O/S level interrupt prediction for performance and energy management on Android

Daniel Lezcano, Georges da Costa

O/S level interrupt prediction for performance and energy management on Android

Daniel Lezcano, Georges Da Costa

Abstract—Billions smartphones and smart objects running on batteries use Android, i.e. on the Linux kernel. To save energy, the main kernel leverage is to put processors in a low power state as soon as they are idle. It predicts the next event to estimate the sleep duration and choose a sleep state accordingly. Several wake-up sources (interrupts, events...) impact this prediction which is usually done considering them as a single source. The resulting signal is nearly random and difficult to predict. Over the years, CPUs increasingly supported deeper idle states but the prediction paradigm was never put into question. We propose to predict the next event by splitting the wake-up source signal into smaller and simpler event patterns. We describe a fast and efficient algorithm along with a kernel-level performance evaluation. We compare our approach with multiple reference sleep state selection algorithms on actual ARM and x86 boards using classical mobile workloads. Our proposal detects correctly (up to 20% improved correctness leading to 5% reduced energy consumption) the time of next interrupt, and thus the right sleep level for the processor. We show and discuss the energy impact of the tested prediction algorithm and we compare it with the different generations of sleep level managers in the Linux kernel.

Index Terms—Operating System, Android, Energy Efficiency, Idle Selection, Deep Sleep.

1 Introduction

Energy efficiency of mobile platforms has a direct link to their autonomy. As Android platform runs on billions of devices[23], its operating system powered by Linux is of the utmost importance energy-wise. Linux also runs on most servers (96% of web servers for example[1]) and most supercomputers (100% of the Top500 of June 2022)

These embedded systems, like smartphones, chromebooks or tablets, have an architecture close to a desktop system: they are multi-cores; they run up to multi-GHz frequency; and they have several GB of RAM. As they are running on batteries, the power management for energy efficiency is no longer an option but a critical aspect of the hardware support.

Hardware provides several features to reduce the energy consumption and the operating system (O/S) uses them with different techniques to achieve the best trade-off between performance and energy savings. Even if hardware can handle reactive energy savings, the O/S can better predict future actions of applications based on its fine-grained knowledge. Thus it can help manage the hardware.

One of the power management techniques used to save energy consists in putting the processor in low power mode when idle. There are several low power modes, called idle states, each of them with their own characteristics. These idle states have an enter/exit energy cost[14]. Depending on the depth of the sleep mode, the energy cost can be significant, and the latency can slow down the system if the idle state is not wisely chosen. The processor has to stay idle long enough to reach the break even between the overhead cost[11] to enter and exit the idle state and the total energy saving. If the break even is not reached, the CPU ends up consuming more energy with a latency penalty. On the other hand, if a shallower (less profound) state is used with a processor idle for a longer time, a deeper sleep mode would be more adequate to save energy[28].

Classically, the Linux scheduler is performance focused, thus most of the decisions are using a performance based policy[6]. The standard way to take into account energy efficiency in the Linux kernel is to race to idle[16, 1]. The Linux scheduler[2] implements some basic mechanisms to estimate the idle duration. The estimation of this idle duration is obtained using a sliding average. However, this idle duration does not imply the CPU will be idle during all the time, as it might have to process some interrupt handlers.

There can be up to several hundreds of busy-idle transitions per second, and wrong idle state selections can significantly impact the user experience and battery life.

To optimize the consumed energy during the idle duration, it is necessary to predict the length of the next idle period. Using this period it becomes possible to choose wisely the most efficient low power mode. Knowing the next wake up time makes it possible to find this idle duration. The main goal of our approach is thus to build an efficient predictor of the next interrupt time.

In traditional methods, predictions are based on a recent idle's duration considering there is one single wake up source. It is assumed we can statistically predict the next event based on past events. This approach works if there is a coherence between the interrupts. In reality[15], these interrupts depend on multiple wake up sources. Processing

- *D. Lezcano is working for Linaro*
  E-mail: daniel.lezcano@linaro.org
- *G. Da Costa is with Toulouse University, IRIT, France*
  E-mail: georges.da-costa@irit.fr

Manuscript received XXX; revised XXX.

1. Zdnet, published on Oct 2015, https://www.zdnet.com/home-and-office/networking/can-the-internet-exist-without-linux/

2. Idle duration computation when waking up a task: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/sched/core.c

those interrupts as coming from a single source results in trying to find what number follows after a series of random numbers. For this reason, the idle state selection is usually inaccurate. The proposed approach separates the multiple wake up sources depending on their origin. It models and proposes an efficient predictor for the main wake up source. This model is more precise than using all sources as it removes polluting interrupts.

Improving the idle duration prediction allows the operating system to select the best idle mode. This improved prediction lets the operating system optimize precisely the power consumption and the performance of the system. Using a shallower than possible idle state leads to over consumption. Using a deeper than needed one leads to a higher overhead to exit the state, thus reducing the performance. So improving the idle duration prediction improves either performance or energy efficiency depending on the type of workload.

The main contributions presented in this article are the following:

- New low overhead interrupt prediction algorithm with high correctness (up to 20% improvement) and energy improvement (up to 5%) on several workloads on ARM and x86 along with its implementation in the Linux kernel;
- Experimental performance evaluation of idle state selection (correctness and energy) on ARM and x86 boards;
- Comparison of multiple idle state selection algorithms including the one included in the Android O/S (menu governor) and one currently worked on (TEO governor) on actual Android boards.

The next section will cover the state of the art. Section 3 will describe the links between interrupt prediction and processor idle modes present in modern processors. Then section 4 will show our proposed interrupt prediction approach. Section 5 will present the implementation and validation of this approach and section 7 will conclude and provide several perspectives.

2 State of the art

Heterogeneous processor architectures like big.LITTLE[27] or dynamIQ[17] running at very high frequencies received all the attention to apply dynamic voltage and frequency techniques driven by an energy aware scheduler. This keen interest has put the idle path energy saving in the background, resulting in a paradigm that has never been questioned in more than a decade.

In 2007, the Linux kernel had the power management framework enriched with a new framework dedicated to handling CPU idle states[18]. The implementation has since been split into different components including the governor. This component is there to select the CPU idle state. It includes all the prediction logic, based on *idle - busy* transitions and has been tuned over time with the x86 hardware evolution.

These first implementations were naive, tended to ignore some idle states and needed to have all the cores idle at the same time to operate. In [13] authors propose an array of bits reflecting the idle-busy activity of all the CPUs and sampled at a very high frequency. The resulting past events array is used for a dynamic Bayesian network in charge of the events forecasting.

Prediction is a recurrent topic in power management because the knowledge of the very near future enables one to take decisions proactively instead of simply reacting to a specific situation. For instance, instead of relying on the CPU load, sampled at a low rate, user space can take control of all actions based on a power model via a dedicated multi purpose register mapped to user space. User space can then handle a more complex computation and request a specific performance state to achieve better performance with less power penalty[5].

Carrying out the prediction logic from the kernel to the user space was used to switch to an idle state selection policy which depends on the running application, the idea being that a generic prediction system can not exist as it is impacted by the characteristics of the workload[19].

Recently, Neural Processor Units (NPU) are getting more common on embedded systems and the power management system tries to harness the capabilities of this new type of hardware. In [4], authors use machine learning to build a prediction model based on the system load.

For GPUs, using Idle time to put the GPU in low consumption mode[26] uses an idle time estimation method. Authors increase the deepness of the sleep mode over time to reach the best balance between energy and performance. Similarly as for NPU, GPU are simpler than CPU are they are not running multiple user and operating system processes and their number of wake up sources is lower. Also for these types of accelerators, the operating system is still able to monitor the activity when they are in sleep mode. For CPUs, the operating system itself is suspended in idle mode, making it difficult to reverse an inappropriate decision.

