly, the PC-state is determined by the lowest CC-state of all cores incorporated on the package [6,7, Sect. 4.2.5].

Modern Intel server CPUs implement at least four CC-states: *CC0*, *CC1*, *CC3*, and *CC6* [6,7, Sect. 4.2.4]. The processor core is active and executes instructions in CC0. In CC1 the processor core is still active and caches are not flushed. The additional *C1E* does not differ from CC1 for the core itself, but allows the package to enter *PC1E*, if all cores are in C1E or higher. In CC3, core clocks are stopped, and caches are flushed. In CC6, the core applies power gating, storing its internal state to a dedicated SRAM. The architectural state is restored when the core returns to a lower CC-state. Another feature, called *delayed deep C-states (DDCst)* is described in [7, Sect. 4.2.4.5]. Here lower CC-states are used for a short period of time before switching to higher C-states.

To the best of our knowledge, a documentation of the mechanisms of newer Intel server processors is currently not available. However, since they are handled like their predecessors and their desktop counterparts, one can assume that the general mechanisms are the same.

There are six different PC-states [7, Sect. 4.2.5]: PC0, PC1, PC1E, PC2, PC3, and PC6. Similarly to CC0, PC0 refers to the normal operation of the package. While in PC1, "No additional power reduction actions are taken" [6, Sect. 4.2.5], core voltage and frequencies are reduced in PC1E. In PC3 and PC6 the last level cache becomes inaccessible, voltages are lowered, and the power consumption of uncore components is reduced [7, Sect. 4.2.5].

Higher C-states provide a significant power saving potential, at the cost of higher exit latencies [13]. Intel describes hardware mechanisms that counter inefficient usages of C-states [14]. These mechanisms, called *promotion* and *demotion*, use hardware loops to track C-state residency history and automatically re-evaluate OS decisions. For promotion, the hardware automatically increases the C-state, for demotion it lowers the C-state. Intel desktop processors and previous server processors include a feature called *C1E auto-promotion* [6,7, Sect. 4.2.4]. There is no promotion to higher PC-states than PC1E. The processor core can perform demotion by choosing: (1) CC3 instead of the requested CC6, and (2) CC1 instead of CC6/CC3. To correct wrong decisions demotions can be reverted by a mechanism called *undemotion*. Whether promotion, demotion, and undemotion are enabled is encoded in the `PKG_CST_CONFIG_CONTROL` register. On Intel processors, C-states can be requested by the OS in the form of a hint argument to the `mwait` instruction.

2.2 Idle Power Conservation Techniques in Linux

An important feature of modern Linux systems is the so called *dyntick-idle* mode, also called *nohz* mode or *tickless*. This feature reduces the number of scheduling-clock interrupts for idle cores as opposed to having regular scheduling ticks, e.g., every 4 ms. In dyntick-idle mode, a core can remain in idle indefinitely. This is the default behavior on modern systems [9].

Whenever a core has no task to be scheduled, an idle state is selected. The `cpuidle` *governor* implements the selection policy while the `cpuidle` *driver* implements the architecture-specific mechanism to request an idle state from the CPU [10]. In our evaluation, we focus on high performance systems with Intel processors, using the `intel_idle` driver. Idle states correspond to C-states. Linux currently provides two governors to select idle states.

The `ladder` governor evaluates on each call whether the previous C-state was predicted correctly and increases or decreases the depth stepwise. Pallipadi et al. [10, Sect. 4.1] describe that while "this works fine with periodic tick-based kernels, this step-wise model will not work very well with tickless kernels".

The `menu` governor, which is the default on tickless Linux systems, combines several factors as a heuristic. It uses an *energy break even point* based on the `target_residency` provided by the architecture specific `cpuidle` driver. The challenge is to predict the upcoming idle duration.

The prediction algorithm starts with the known *next timer event,* and applies a correction factor based on an exponential moving average on how accurate this prediction was in the past. Idle times predicted to be longer than 50 ms are always considered to be perfect, on grounds that longer sleeping times provide no additional power improvement. Additionally, the *repeatable-interval-detector* records the duration of 8 previous intervals and uses their average. Up to two high values are ignored if the variance across all eight values would be too high. If the variance is still too high among the six lowest previous times, this predictor is ignored, otherwise the minimum of the next timer event and the repeatable-interval-detector is used.

