Automated OS-level Device Runtime Power Management

Chao Xu
Rice University

Felix Xiaozhu Lin
Purdue University

Yuyang Wang
Tsinghua University

Lin Zhong
Rice University

Abstract

Non-CPU devices on a modern system-on-a-chip (SoC), ranging from accelerators to I/O controllers, account for a significant portion of the chip area. It is therefore vital for system energy efficiency that idle devices can enter a low-power state while still meeting the performance expectation. This is called device runtime Power Management (PM) for which individual device drivers in commodity OSes are held responsible today. Based on the observations of existing drivers and their evolution, we consider it harmful to rely on drivers for device runtime PM.

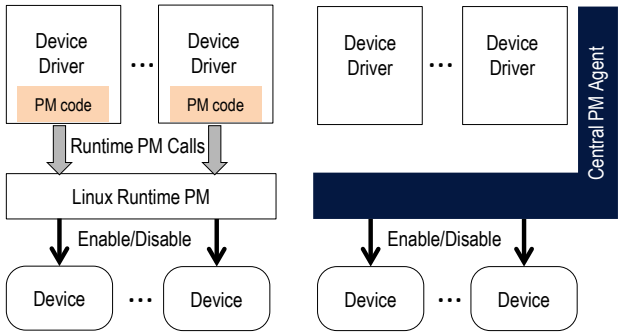
This paper identifies three pieces of information as essential to device runtime PM, and shows that they can be obtained without involving drivers, either by using a software-only approach, or more efficiently, by adding one register bit to each device. We thus suggest a structural change to the current Linux runtime PM framework, replacing the PM code in all applicable drivers with a single kernel module called the central PM agent. Experimental evaluations show that the central PM agent is just as effective as hand-tuned driver PM code. The paper also presents a tool called PowerAdvisor that simplifies driver PM efforts under the current Linux runtime PM framework. PowerAdvisor analyzes execution traces and suggests where to insert PM calls in driver source code. Despite being a best-effort tool, PowerAdvisor not only reproduces hand-tuned PM code from stock drivers, but also correctly suggests PM code never known before. Overall, our experience shows that it is promising to ultimately free driver developers from manual PM.

Categories and Subject Descriptors D.4.7 [*OPERATING SYSTEMS*]: Organization and Design

Keywords Power management; Mobile system; System-on-a-chip; Operating system

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey..

http://dx.doi.org/10.1145/2694344.2694360



(a) Current Linux PM (b) Central PM in This Work

Figure 1: An overview of this work. (a) In the current Linux, each driver is responsible for implementing its PM under the runtime PM framework, which incurs high development burden and is error-prone. (b) The central PM agent relieves drivers from PM.

1. Introduction

A System-on-a-chip (SoC) [1–4] is often the heart of a wide spectrum of embedded systems from tablets to wearables. It is complex, incorporating not only multiple general purpose cores (CPU) but also dozens of IP (intellectual property) modules, most notably I/O controllers and accelerators such as multimedia codec, DSP, and GPU. These IP modules not only bring specialized functions to an SoC efficiently, but also account for the majority of the SoC chip area [5]. Because commodity operating systems (OSes) treat these non-CPU modules as I/O devices and manage them with device drivers, we refer to them as *devices* in this work.

In a system that employs an SoC, power management (PM) is an important design issue that requires software and hardware collaboration. Its goal is to ensure that performance requirements of workloads are met with the lowest possible energy consumption. PM is critical to the energy efficiency and therefore usability of the overall system.

This paper focuses on *device runtime PM* that enables/disables individual devices properly and timely when the system is in use. Its actions are bubbled up to the hardware clock and power hierarchy, finally leading to clock (un)gating, power supply switching, or voltage change. Note that de-

vice runtime PM is not concerned with the CPU, which is an orthogonal research problem [6] addressed by other OS subsystems, e.g., `cpufreq` and `cpuidle` in Linux.