Machine learning has its limitations, especially when tracking the system events from the kernel where the memory is limited and where patterns often change. In [22], authors present an adaptive online algorithm based on the sparse online learning algorithm. It leads to a significant amount of energy savings under certain circumstances.

The system can run with a periodic timer preventing any form of deep idle state because the tick timer stays on even during the idle periods. A power efficient system must power down the tick timer when there is nothing to do, so it can go to deeper idle states. Such a tickless system has one drawback: if the system selects the shallowest state because the prediction estimates the next wake up to occur very soon while it actually occurs way later (eg. tenth of seconds), the CPU will busy loop, draining a lot of energy and overheating with the risk of a thermal emergency reboot. This effect is not only a theoretical one, it has been observed[24] and evaluated[3].

Another predictor called TEO governor[9] is based on the observation that events are usually timer based. Hence the prediction can ignore the other source of wake ups. The results of such an approach allows a very significant improvement in terms of energy saving and performance. Another aspect is that the predictor tells the framework to not switch off the tick if the next predicted idle period is shorter than a tick. Thus, if the shallowest state is selected,

the tick will force a re-computation of the next event with a new idle state selection.

Observation of idle periods is complex as any traditional monitoring tool might have an impact on the idle duration. In [14], authors propose a low level monitoring tool to help detect and characterize idle periods and the events related to exiting these idle periods.

Compared to these approaches, we propose an integrated approach at the kernel level. It allows us to have more fine-grained control and low overhead. Also, as the goal is to manage the processor itself, errors in prediction cannot be corrected later on as the operating system itself can be suspended. Also as the proposed approach is integrated in the Linux kernel, its behavior is constrained by the existing capabilities of this operating system.

3 Interrupt prediction and Idle mode

When a CPU has no more tasks to run, the scheduler will switch to the special *idle* task. This task contains an infinite loop entering and exiting the sleep mode after choosing the most adequate processor idle state given its assumption of the future events as shown on Figure 1. This place is very particular, the kernel inserts all kinds of internal tasks before going idle, like synchronizing shared structure. It also saves the current context, deals with the cache coherency mode and switches to a timer belonging to a different idle state's power domain. Investigating the power benefit of the idle loop is very hard from a software point of view without the help of a physical measurement with dedicated sensing resistors or energy counter registers (such as RAPL on Intel processors). Even with the latter, the firmware can take control of the idle mode and overcome the kernel decision by auto-promoting the idle state to the deepest one. So it is not possible to directly measure the power consumption of a particular idle state as there is no guarantee the firmware will not change it during the measure.

In this article we will focus on improving one part of this mechanism: the prediction of future events. We assume that using a good prediction will improve the energy efficiency by helping the scheduler in choosing the best idle state.

3.1 Break even

The terms *shallow* and *deep* idle states are used to describe the depth of the CPU sleep: the deeper an idle state, the lower the energy consumption and the longer the sleeping and wake up period. All the operations to enter and exit an idle state have a cost directly tied with the complexity to enter such an idle state. This cost is the time (and thus energy) needed to flush caches when entering the state and refill them when exiting, CPUs synchronization, in addition with some more operations which are platform dependent for the most complex idle states. An example is shown on Figure 2.

The energy saving during the idle time must absorb this extra energy consumption and still save more energy than the previous idle state. This point where an idle state becomes more energy efficient than all the previous ones is called the *break even* and the sleep duration to reach it is called the *target residency*.

The constants needed to compute the target residencies of each idle state are usually considered by processor vendors as intellectual property, thus unavailable. Only the already computed target residency time for each idle state is available and is provided by the firmware.

The break even point is reached when the Energy consumption of the idle states are crossing, thus the equality:

$$E_{idle1}(ResidencyTime) = E_{idle2}(ResidencyTime) \quad (1)$$

The consumed energy for a specific idle state at a specific OPP[3] is the power drained during an amount of time, added with the cost to enter/exit the idle state. To simplify, the notation the enter and exit costs will be summed as the wakeup cost (Wakeup).

$$E_{idle}(time) = P_{idle} \times time + Wakeup_{idle} \quad (2)$$

So by using Eq. 1 and Eq. 2, we can obtain the Residency Time:

$$ResidencyTime = \frac{Wakeup_{idle2} - Wakeup_{idle1}}{P_{idle1} - P_{idle2}} \quad (3)$$

3.2 The wake up sources

As we have seen, the choice of the best idle level is important to optimize energy consumption. Predicting the next waking event is needed to choose the best idle level.

The current prediction paradigm in the Linux kernel is based on idle duration statistics. The idle duration is measured by computing the delta time between the enter and the exit times. In order to reduce the exit latency as much as possible, the statistics for the prediction are postponed to the next idle entering sequence. The idle duration can give a hint of the correctness of the state selection: if the duration is greater than the idle state's target residency but less than the next one, then the idle state selection is correct. The Linux kernel does this computation and exports the statistics to user-space via the debug file system.

The idle duration computation is done regardless of the origin of the wake up which can come from different places with very different characteristics. There are basically three main sources of wake up: timers, device interrupts and inter processor interrupts as shown on Figure 3. They are managed by an IRQ manager chip called the Generic Interrupt Controller (GIC)[4]. They can be categorized as: deterministic, predictable, and random. With a better knowledge of the wake up sources, the next prediction can be characterized in terms of accuracy.

3.2.1 Timers

By essence the next timer event is predictable. Its precision has been considerably improved during the last decade with the support of the high resolution timer providing up to a microsecond resolution. In addition, the time framework has adopted the nanosecond time unit. The trend in kernel development is towards increasing the precision. The timer framework deals with the timer wheel[8] but always sets the next timer event in any case, so it is trivial to get this information. This source of wake up is deterministic and can be used as a base to sort out idle states.

3. Operating Performance Points. One of the specific combinations of frequency and voltage which are available for the processor.

4. https://developer.arm.com/documentation/ddi0471/b/

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3253798

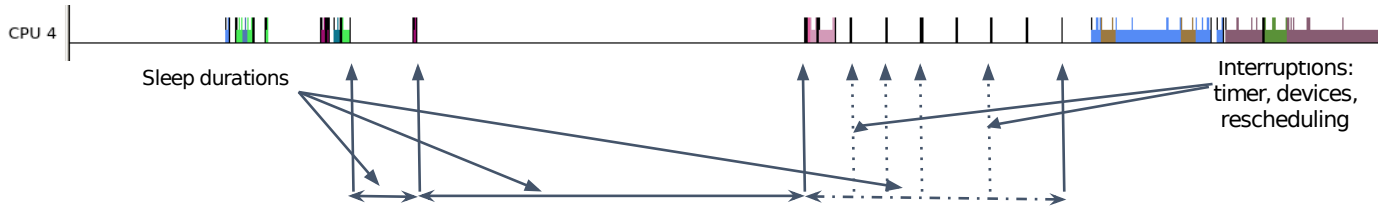

Fig. 1. Timeline of the state of a core. Peaks represent the wake-up of the core. In some phases the core is woke-up by timers while having no work to execute (such as in the middle).

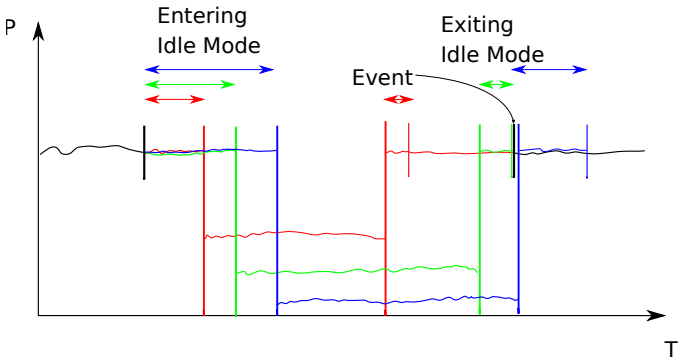

Fig. 2. The red idle-level is faster to enter and exit but it does not compensate for the increased power consumption during idle. The blue one is more profound but is interrupted by an event leading to lateness and overhead of the entering cost.