Further, the `menu` governor tries to *limit the performance impact* by choosing C-states with shorter exit latencies on busy systems. Based on the load average and number of IO wait tasks, a `performance_multiplier` limits the ratio of predicted idle time and exit latency. A device or user can request a maximum DMA latency for QOS purposes (`PM_QOS_CPU_DMA_LATENCY`). The latency requirement is the minimum of both values. Finally, the `menu` governor selects the highest enabled C-state with a `target_residency` of no more than the predicted idle time and an `exit_latency` that does not exceed the latency requirement.

The heuristic relies on many values that have been determined experimentally. Since its last big change[1] in 2009, the `menu` governor has operated like that. In principle, the `ladder` governor has not changed since 2007. In our work, we focus on tickless systems running the `menu` governor, since it is essential from an energy-efficiency perspective to avoid unnecessary scheduling-clock interrupts.

Considering the impact on performance and power consumption, the `cpuidle` governor was a research target before. Roba and Baruch [11] propose two possible improvements for the C-state selection heuristic claiming a constant improvement of 10% for their combination. One approach is based on machine learning, the other tries to improve the responsiveness of the already existing repeatable-interval-detector in the `menu` governor. Kanev et al. [8] conducted a state-of-the-art study for typical datacenter applications. They argue that "the maximum improvement for both power and latency with a single policy is unlikely" and thus the `menu` governor is a compromise in-between. Given the strict latency requirements in the datacenter domain, Kanev et al resort to DVFS for power savings. However, both works focus more on improving latency instead of energy.

3 Analysis of Inconsistent Power Saving in Idle

In this section, we identify inefficient power saving decisions in Linux and demonstrate how to trigger the effect. We also show real world occurrences on an individual machine and across a production HPC system.

[1] cpuidle: fix the menu governor to boost IO performance: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=69d25870f20c4b2563304f2b79c5300dd60a067e.

Table 1. Properties of systems under test

	Testsystem *Diana*	HPC system *Taurus*
CPU	2 × Xeon E5-2690 v3	2 × Xeon E5-2680 v3
Measurement	Per socket at 500 kSa/s [4]	Per socket at 100 Sa/s
	System (AC) at 20 Sa/s	Node (DC) at 1000 Sa/s [2]
Kernel Version	4.11.0-rc8 (8b5d11e)	2.6.32 (Bull SCS4 / RHEL 6.8)
Total system power consumption in different states		
All cores C6	73.9 W (total system)	87.0 W (total node)
Core 0 C1E, others C6	106.3 W (total system)	n/a*
All cores C1E	126.1 W (total system)	130.8 W (total node)

*The old kernel does not support disabling C-states for individual cores.

3.1 Observation

During energy efficiency research on a system with sophisticated power measurement instrumentation [4], we have observed an unexpected behavior: Even though the system is specifically configured for low idle power consumption, i.e. few interrupts, the power consumption during idle phases was erratic. While the baseline idle power is 73 W, sometimes after inconspicuous activities, the total power consumption remained over 100 W for several seconds. This happened during times without any explicit activity on the system and was observed by the external power measurement. The behavior persisted across a wide range of recent and historic kernel versions. We also observed this effect on a production HPC system. The underlying issue eluded investigation for a while, in particular because any active measurement directly impacted the effect under investigation.

3.2 Experimental Platform

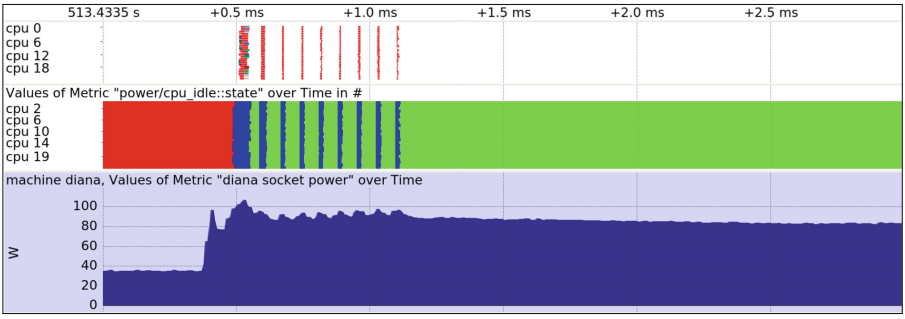
For reproduction and analysis of the effect we used two systems: a workstation for energy measurements (*Diana*) and a node of a petascale production HPC system (*Taurus*). Both are equipped with dual socket Intel Haswell-EP CPUs and sophisticated energy measurement instrumentation (see Table 1). All energy measurements are calibrated and verified to high accuracy [2,4,5]. On *Diana* we use the high resolution socket power measurements when observing small time scales. Full system (AC) measurement allows us to perform analysis at larger time scales. We fixate the core frequency with the userspace P-state governor and disable HyperThreading to reduce the variance of the measurements. Most of our observations relate to PC1E or higher, in which power consumption is not affected by core frequency.