Although device runtime PM has been recognized as vital [7] due to emerging always-on workloads, it is poorly implemented in commodity OSes such as Linux. In this paper, we highlight the shortcomings of Linux device runtime PM: as SoCs appear in the market, their device runtime PM support is often long overdue, and is likely to be minimal when finally implemented.

The root cause of these problems is that today's commodity OSes hold drivers responsible for PM. To understand this problem, we identify three pieces of information as essential to runtime PM: (*i*) the Quality of Service (QoS) requirements, such as device wakeup latency, (*ii*) characteristics of hardware low-power states, such as power consumption and (*iii*) whether there are pending tasks for the device, that is, tasks buffered either in the driver or in the device. While (*i*) and (*ii*) are conveniently available to the OS, Linux obtains (*iii*) by relying on drivers' correct behaviors in invoking runtime PM API, as shown in Figure 1(a). This reliance effectively makes the runtime PM driver-directed.

As SoC design life cycles tighten and the software/hardware stack bloats, we consider driver-directed PM to be harmful. We argue for an architectural overhaul as shown in Figure 1(b): introducing a central OS component, or central PM agent, to relieve device drivers from their runtime PM responsibilities. To build a central PM agent, our key insight is that the above information (*iii*) can be made available without the help of device drivers. We present two alternative ways to infer this information: using software to monitor device register access or adding one register bit to the device for exposing device busy/idle status.

Built atop the two alternative inference approaches, the central PM agent is effective. Under interactive workloads that frequently exercise the devices, it can automatically put the devices into the disabled mode over 77% of the time. Compared to manual, fine-tuned PM that already exists in a couple of Linux drivers, this disabled time is only 3.3 percentage points less. The central PM agent incurs very low overhead: in our FPGA-based implementation, the added hardware register introduces at most 34 gates as estimated by the synthesis tool; the device performance loss is unnoticeable.

To ease driver development under the *current* PM framework, we further develop a best-effort tool called PowerAdvisor. It observes the target driver's behaviors in test runs, and advises developers on where to add calls to runtime PM API in source code. We show that PowerAdvisor is effective: it discovers previously unknown locations for adding PM in a complicated display controller driver with 22 thousand lines of code.

In summary, this paper makes three major contributions:

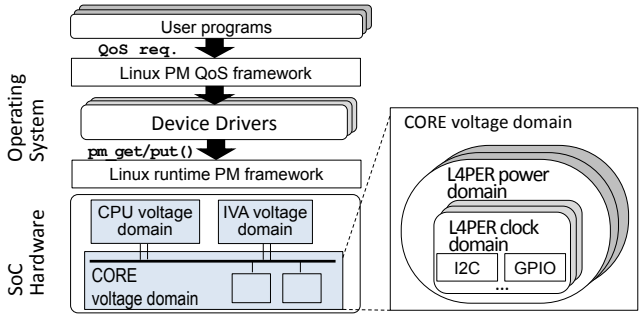


Figure 2: Hardware PM infrastructure in SoC (as exemplified by OMAP4) and the Linux PM frameworks

- We experimentally show that device runtime PM can be effectively done outside of drivers, either using a software approach based on existing hardware, or using software with small hardware support.
- A complete design and implementation of the central PM agent, whose performance is comparable with manual runtime PM code in Linux device drivers.
- PowerAdvisor, a tool that effectively simplifies the development of runtime PM in drivers under the current Linux framework.

All source code and trace data used in this paper can be downloaded from `http://www.recg.org`.

2. Background

Other than a powerful, multicore CPU, a modern SoC integrates tens of hardware devices from computational devices like GPU and face detection integrated circuit, to controllers for I/O like I2C and display. These devices are connected together via on-chip interconnects, and expose functions to the CPU via their registers. As shown in Figure 2, device runtime PM on SoC is a collaboration between hardware and software.

2.1 Hardware PM Infrastructure

We next sketch the hardware support for PM available on SoCs. Although our descriptions are based on our understanding of the TI OMAP4, a popular mobile SoC with abundant public information [2], we observe similar hardware support on other SoCs [1, 3, 4, 8–12].