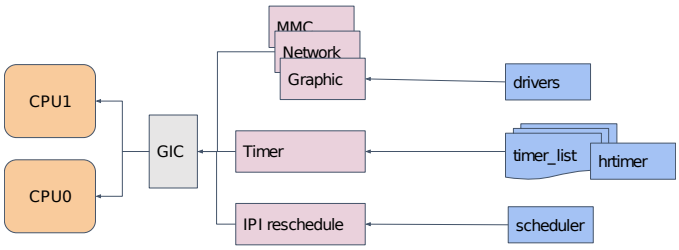

Fig. 3. Wake up sources in a multi-core context. GIC is the Generic Interrupt controller.

3.2.2 *Device Interrupts*

Devices usually have a dedicated task to achieve (send a network packet, write or read on disk, render a frame...). While the duration to accomplish a specific task may vary, it is usually periodic, so those wake up sources can be considered as predictable.

Some devices are much less predictable when depending on external events, such as network incoming traffic, a keystroke or a touch on the screen. However, the time scale is so large compared to the microsecond based prediction, that they can be considered as low density noise and ignored.

3.2.3 *Inter Processor Interrupt*

The main goal of the Inter Processor Interrupt (IPI) is to give the scheduler the ability to wake up another CPU when there is something to do. Basically a task migration or a bottom-half[5] lead to an IPI. The latter could be mapped to the corresponding interrupt event but the migrations are totally dependent on the scheduler and can be considered as unpredictable. The scheduler takes the decision to do IPI depending on a large number of factors with different timescales: Load balancing between cores and clusters, thermal pressure, kernel threads, real-time tasks and more.

5. Asynchronous handling of interruptions.

4 Methodology

When a system is not fully loaded, it alternates between busy and quiescent periods. The importance of the idle routine comes from its usage during the latter. Idle usage happens when the CPU is waiting for a device completion notification via an interrupt, when there are less tasks running than the available processors computing power, or if the tasks are blocking each other like a video / audio pipeline rendering.

In the signal processing field, a complex periodic signal usually results from the sum of simpler periodic signals. Usually to go back to the simpler signals, a Fourier transformation is needed. We make the hypothesis that splitting the wake-up signal depending on the different sources is sufficient: instead of measuring the idle duration, we measure the interrupt events by category before they are mixed between predictable timer events and random rescheduling events.

Our proposal will focus on improving the prediction of the interrupt wake up source. The method for idle prediction is to measure the idle duration period and anticipate the next duration. However, we have basically three sources of wake up: interrupts; timers; and inter processor interrupts. By measuring only the idle duration period, we measure all these sources as they were coming from the same place. In other words, we try to find out a repeating pattern inside mixed signals. However, timers are perfectly predictable while inter-processes interrupts are mostly a random signal. In between, the other interrupts show a repeating pattern. This paper focuses on splitting all these sources of wake up to predict the interrupt wake up sources individually in order to improve the overall prediction.

Prediction only makes sense if the wake up source has some regularity. We will show that on several mobile platforms the wake up source indeed has regularity. Then we will describe the algorithm to detect such a pattern. Afterward, in order to check the correctness of the prediction, we will disable all the idle states. It will guarantee the timestamp measurement of events won't be delayed by the idle state's exit latency. And finally, in order to measure the improvement of the prediction without being impacted with IPI noise, we will run the tests without the SMP (Symmetric Multi Processor) support, limiting the usage to a single core.

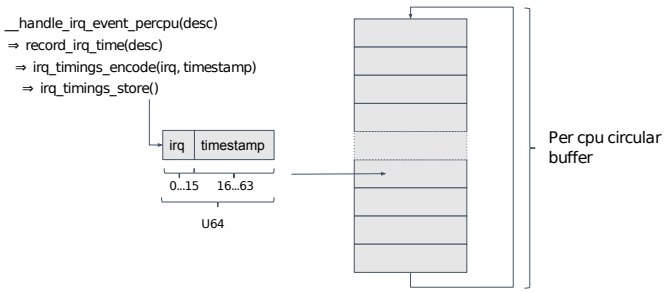

Fig. 4. Captured interrupt timings are stored in a circular buffer using a structure stored in 64bits.

4.1 Predicting the next interrupt

Strictly speaking, all the wake up sources described above are actually interrupts. For the sake of clarity, the next interrupt prediction will refer to the interrupts which do not originate from a timer device or an IPI.

The goal of the prediction will be to find the period of a repeating pattern.

4.1.1 Interrupt capture system

The Linux kernel now provides a single hook where it delivers interrupts with their private data to the registered handlers. We merged a change in the Linux kernel[6] to capture the interrupt timings. Using this handler, this entry is added to a circular buffer array with 32 cells. The interrupt events history is thus recorded in a circular buffer array (Figure 4). The circular buffer contains the interrupt number and the timestamp information. This one is a 64bits size. With a separate definition, that is a 32bits integer for the interrupt number and a 64bits for the timestamp, at each interrupt we trash several cache lines when browsing the circular array. In order to optimize the size of the information and reduce the impact on the cache, we store an encoded 64bits value with the interrupt number on the first 16bits and the time-stamp in nanosecond for the last 48bits. This approach has been debated on the public Linux kernel development mailing list[7].

If a device is generating a large number of interrupts, these will fill the circular buffer by overwriting the events coming from other devices with a smaller number of occurrences. Our hypothesis is that this interrupt will have a much greater probability of happening in the near future than the other devices with fewer events in the same time frame.

4.1.2 Intervals

At the start of an idle period, interrupts are disabled on the local CPU, so the interrupt history is frozen while processed. For each interrupt in the circular buffer array, we compute the interval between the occurrences in microseconds and store them in a dedicated circular buffer, one per CPU and per interrupt. Any interval more than one second apart is considered too old and will result in a drop of all the previous values, thus preventing predictions for old events being endlessly computed.

The range of intervals can greatly vary even with one buffer per source event. It is still difficult to compute a prediction based on the occurrences with high amplitude when they are not regular. In order to adapt to the different timescales, we store the intervals into log_2 ranges. As we consider intervals longer than one second too old, the maximum range will be between 500000 and 1000000 μs, so reduced to log_2 it is 20. The set of values to predict is then limited to [0,20]. The reasoning behind using the log_2 to reduce the interval numbers is the following: the larger the interval is, the bigger the error can be which is considered acceptable. There is a reduced number of idle states and the maximum target residency is rarely greater than 5ms. So for a large interval, there is a high probability the value will be greater than the maximum target residency. At the opposite end of spectrum, if the intervals are small, a high precision is important to discriminate between nearby target residencies. The log_2 operation fits perfectly for this purpose. The Linux kernel low resolution timers use the same approach. The usage of log_2 in the algorithm mimics how timers are ordered in the timer wheel. The longer is the duration, the lower is the precision. Also, all recent processors have a dedicated assembly log_2 instruction. These data will be stored in an array called $classification$ with 20 cells, each one counting the number of intervals between 2^i and $2^{i+1}\mu s$.

The log_2 reduction provides a $classification$ of recent intervals between recent events. To keep more ancient knowledge of previous intervals, a third array (ema) is introduced with the same 20 slots. This array will store the moving average of the classification array, *i.e.* each cell will store $\alpha\ classification_t + (1 - \alpha\ ema_t)$. Such an exponential moving average is heavily used in signal filtering[2] or stock tracking[20], its response to changes can be tuned[7] with the α factor, thus giving an area of tweaking when tracking the interrupt event signals.

The EMA is the exponential moving average also known as an exponential weighted moving average. The main usage of such a statistical formula is to give more weight to recent values on a sliding average. By this way, old values have less impact on the computed average than the recent ones. The EMA helps to quickly react to the interrupt behavior changes like a burst of activity or at the opposite a return to a quiescent point. The α factor gives more or less weight to the recent values, thus influencing how fast the decay of the old values happens.