3.3 Tools to Isolate the Effect

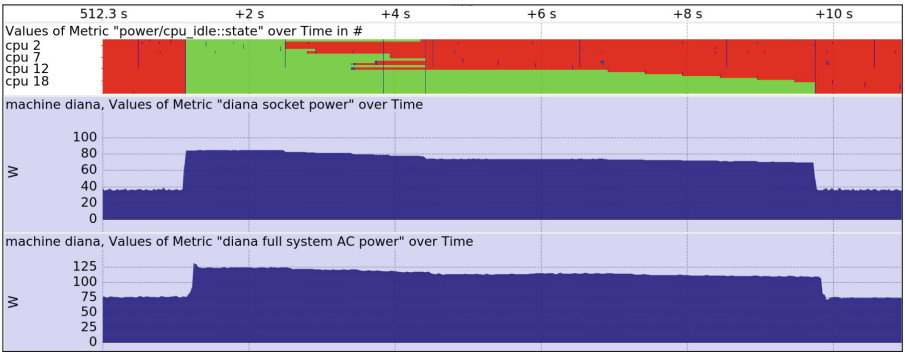
To isolate the effect, we combined existing and newly implemented kernel tracepoints, measurement of C-state residencies via the `x86_adapt` kernel module [12],

and high resolution power measurements. The `power/cpu_idle` (for older kernels `power/power_start`) tracepoint provides the selected C-state for each idle governor decission. The `sched/sched_switch` tracepoint correlates tasks with the CPU they are scheduled on. We added a `power/menu_idle` tracepoint to record the internal decision parameters of the heuristic within the `menu` governor.

Since we want to observe an idle system, we designed the measurement to avoid activities as much as possible. Tracepoint events are recorded by the kernel in a ring buffer. The measurement threads idle in `poll()` until the buffer is nearly full. The only regular interruption is from reading the C-state residency counters of the CPU via `x86_adapt` every 5333 ms. Even the activity of the measurement process itself is recorded through the scheduling events. Power measurements are recorded externally and merged into a common trace file after the experiment.



(a) Repeatedly sleeping for short intervals causing an idle time misprediction. Top: scheduled tasks per CPU, middle: core C-state requested by the `menu` governor (active [blue], C1E [green], C6 [red]), bottom: power consumption. Note that only socket power measurements are available at this time granularity.



(b) Full duration of the Powernightmare: requested idle states, power consumption of both sockets and full system.

Fig. 1. A synthetically triggered Powernightmare on *Diana*. (Color figure online)

3.4 Cause, Trigger, and Contributing Factors

The cause of the unusual high idle power consumption is a severe underestimation of the upcoming idle time by the `menu` governor. Its heuristic has to resort to historic knowledge, as not all events can be known in advance. Bursts of activity with short idle times after which the CPU idles for a long time can confuse the heuristic. Based on the observation of recent idle times, the heuristic concludes that a short idle time will follow, regardless of the next known timer event being far in the future. Moreover, due to discarding long intervals in cases of high variance, the algorithm will often not correct its prediction after the first long idle time. As up to two outliers are ignored by the heuristic, it can take up to three consecutive wake-up events to recover from a misprediction. If a CPU goes into a shallow sleep state but stays there for a long time, it wastes power because it could be sleeping much deeper. We call this a *Powernightmare*.

```
#include <unistd.h>
int main() {
  #pragma omp parallel
  while (1) {
    for (int i = 0; i < 8; i++) {
      #pragma omp barrier
      usleep(10);
    }
    sleep(10);
  }
}
```