In addition, the computation of the average on the fly is very simple and helps to reduce the impact of the overhead in a path dedicated to saving energy. An example is shown in Fig. 5.

4.1.3 Pattern detection

At this point, all the necessary information is present to put in place the computation for the next interrupt event. The predictability of an interrupt depends on the repetition of a given suite of numbers. In our case, those numbers are the history of the past events under their log_2 form. This

6. Interrupt timings capture: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b2d3d61adb7b73cfe5f82404f7a130a76fc64232

7. https://lore.kernel.org/all/alpine.DEB.2.20.1703232037560.3723@nanos/

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3253798

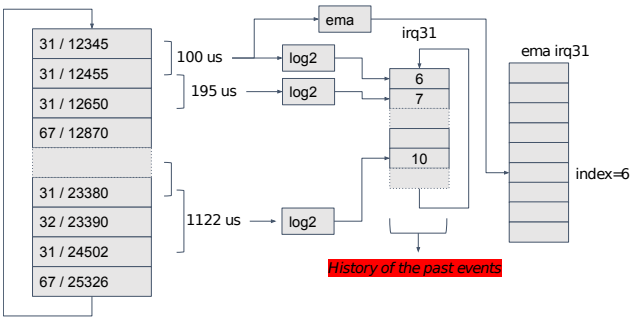

Fig. 5. Architecture of pattern detection. On the left: the *kernel interrupt* circular buffer. In the middle: the $classification$ buffer. On the right: the ema buffer obtained using an Exponential Moving Average.

Interval	4	5	112	4	6	4	110	4	4	5	112	4	7	4	110
log2	2	2	7	2	2	2	7	2	2	2	7	2	2	2	7
p=5	2	2	7	2	2	2	2								
p=4	2	2	7	2	2	2	7	2	2	2	7	2	2	2	7
p=3	2	2	7	2	2	7									
p=2	2	2	2												

Fig. 6. Suite search method: $Interval$ contains the raw data of time in μs between the most recent events. Each period p from five to two is tested with the log_2 of the interval circular buffer (log_2). In this particular example, the comparison succeeds only with $p = 4$.

repetition has a period which is the length of the suite of numbers, so wherever the pointer is in the repetition, the suite is different but the period remains the same.

In order to detect the repetition, we will use a dedicated data structure, the array suffix[21], to build the different possible suites from the history of the past events and from there move a cursor to find out which one of these suites matches the most recent events. We merged a change in the Linux kernel to provide the array suffix data structure[8]. The length of the suite is the period of the events.

The array suffix is a data structure usually used for full text index search, data compression algorithm, bibliometric, combinatorics on words or bioinformatic. In order to build a set of suites, we use a simplified array suffix data structure, the suffixes are not sorted at the end because for our purpose we obviously want to keep the same order. As an example, let's consider the word 'banana', the derived suffixes will be 'anana', 'nana', 'ana', 'na' and 'a'. This applied to the log_2 interrupt events history will build from the end of the history a list of suffixes. However, we can not build an infinite number of suites of different periods. Our experiments show that for devices generating repeating patterns, the period of five numbers is usually the maximum observed before having a non deterministic repetition.

Each computed suite with length from five to two, are sequentially compared from a beginning index equal to three times the maximum period, the first suite matching three times is considered valid as shown in Figure 6.

When the period is found, the next number of the suite can easily be found with a modulo operation on the

8. Array suffix data structure: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bbba0e7c5cdadb47a91edea1d5cd0caadbbb016f

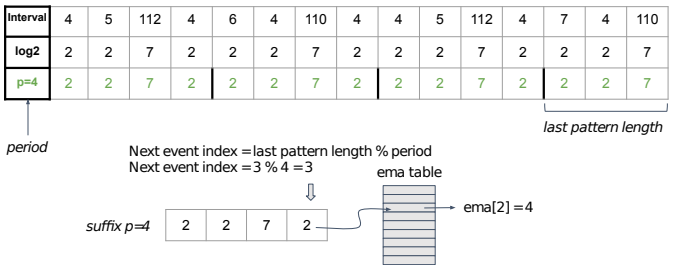

Fig. 7. The scale of the duration before the next event is the next element of the suite. An approximation of this value can be found in the ema table.

remaining length of the suite (Figure 7). The number is the index in the exponential moving average array which gives the average of all these intervals from recent history.

5 Experimentation

5.1 Description of experiments

The first metric used for evaluation is the correctness. The kernel measures the idle duration taking into account the exit delay and compares the results with the selected idle state. If the idle duration is greater than the idle's state target residency but below the next idle's state target residency, then the selection is considered correct. Otherwise the prediction is considered under or overestimated.

The second metric will be energy consumption: the final goal of finding the right idle state is to save energy.

The first experiment will run different governors on two different platforms: x86_64 and ARM64. The former has a higher number of idle states, statistically the wrong predictions will result in a significant increase of the wrong idle state selections. The latter has a reduced number of idle states, thus we can see the impact of wrong predictions on the selection.

The ARM64 platform is based on a 96boards Hikey960[9], a Hisilicon evaluation board with a big.Little architecture. It integrates 4xCortex-A73 and 4xCortex-A53 with 3GB LPDDR4 SDRAM and 32GB of storage. The internal architecture of this board is shown on Figure 8.

This platform is instrumented by soldering a shunt resistor on a dedicated probe place for the PMIC (Power Management Integrated Circuit) supplying the Vcore voltage of the CPUs. The ARM energy probe[10], a 10KHz sampling USB measurement has three channels to read the voltage / current allowing to compute the instantaneous power and thus the accumulated energy. The picture of the experimental board is shown on Figure 9.

The x86_64 platform is an i5 Intel processor based laptop with 8GB of memory and 300GB of storage. The energy measurement uses the Running Average Power Limit register which can be used as an energy counter giving the energy consumption of the group of cores. The interface to measure the power consumption is supported by the Linux kernel in the powercap framework[11] which exports a set of

9. Hikey960 EVB: https://www.96boards.org/product/hikey960/
10. ARM energy probe: https://git.linaro.org/tools/arm-probe.git
11. Power capping: https://www.kernel.org/doc/html/latest/power/powercap/powercap.html

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3253798

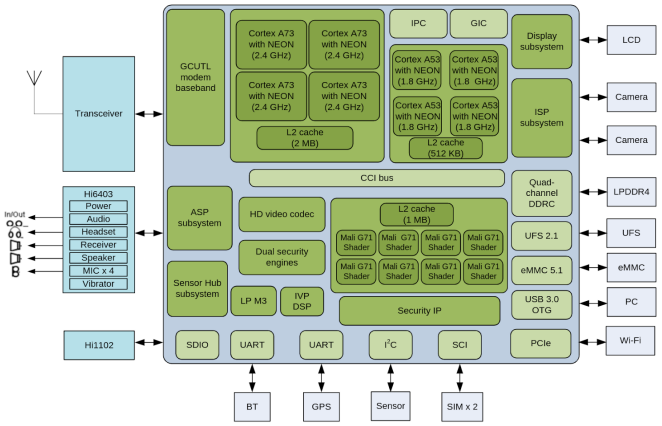

Fig. 8. Hikey960 is based on the hi3660 System on Chip (SoC). Figure from the hi3660 datasheet.

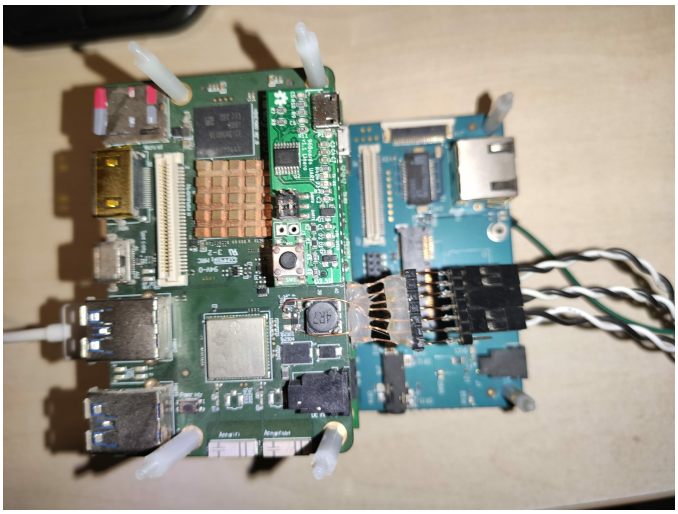

Fig. 9. Hikey960 board with power probe.

files to read the accumulated energy consumption via the filesystem.

5.1.1 Workloads

The set of workloads used for testing is based on typical workloads soliciting different groups of hardware components on the system. They are all part of Ubuntu 18.04 LTS. We are not interested in having full performance benchmarks but instead we want to see workloads where the CPUs alternate quiescent point and burst of activities which are switched by device interrupts.

glmark2: A benchmark running OpenGL with a set of scenes where the 3D hardware acceleration is enabled. The purpose of this benchmark in our test is to have part of the hardware decoding and rendering images and thus generating display interrupts. Given the frames are synchronized with the refresh rate of the display, the interrupts will happen periodically. We expect the interrupt prediction to detect such a pattern.

mp3 : A workload based on multiple decoding pipelines spread across the CPUs while the DSP generates interrupts.

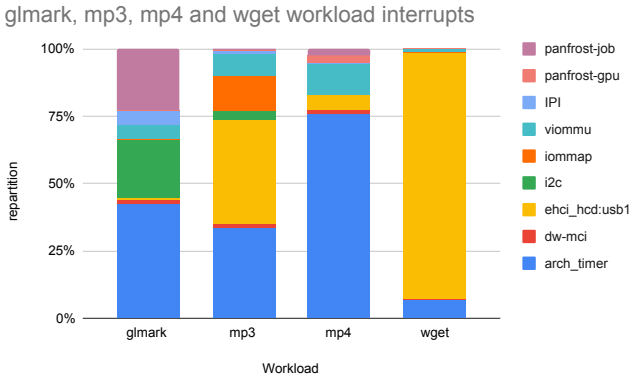

Fig. 10. Distribution of the interrupt sources for different types of applications

workload	**glmark**	**mp3**	**mp4**	**wget**
arch_timer	19828	37532	88441	6054
dw-mci	778	1590	1535	147
ehci_hcd:usb1	414	42888	6595	79667
i2c	10059	3810	30	15
iommap	0	14583	0	0
viommu	2536	9390	13480	1033
IPI	2499	919	15	9
panfrost-gpu	30	321	3369	36
panfrost-job	10569	428	2984	48

TABLE 1
Distribution of the interrupt sources for different types of applications

mp4 : A more complex workload where the hardware enables frame rendering and DSP decoding, providing multiple sources of interrupts where each of them are periodic.

wget : A typical network workload limited by a 100MB/s bandwidth ensuring alternate bursts of activity followed by idle duration. The program downloads a 100MB file size from a local server.

5.1.2 Workloads interrupt statistics

This section gives an overview of the interrupt occurrences on the system while running the workload. It gives another indication of the interrupt activities in order to interpret the testing results.

Figure 10 (using the same data as Table 1) shows the distribution of the interrupt events for several workloads. The **glmark2** workload has half of the interrupts coming from the timer, a quarter from the GPU and another quarter from the i2c interrupts. The **mp3** workload has most of the interrupts coming from the timers and from the USB. Actually the mp3 file is read from the network and the network card is a USB-ethernet adapter. The **mp4** workload has the vast majority of the interrupt coming from the timers. Finally the **wget** workload is mostly characterized by the USB interrupts coming from the network usage.

Figures 11, 12, 13 and 14 show the frequency of the interrupt time intervals for each workload. The intervals are defined in the log_2 space similarly as the presented algorithm. Each figure shows the peaks resulting from the repeating occurrence of the same interval. The initial hy-

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3253798

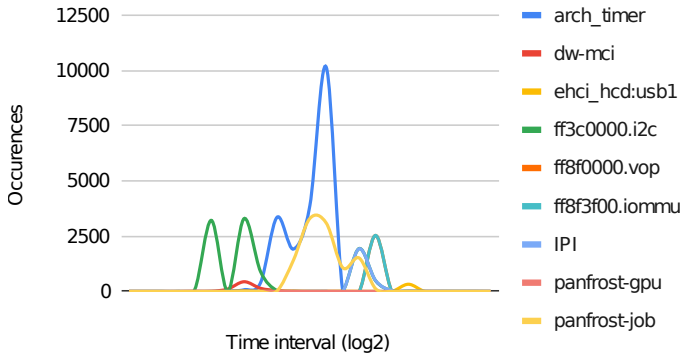

Fig. 11. Frequency of interrupt time interval for the glmark2 workload

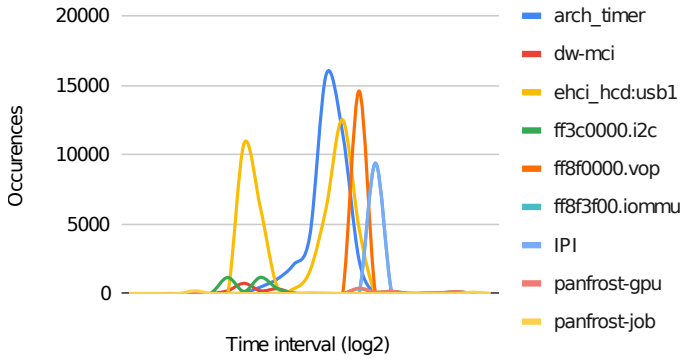

Fig. 12. Frequency of interrupt time interval for the mp3 workload

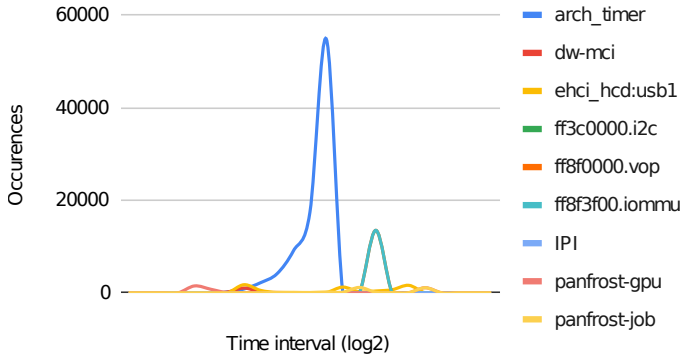

Fig. 13. Frequency of interrupt time interval for the mp4 workload

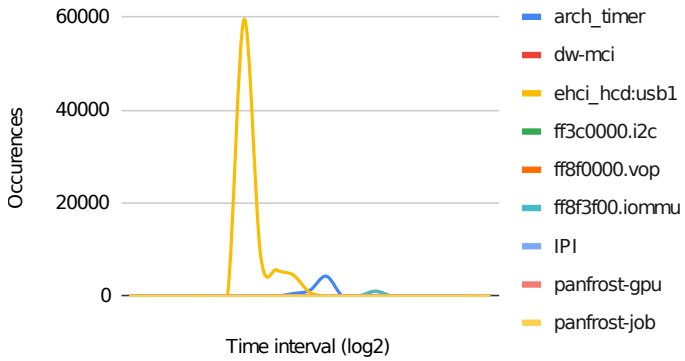

Fig. 14. Frequency of interrupt time interval for the wget workload

pothesis (per interrupt intervals are stable) is confirmed by the shape of the different figures for each workload.