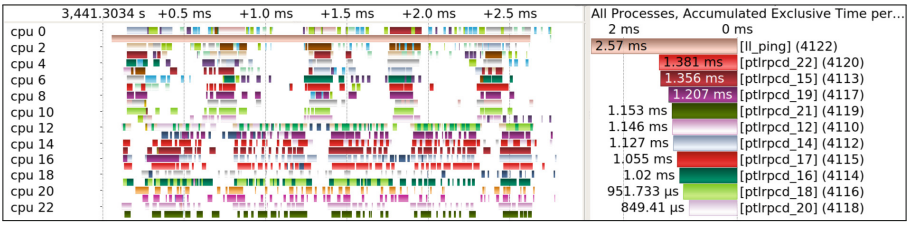
Listing 1. Code to reproduce underestimation of the `menu` governor

We use the code from Listing 1 to reliably trigger a Powernightmare. This code repeatedly sleeps for a very short time tricking the `menu` governor into predicting an upcoming short idle phase, hence requesting a low C-state. An execution of this code is shown in Fig. 1a. Our test system *Diana* is optimized for little background activity, tasks are scheduled infrequently. Therefore it takes up to 10 s before all involved cores are able to end their Powernightmare—especially because it takes up to three wake-up events to correct the misprediction (see Fig. 1b). During the shown Powernightmare, the idle consumption increases from 73 W to 125 W–109 W, depending on the number of cores in CC1. As long as at least one core is in CC1 state, the respective socket cannot enter the PC6 state, wasting most of the energy saving potential. The hardware C-state residency counters are consistent with the selected C-states by the `menu` governor.

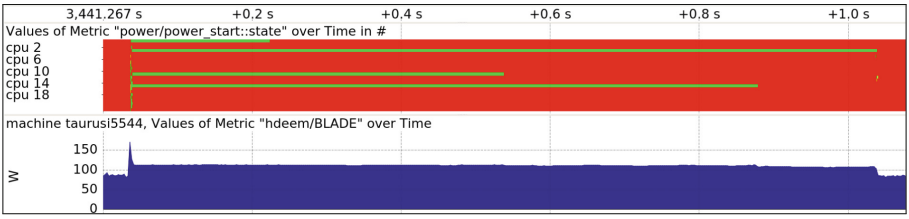
We have seen Powernightmares being triggered in normal operation. On the production HPC system *Taurus* it occurs regularly when no jobs are running on a node. Figure 2a shows a scheduling pattern that happens every 25 s. This regular activity is related to the parallel Lustre filesystem and the interaction of its pinger thread (`ll_ping`), the OFA Infiniband network driver (`kiblnd_sd`), and PortalRPC daemon tasks for each CPU (`ptlrpcd`). As shown in Fig. 2b, several cores remain in C1 for up to one second[2]. Due to regular background activity, the Powernightmare rarely lasts more than one second on this system.

There are various other causes for Powernightmares on our test systems. Most of the time, the cause is communication between user and/or kernel tasks scheduled on different cores, such as `systemd-journald` and I/O related kernel

[2] The `power/power_start` tracepoint event does not distinguish between C1 and C1E.



(a) Scheduling of Lustre related kernel tasks causing an idle time misprediction.



(b) After the Lustre ping (short power spike) several cores remain in C1 (green) instead of C6 (red) for ≈1 s causing increased node power consumption.

Fig. 2. A Powernightmare in normal idle on a *Taurus* HPC node. (Color figure online)

tasks. Another example are updates from a GNU `screen` status bar that affect a shell process and the kernel task handling the respective tty, all waiting for one another for very short time periods. Further causes invole reading model specific registers, which is done by a kernel worker scheduled on the specific core.

We also observed Powernightmares on an Intel Xeon Phi 7210 machine. However, the impact there is reduced, due to recurrent events on all cores every 100 ms, which allows the governor to correct a misprediction within 300 ms.

4 Optimization and Results

We identify several approaches to address the problem of wasted energy due to Powernightmares. Further we describe our selected solution and evaluate it.