These figures show that most of the interruptions are usually following a normal distribution. The others have peaks showing their frequency are the sum of a small number of normal distributions. So there is a deterministic aspect of the interrupt timing. In some rare cases, a flat figure indicates a random distribution. But actually that must be correlated with the number of occurrences which is usually small regarding the other interrupts. For instance, on the mp3 workload, the panfrost-gpu interrupt has a flat figure but there are 918 occurrences for a total of 120414, so it represents 0.76% of the events.

Regarding activity bursts, several questions must be addressed: What can happen if there are a lot of interrupts on the system? If they are coming from different sources, will they scramble the algorithm? Is there a risk of data overrun?

In the case there are a lot of interrupts, the CPU is constantly waking up or busy looping in the case of network traffic[10]. In this situation, the path taken in the kernel goes rarely in the idle path and the prediction is not needed.

When the system reaches a quiescent point, the CPU goes idle. If the prediction is incorrect and the state selected is the shallowest one for a very long idle duration, the system may be consuming more energy than needed. In order to prevent this situation the kernel does not shutdown the tick timer when the idle state is the shallowest. So in any case, the CPU exits the shallowest idle state and reevaluates the next event.

5.2 Existing governors

In the kernel, the cpuidle framework is responsible for selecting the right idle state. Several implementations exist in the kernel to provide this service.

Currently, there are two governors in the Linux kernel. The first one is called *ladder* and the second one is the *menu* governor.

The *ladder* governor is used on a system configured with periodic ticks, with small wake up latency constraint and does not apply to our purpose because it is focused on the server side where performance is more important than energy savings. It escalates the idle state when the last residency is greater than the idle state's residency. Its main limit is that the maximum sleep length is limited by the tick. It has not been used for more than a decade.

The *menu* governor is used on tickless systems where the periodic tick is disabled at idle time in order to allow long idle period duration and to let the system enter deeper idle states (see Section 2). Initially, the menu governor was introduced in 2007 with an algorithm rewrite in 2009. The new algorithm takes into account the idle duration of the CPUs, that is when it enters and exits the idle state. A small history of the previous events is stored and the prediction uses it to estimate the next event. The result is then processed using an empirical formula resulting from fine tuning on x86 legacy hardware.

5.3 Reference governors

The menu governor is the default governor used for all mobile nowadays and will be our reference governor for

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3253798

our study. All results in the following will be provided as a ratio to this reference governor. The Teo governor is the current state of the art Linux governor and is expected to become the next generation default one.

We propose to add some simple and baseline governors as reference and to help provide insight on the results: one targeting maximum performance; another one aiming to achieve a maximum power saving; and finally the last one always choosing random idle states.

The governors used in the experimental section are:

- shallowest: The selected idle state is always the shallowest one
- deepest : The selected idle state is always the deepest one
- random: The selected idle state is selected randomly
- menu : The widely used governor in mobile systems
- teo : A recently introduced timer oriented based governor (see Section 2)
- mobile : The governor using our proposed next interrupt prediction system

These different static decision governors added with the existing ones help to compare the result of a particular decision regarding a workload. The precision of a prediction does not guarantee we have the best energy saving or the best trade off with the performance. The idle states characteristics with the target residency and the exit latency can be wrong or inaccurate. Also other hardware mechanisms can be involved preventing a CPU to go to a specific idle state like a cluster power down idle state where the last man standing algorithm applies. For instance if a shallowest or a deepest governor is the most efficient governor given a specific workload, then the governor having a prediction mechanism should have similar results for both cases. The random governor is the most naive one and a comparison with a prediction governor helps to understand its accuracy.

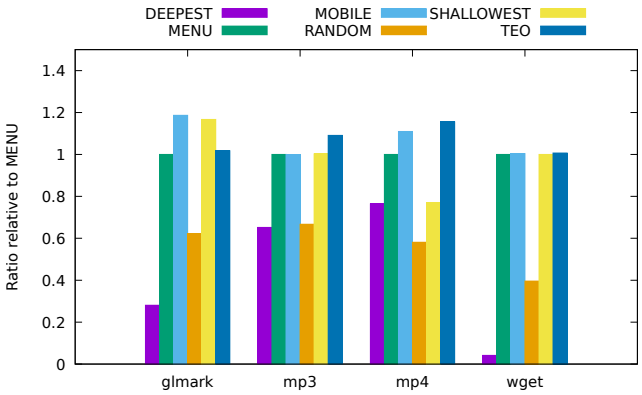

Fig. 15. Measure of the correctness (in number of correct predictions) of the algorithms on ARM. Values are relative to MENU governor (higher is better).

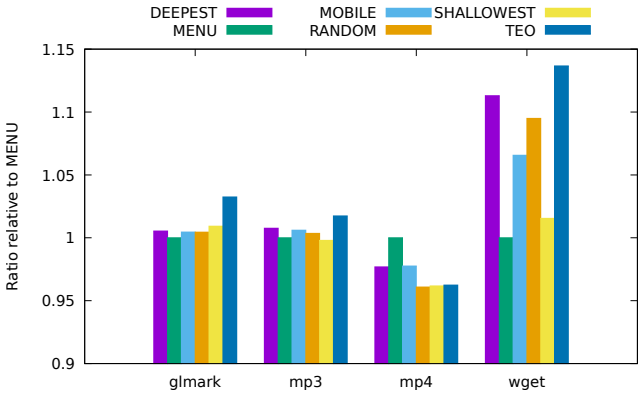

Fig. 16. Measure of the Energy (measured in Joules) of the algorithms on a single core to reduce impact of IPI on ARM. Values are relative to MENU governor (lower is better).

6 Implementation and Validation

The measurements shown in the different figures are all scaled against the menu governor which is the reference governors used for mobiles, desktops and servers.

6.1 ARM

The ARM architecture tests are divided into two sets: one for mono-processor and the other for multiple processor support (SMP). The reason for taking such an approach is because on SMP, one CPU wakes up another CPU using an inter-processor interrupt which is another source of wake up. It results in a CPU exiting the idle state: it becomes difficult to do the correlation with the correctness of the prediction as IPI are not taken into account in the prediction model. Otherwise it is an attempt to predict the scheduler behavior.

The legitimate question is why the IPIs can not be part of the prediction? IPIs are used by the scheduler only as a 'rescheduling interrupt' (other IPIs can be ignored as their occurrences are negligible) and result in a non periodic signal. An application can sleep at any time and if at wake up time the CPU is busy running something else, the process will be migrated to another idle CPU. This migration results in an IPI sent to the target CPU which in turns exits the idle state and executes the process put in the run queue by the scheduler. It is not possible to predict the choice of the scheduler to wake up this CPU instead of another idle one because the idleness of the CPUs depends on the processes' behavior running on the system. That is even more true when the kernel threads are involved in this migration because they add more randomness in the IPI signals.

Figure 15 shows the correctness of the prediction without the IPI involved on ARM64. We can observe that the mobile governor can challenge other governors in terms of prediction even if it is not the best one. As a reminder the mobile governor is very simple and does not integrate anything else than the next interrupt and the next timer event prediction. Other governors base their prediction on experimental observation and hard code some magic numbers to adjust the next event [12]

Figure 16 shows the menu governor beats the other governors in terms of energy when there is one CPU on the system. Given the history of the menu governor, the result is not surprising. The governor was implemented when single core processors were the standard. Unfortunately the

12. Menu governor performance multiplier https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/cpuidle/governors/menu.c#n153

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3253798

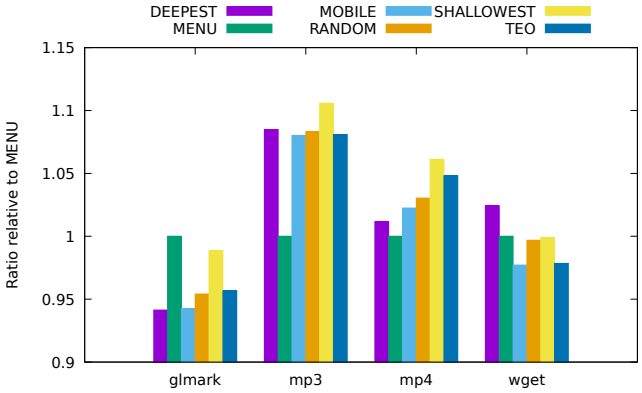

Fig. 17. Measure of the Energy (measured in Joules) of the algorithms on multiple cores on ARM. Values are relative to MENU governor (lower is better).

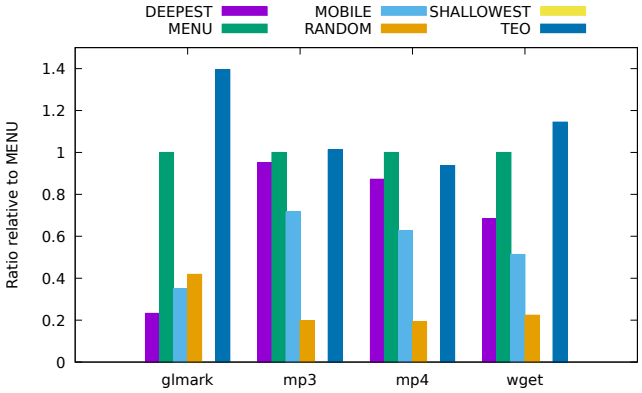

Fig. 18. Measure of the correctness (in number of correct predictions) of the algorithms on x86. Values are relative to MENU governor (higher is better).

paradigm of the menu governor has not been put in question with the multi cores processors. It has been tweaked for server platforms with the introduction of experimental based heuristics. The strength of the menu governor is the idle-busy pattern repetition detection. Without the IPI which introduces a higher randomization of the events, the menu governor performs very well.

Figure 17 shows the mobile governor performs well in terms of energy consumption except for the multimedia workloads where it is similar to the other governors. For energy efficiency reasons, on the ARM platform, all the interrupt affinities are set to the CPU0. This one is usually a low profile processor with little power consumption. The mobile governor uses the next timer and the next interrupt to predict the duration of idle time. Hence on other processors than CPU0, the probability to correctly find out the next event is higher as only the timers are involved. On the other hand, the multimedia workloads are using dedicated threads as pipeline to decode and render the stream. The threads are usually spread across the different cores and their execution is serialized, the IPIs are more involved in those scenarios and the next interrupt prediction is not enough. The task serialization introduces a periodicity in the events. Those, in turn, produce a periodic busy-idle pattern easily detected by the menu governor.

6.2 X86

The x86 architecture aims at achieving the performance needed for the server market. Until recently power management was not the highest priority and having the best performance was an acceptable trade off for the extra power consumption. Even if the mobile governor is designed and tested for the ARM64 architecture, it is interesting to compare it with the x86 architecture to see how it behaves.

This architecture has a specific idle state, called POLL idle state[13]. In order to have the fastest response to a burst of IO requests, a software idle state was created to be inserted as the shallowest idle state. It does a busy loop on the idle task $need\_resched()$ condition when there are tasks blocked on pending IOs. Obviously, there is no wake up latency but at the cost of a full CPU power consumption and extra heat production.

13. POLL idle state: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/cpuidle/poll_state.c

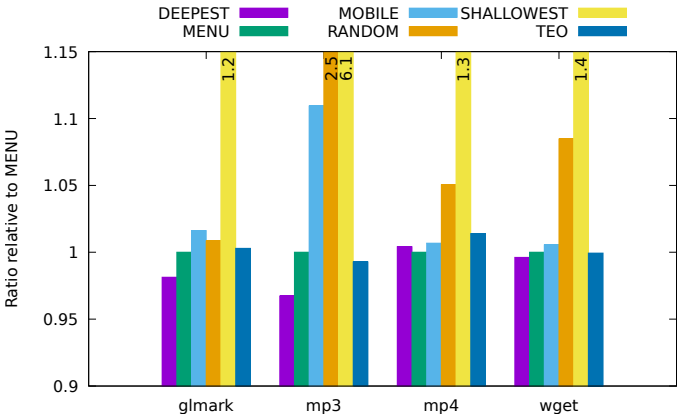

Fig. 19. Measure of the Energy (measured in Joules) of the algorithms on a single core on x86. Values are relative to MENU governor (lower is better).

Figure 18 shows how accurate the predictions are on the Intel platform. The shallowest idle state is the poll idle state, consequently there is no prediction and we can ignore it in the figure. The mobile governor never presents better results than the menu governor. It under-performs in all the cases. The hypothesis is the wake up path could be different under some circumstances and may not be related to the interrupts directly. Another aspect is how the interrupts are configured on the system. The affinity is set on all the CPUs and the interrupt controller can randomize / round robin the interrupt delivery on any CPU. That breaks the irq prediction model as this one is per CPU. Moreover the x86 platform uses different mechanisms to wake up a CPU and its firmware can overcome the governor decisions. Unfortunately, it is difficult to investigate this observation as the x86 firmware is a black box and does a lot of actions on the back of the operating system.

Figure 19 shows the shallowest governor is consuming much more energy than the other governors. This is not surprising as, as aforementioned, it is a busy loop idle state. Except for the random governor, other governors have simi-

lar results. This observation tends to confirm the firmware is acting after the governor decision. For better correctness in the prediction, the energy consumption is almost the same.

The deepest governor chooses an idle state where there is a higher number of components powered down. However, on the x86 architecture, the tests run in the UP processor context, so there are fewer actions done under the hood making the idle state transitions faster, like running the last man standing algorithm in the firmware. It results in a faster full power efficient idle state with a lower latency.

7 Conclusion and perspectives

This article presents our proposal of a prediction based idle mode selector. We compare this algorithm with existing implementations in the Linux kernel along with several naive algorithms.

The current paradigm for the idle state selection is to predict the idle duration based on the past idle periods. Unfortunately this approach can not differentiate the origin of the wake up and it results in attempts to predict the entire system behavior, including the scheduler. Under a medium activity involving all the wake up actors, the current menu governor does not perform better than a governor which randomly chooses an idle state. The menu governor has an acceptable number of correct predictions when the running application performs cyclic tasks like a frame decoding or packet receiving.

The approach consisting of identifying the source of wake up and predicting each of them individually showed good results for a very simple governor which only takes the earliest next events from the three wake up source categories: device interrupt, timer interrupt and inter processor interrupt. Unfortunately, the latter is out of scope for this study and a biased heuristic was necessary to overcome the lack of this support.

While this study happened, the timer oriented governor was implemented with yet another paradigm. As explained by the author of the TEO governor[25]:

> *"The concept behind TEO is that the most frequent source of CPU wakeups on many systems is timer events, not device interrupts. Wysocki notes that timer events might be even two or more orders of magnitude more frequent than other interrupts. So the time until the next timer event alone provides a strong predictive clue."*

TEO governor simply bases all the predictions on the deterministic timer next events. On top of it, it adds some experience based heuristics. The results of this governor clearly surpasses all the other governors including the next interrupt prediction governor. The mobile governor could have been investigated more with the inter-processor interrupts and would have been a good candidate to replace the menu governor. However, given the results of the timer oriented governor, it was considered useless to go further in the study as we are not able to provide the same level of performance / energy without introducing biased heuristic which is exactly what we want to prevent.

The third source of wake up originating from the inter-processor interrupt is an area of investigation for even more accuracy in the prediction. Trying to predict the IPIs is an attempt to predict the scheduler behavior and it would be much more interesting to have the scheduler rely on the next interrupt prediction to choose wisely which CPU to wake up. That implies a better collaboration between the idle framework and the scheduler which is very limited today[14]. The task migration is also a source of confusion for the prediction. Even if the task is periodic, there is no guarantee it will wake up on an expected CPU. As spotted by the article [12], the kernel threads are responsible for the task migrations. This could be an area of improvement to reduce the randomness of the IPIs signal in order to add the IPI prediction support in the mobile governor.