4.1 Approaching the Problem

Changing task behavior to avoid triggering a Powernightmare. In many cases it would be possible to tune the applications or kernel tasks such that they no longer trigger an idle time misprediction. For instance, pinning tasks that communicate with each other on the same core could prevent short sleep times while one task is waiting for the other. The pattern exhibited by Lustre involving several kernel tasks per core appears to have significant potential for general improvement. However modifying a wide variety of legacy code is intractable as a solution. Even for newly written software, the complex interactions between different components make it hardly feasible to address the problem this way.

Improving the idle time prediction. A C-state governor must function with incomplete information. It is conceivable to improve the prediction in some cases, e.g., using improved heuristics or software hints. However, there will never be perfect information about upcoming events in general. Applications or outside influences such as network packets cannot be generally predicted.

Biasing the prediction error. It would be possible to tune the heuristic towards over-predicting idle times instead of under-predicting them. The resulting energy savings come at the cost of increased latency. This trade-off could be tunable according to user preferences. One aspect that can certainly be improved is to not generally discard long idle times as outliers in the analysis of recent history.

C-state selection by hardware. For Intel processors, the C-state requested by the `mwait` command is only a hint to the hardware, which may chose to override this decision. However, current Intel processors offer no feature to automatically promote the cores into CC3 or higher. A possibility would be to always request the highest C-state and then relying on auto-demotion or delayed deep C-states for low latency as well as auto-undemotion for low power. Then the OS would no longer be able to enforce latency requirements.

Mitigating the impact. As any modification of the heuristic cannot improve every possible situation, we focus on mitigating the impact of a misprediction. A simple workaround is a program that runs a thread pinned to each core which sleeps for 10 ms in an endless loop. Using a kernel with regular scheduling clock ticks has a similar effect. This avoids staying in an inefficient sleep state for a long time but comes at the cost of a permanent background noise. And while it may save power in some situations, it does increase idle power measurably compared to perfect deep sleeping. Inspired by this workaround, we describe a solution in the `menu` governor without the disadvantages.

4.2 Fallback Timer

To mitigate the effect while avoiding permanent background noise, a core has to wake up from a shallow sleep state only. To achieve this in the `menu` governor, we set a special *fallback timer* if there is a very large factor between the next known timer event and the predicted idle time. This fallback timer is set so that if the prediction heuristic was right, the core wakes up before the timer triggers. We then cancel the additional timer to avoid generating noise. If the heuristic was wrong, the programmed wake-up allows to go into a higher C-state and continue sleeping with lower power consumption. To achieve this, we instruct the kernel to ignore the recent residency history for the upcoming idle state selection. We choose to use the *hrtimer* API of the Linux kernel for our implementation. Regular timers have too low resolution and will miss their deadline by a significant margin on tickless kernels, which would render our solution ineffective.

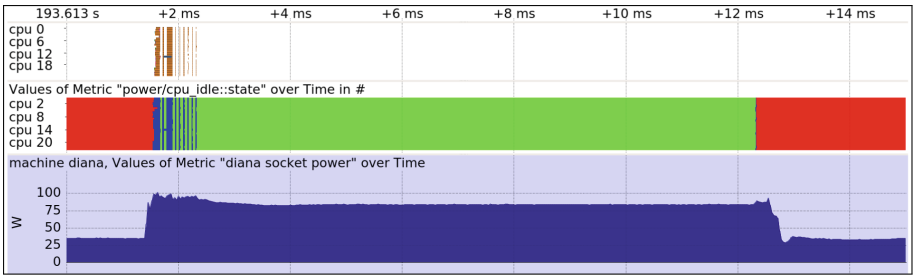


Fig. 3. The fallback timer corrects a wrong C-state selection after 10 ms of shallow sleep. Timeline diagram from top to bottom: scheduling activity, core C-states (active [blue], C1E [green], C6 [red]), power measurements. Note that only socket power measurements are available at this time granularity. (Color figure online)

4.3 Verification

To determine the effectiveness of our solution, we compare an unmodified kernel against a patched kernel with enabled fallback timer. We compare normal idle with no user activity and the worst-case trigger workload as described in Listing 1. Powernightmares that occur in normal operation often exhibit a strong variance and depend on many environmental factors. For the sake of statistical significance and reproducibility, we focus the quantitative verification on the simple synthetic workload, that is >99.99% idle, and normal idle configuration.