However, even if the next interrupt prediction support is not used to select an idle state, there is potentially an application in the real time area. The deadline scheduler needs to know when the next interrupt will happen on a CPU in order to estimate if the deadline constraint will be fully filled or not.

Acknowledgment

We thank Linaro for funding this research. Thanks to Vincent Guittot who took the time to review the technical aspects of the document. A special thanks to Ebba Simpson for reviewing the document.

References

[1] Susanne Albers and Antonios Antoniadis. Race to idle: new algorithms for speed scaling with a sleep state. *ACM Transactions on Algorithms (TALG)*, 10(2):1–31, 2014.

[2] Belyaev Alexander, Tutov Ivan, and Butuzov Denis. Analysis of noisy signal restoration quality with exponential moving average filter. In *2016 International Siberian Conference on Control and Communications (SIBCON)*, pages 1–4. IEEE, 2016.

[3] Abdullah Aljuhni, C Edward Chow, Amer Aljaedi, Shaji Yusuf, and Francisco Torres-Reyes. Towards understanding application performance and system behavior with the full dynticks feature. In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 394–401. IEEE, 2018.

[4] Manish Arora, Srilatha Manne, Indrani Paul, Nuwan Jayasena, and Dean M Tullsen. Understanding idle behavior and power gating mechanisms in the context of modern benchmarks on cpu-gpu integrated systems. In *2015 IEEE 21st international symposium on high performance computer architecture (HPCA)*, pages 366–377. IEEE, 2015.

[5] William Lloyd Bircher and Lizy John. Predictive power management for multi-core processors. In *International Symposium on Computer Architecture*, pages 243–255. Springer, 2010.

[6] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Tappan Morris, Nickolai Zeldovich, et al. An analysis of linux scalability to many cores. In *OSDI*, pages 86–93, 2010.

14. Leverage the idle state info when choosing the "idlest" cpu: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=83a0a96a5f26d974580fd7251043ff70c8f1823d

[7] Chris Chatfield, Anne B Koehler, J Keith Ord, and Ralph D Snyder. A new look at models for exponential smoothing. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 50(2):147–159, 2001.

[8] Jonathan Corbet. The kernel mechanism managing the different timescale lists of planned events. https://lwn.net/Articles/646950/, 2015. Accessed: 2022-01-27.

[9] Jonathan Corbet. The weighted teo cpuidle governor. https://lwn.net/Articles/820432/, 2020. Accessed: 2022-01-27.

[10] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux device drivers*. O'Reilly Media, Inc., 2005. Chap. 17.

[11] Carlos Cortes, Hideharu Amano, and Nobuyuki Yamasaki. Break even time analysis using empirical overhead parameters for embedded systems on sotb technology. In *2017 32nd Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6. IEEE, 2017.

[12] Gaurav Dhiman, Vasileios Kontorinis, Dean Tullsen, Tajana Rosing, Eric Saxe, and Jonathan Chew. Dynamic workload characterization for power efficient scheduling on cmp systems. In *Proceedings of the 16th ACM/IEEE international symposium on low power electronics and design*, pages 437–442, 2010.

[13] Qian Diao and Justin Song. Prediction of cpu idle-busy activity pattern. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 27–36. IEEE, 2008.

[14] Thomas Ilsche, Marcus Hähnel, Robert Schöne, Mario Bielert, and Daniel Hackenberg. Powernightmares: The challenge of efficiently using sleep states on multi-core systems. In *European Conference on Parallel Processing*, pages 623–635. Springer, 2017.

[15] Thomas Ilsche, Marcus Hähnel, Robert Schöne, Mario Bielert, and Daniel Hackenberg. Powernightmares: The challenge of efficiently using sleep states on multi-core systems. In *Euro-Par 2017: Parallel Processing Workshops: Euro-Par 2017 International Workshops, Santiago de Compostela, Spain, August 28-29, 2017, Revised Selected Papers 23*, pages 623–635. Springer, 2018.

[16] David HK Kim, Connor Imes, and Henry Hoffmann. Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics. In *2015 IEEE 3rd international conference on cyber-physical systems, networks, and applications*, pages 78–85. IEEE, 2015.

[17] Agostino Mascitti, Tommaso Cucinotta, and Mauro Marinoni. An adaptive, utilization-based approach to schedule real-time tasks for arm big. little architectures. *ACM SIGBED Review*, 17(1):18–23, 2020.

[18] Venkatesh Pallipadi, Shaohua Li, and Adam Belay. cpuidle: Do nothing, efficiently. In *Proceedings of the Linux Symposium*, volume 2, pages 119–125. Citeseer, 2007.

[19] Madhu Palmur, Zhichao Li, and Erez Zadok. Cpuidle from user space. Technical report, Technical Report FSL-13-05, Computer Science Department, Stony Brook University, 2013.

[20] Puchong Praekhaow. Determination of trading points using the moving average methods. *Bangkok, Thailand*, 2010.

[21] Simon J Puglisi, William F Smyth, and Andrew H Turpin. A taxonomy of suffix array construction algorithms. *acm Computing Surveys (CSUR)*, 39(2):4, 2007.

[22] Tumaati Ramesh and Suresh R M. Energy efficient utilization of computational resources by predicting cpu idle time. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, Volume-8(Issue-6S3), 2019.

[23] Vijay Janapa Reddi, Hongil Yoon, and Allan Knies. Two billion devices and counting. *IEEE Micro*, 38(1):6–21, 2018.

[24] Marta Rybczyńska. Improving idle behavior in tickless systems. https://lwn.net/Articles/775618/, 2018. Accessed: 2022-01-27.

[25] Marta Rybczyńska. Improving idle behavior in tickless systems. https://lwn.net/Articles/775618/, 2018. Accessed: 2022-01-27.

[26] Mohammad Sadrosadati, Seyed Borna Ehsani, Hajar Falahati, Rachata Ausavarungnirun, Arash Tavakkol, Mojtaba Abaee, Lois Orosa, Yaohua Wang, Hamid Sarbazi-Azad, and Onur Mutlu. Itap: Idle-time-aware power management for gpu execution units. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(1):1–26, 2019.

[27] Ashley Stevens. Introduction to amba® 4 ace™ and big. little™ processing technology. *ARM White Paper, CoreLink Intelligent System IP by ARM*, 2011.

[28] Li-Chuan Weng, XiaoJun Wang, and Bin Liu. A survey of dynamic power optimization techniques. In *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, 2003. Proceedings.*, pages 48–52. IEEE, 2003.

Daniel Lezcano is Senior Engineer at Linaro Ltd company as power management specialist. He worked 1998 in the Space Industry and Air traffic management for a distributed system project in life safety constraints where he acquired a system programming expertise. In 2004, he contributed to the Linux Containers by implementing the network virtualization and other bricks in the system as well as the core container engine in user space. In 2012, he joined Linaro to work in the power management team. Deeply involved in the Linux kernel development, he is a Linux maintainer of the timer drivers, the CPUidle drivers, the thermal and the dynamic thermal power management frameworks.

Georges Da Costa is Professor in Computer Science at the University of Toulouse. He received its PhD from LIG (Grenoble, France) in 2005. He is a member of the IRIT Laboratory. His main interests are related to large-scale distributed systems, algorithmics, performance evaluation and energy-aware systems.

He serves on several PCs in the Energy aware systems, Grid and Peer to Peer fields. He was chair of the COST1305 working group 2 on 'Programming models and runtimes'. His research currently focuses on energy aware distributed systems. His research highlights are grid cluster & cloud computing, peer to peer, large scale energy aware distributed systems, performance evaluation, and ambient systems.