Figure 3 shows the timeline of a mispredicted idle time. While all cores enter C1E after the trigger executes, the fallback timer is activated and all cores can enter a higher C-state. The duration of 10 ms for the fallback timer is an initial estimate and can be further experimentally refined or dynamically adapted based on target residencies. Figure 4 shows the statistical density distribution of power consumption samples during 20 min. If the trigger workload is active every 10 s, the average system power consumption with the unmodified kernel is 119 W. The modified kernel with active fallback timer reduces the average power to 74.3 W. During normal idle, in which only few Powernightmares occur, the system consumes 75.5 W with the unmodified kernel and 73.9 W with the fallback timer. The difference is hardly statistically significant, but the amount of outliers is reduced and the standard deviation decreases from 8.1 W to 3.5 W by using the fallback timer. With tickless disabled, which also implies the `ladder` governor, the unmodified kernel is not affected by Powernightmares. The regular timer interrupts increase idle power to 78.5 W. The trigger workload does not increase that further. Our results show that the fallback timer prevents Powernightmares, without causing additional power-overhead in normal idle configurations.

We have not observed any other occurrences of Powernightmares when using the fallback timer. Due to the production nature of the system, we could not apply the patch to *Taurus*. A fair comparison would also require to back-port the patch to the old kernel version normally used on the system.

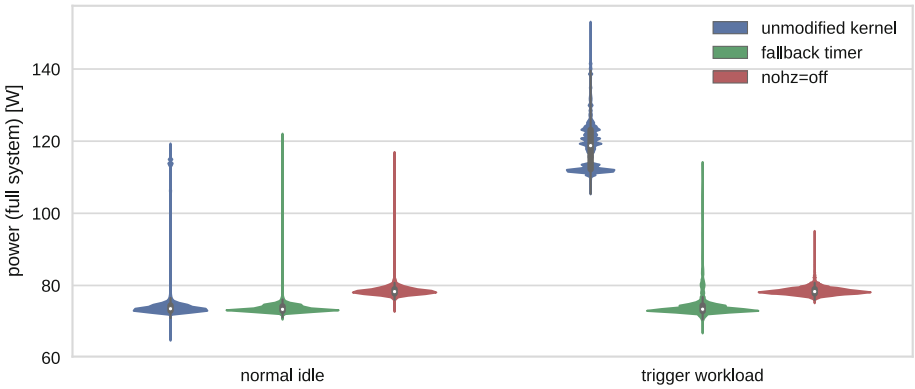


Fig. 4. Combined violin-/box-plot of the *Diana* power consumption during idle and trigger workload using an unmodified Linux kernel, our patched kernel with fallback timer, and an unmodified kernel with disabled tickless (`nohz=off`).

On our system, it takes on average 1735 cycles (0.67 µs) to program a fallback timer. Since the core has no scheduled task a that point, this cost does not impact performance but represents a small energy overhead. When waking up before a set fallback timer triggers, canceling it takes 390 cycles (0.15 µs). This time is added to the wake-up transition latency, which is up to 2.1 µs on our system for C1 and 15 µs for C3 [3], not considering the remaining time spent in the Linux kernel after the wake-up. Considering that this only applies whenever the governor has to deal with conflicting information and the overhead is an order of magnitude lower than existing latencies, we conclude that the negative impact is insignificant for all practical purposes.

5 Summary and Outlook

In this study, we described and analyzed a pattern of inefficient use of sleep states that leads to a significant waste of energy on idle systems. We developed a methodology and open-source tools[3] to carefully observe these anomalies without altering them. Our investigation reveals that a misprediction of the default Linux idle governor can cause the system to enter an inappropriate C-state. In particular, systems with little background activity can stay in this shallow sleep state for ten seconds or more, wasting significant energy-saving potential. We designed a solution to mitigate the negative effects by setting a fallback timer if the idle governor is unsure about the duration of a sleep phase. This allows the system to enter a deep sleep instead of remaining in a shallow sleep state for a long time. We demonstrated that our implementation[4] effectively reduces the average power consumption without notable negative side-effects.

[3] https://github.com/tud-zih-energy/lo2s/tree/powernightmares.
[4] https://github.com/tud-zih-energy/linux/tree/menu_idle_fallback_timer.

While this study focuses on HPC systems, the effects discussed are not necessarily limited to a specific architecture. The same imperfect idle governor runs on millions of mobile devices that all rely on effective sleep-state use to conserve battery life. Since core numbers continue to increase in most devices, so does the likelihood that at least one will sleep badly and thus prevent shared resources from saving power. The impact of our work increases with the gap of power consumption between different sleep states. Further efforts to save energy, which rely on increasing the time spent continuously in idle, would be very susceptible to Powernightmares. Our work therefore contributes to the energy-proportionality for a variety of modern and future systems.

Acknowledgement. This work is supported in part by the German Research Foundation (DFG) within the CRC 912 - HAEC and by the European Union's Horizon 2020 program in the READEX project (grant agreement number 671657). The authors thank Thomas Kissinger for the report and initial discussion that led to this investigation.

References

1. Advanced Configuration and Power Interface (ACPI) Specification, Revision 6.1, January 2016. uefi.org. Accessed 30 Jan 2017
2. Hackenberg, D., Ilsche, T., Schuchart, J., Schöne, R., Nagel, W.E., Simon, M., Georgiou, Y.: HDEEM: high definition energy efficiency monitoring. In: International Workshop on Energy Efficient Supercomputing (E2SC) (2014). https://doi.org/10.1109/E2SC.2014.13
3. Hackenberg, D., Schöne, R., Ilsche, T., Molka, D., Schuchart, J., Geyer, R.: An energy efficiency feature survey of the Intel Haswell processor. In: IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW) (2015). https://doi.org/10.1109/IPDPSW.2015.70
4. Ilsche, T., Hackenberg, D., Graul, S., Schuchart, J., Schöne, R.: Power measurements for compute nodes: improving sampling rates, granularity and accuracy. In: International Green and Sustainable Computing Conference (IGSC) (2015). https://doi.org/10.1109/IGCC.2015.7393710
5. Ilsche, T., Schöne, R., Schuchart, J., Hackenberg, D., Simon, M., Georgiou, Y., Nagel, W.E.: Power measurement techniques for energy-efficient computing: reconciling scalability, resolution, and accuracy. In: Second Workshop on Energy-Aware High Performance Computing (EnA-HPC) (2017, accepted for publication)
6. Intel Corporation: Desktop 4th Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family, Datasheet, vol. 1 of 2, March 2014. intel.com. Accessed 12 Aug 2016
7. Intel Corporation: Intel Xeon Processor E5–1600/E5-2600/E5-4600 v2 Product Families, Datasheet, vol. One of Two, March 2014. intel.com. Accessed 12 Aug 2016
8. Kanev, S., Hazelwood, K., Wei, G.Y., Brooks, D.: Tradeoffs between power management and tail latency in warehouse-scale applications. In: IEEE International Symposium on Workload Characterization (IISWC) (2014). https://doi.org/10.1109/IISWC.2014.6983037
9. McKenney, P.E.: NO_HZ: Reducing Scheduling-Clock Ticks. kernel.org. Accessed 09 May 2017

10. Pallipadi, V., Li, S., Belay, A.: cpuidle: do nothing, efficiently. In: Proceedings of the Ottawa Linux Symposium (OLS) (2007). kernel.org. Accessed 12 Jan 2017
11. Roba, A., Baruch, Z.: An enhanced approach to dynamic power management for the Linux cpuidle subsystem. In: IEEE International Conference on Intelligent Computer Communication and Processing (ICCP) (2015). https://doi.org/10.1109/ICCP.2015.7312712
12. Schöne, R., Molka, D.: Integrating performance analysis and energy efficiency optimizations in a unified environment. Comput. Sci.-Res. Dev. **29**(3–4), 231–239 (2014). https://doi.org/10.1007/s00450-013-0243-7
13. Schöne, R., Molka, D., Werner, M.: Wake-up latencies for processor idle states on current x86 processors. Comput. Sci.-Res. Dev. **30**(2), 219–227 (2014)
14. Song, J.: System and method for processor utilization adjustment to improve deep c-state use, 1 January 2013. US Patent 8,347,119
15. Weste, N.H.E., Harris, D.M.: CMOS VLSI Design - A Circuits and Systems Perspective, 4th edn. Pearson, London (2011). https://doi.org/10.1177/002072098602300231