A Hierarchy of Domains Device is the basic unit of power management from the perspective of software. A device can be configured by software to be in either ENABLED or DISABLED mode: it is functional only when it is ENABLED, and when DISABLED, it stops receiving clocks to conserve energy. Note that software, e.g., device drivers, only decides when to disable a device, leaving the rest of the power management decisions to hardware. To save chip area, devices on an SoC may share clocks and power supplies. This effectively organizes all devices into a hierarchy of domains.

Table 1: Device drivers of SoC families and the delay (in months) between the driver initial release and the implementation of runtime PM. Asterisks indicate the absence of fine-grained runtime PM at the time of writing (Jan. 2015).

Driver	Delay	Driver	Delay
OMAP UART	12 [13]	Exynos Keypad	18 [14]
OMAP WDT	22 [15]	Exynos USB	14 [16]
OMAP USB	45 [17]	OMAP GPIO	32 [18]
i.MX SD Ctrl	37 [19]	i.MX I2C	9 [20]
Tegra SD Ctrl	48 *	i.MX SPI	31 [21]

- A *clock domain* is a group of devices sharing the same clock source. A domain-level clock gating shuts down the clock source, putting the clock domain as INACTIVE and resulting in further energy saving.
- A *power domain* encompasses one or more clock domains. Its devices are powered by the same power rails controlled by the same switch. The domain could either be ON, RETENTION (a subset of transistors are on to preserve hardware state) or OFF (all transistors are off and hardware state is lost).
- A *voltage domain* encompasses one or more power domains. Its devices share the same voltage source controlled by the same regulator. It can be either ON, SLEEP (supplying regular voltage, but limited current), RETENTION (supplying minimum voltage for preserving hardware state) or OFF (voltage drops to zero).

For these domains, software only chooses the target low-power states (e.g., RETENTION, OFF); hardware decides when to perform the actual state transitions, as discussed below. Notably domains introduce inter-device dependence in runtime PM, because for a domain to enter a low-power state, all its devices must be disabled.

Global Power Manager Global power manager is a special on-chip hardware device that coordinates devices and domains in performing power state transitions. It sets a domain to a low-power state if all encompassed devices of the domain have been configured by software as DISABLED; it brings the domain back to a high-power state if any encompassed device is configured by software as ENABLED. The manager is always on. Even when the entire SoC has been suspended, it listens for external events and wakes up the SoC accordingly.

2.2 Linux Support for Device Runtime PM

Linux, like other OSes, plays two key roles in device runtime PM. Note that the Linux community often abbreviates device runtime PM as *runtime PM*. We will respect the convention when describing the related Linux API.

First, the *Linux runtime PM framework* provides generic set of API for device drivers to track the number of concurrent users of a device in the form of a per-device reference counter. When the reference counter drops to zero, the framework calls a driver callback to set the device to DISABLED. Built under this framework, a device driver should never directly change device modes. Instead, it is responsible for making runtime PM calls, `pm_get()` and `pm_put()`, which increase and decrease the reference counter respectively. In a well-designed driver, the reference counter will reach zero whenever there is no pending task for the device, neither buffered inside the driver nor in the device.

Second, the *PM QoS frameworks* allow users to express QoS requirements (e.g., wakeup latency) to be met by the OS. The device driver is responsible for mapping such QoS requirements to driver parameters, such as timeout value before disabling the device, or hardware configurations, such as the target low-power state of the encompassing domain.

This work aims at replacing the Linux runtime PM framework but retains compatibility with the PM QoS frameworks.

3. Driver-directed PM Considered Harmful

As discussed above, the current Linux runtime PM is essentially driver-directed: its correctness and efficiency fully depend on a driver properly maintaining its reference counter. For modern SoCs, we consider this approach harmful. In this section, we provide three reasons and elaborate upon each with a real-world case.

Runtime PM often supported after long delay. Driver developers almost always consider functionality as their first priority, leaving PM as an afterthought. While many drivers keep receiving new functionality over time, they get stuck for a long time with preliminary, coarse-grained PM. Proper PM support, if any, appears much later than the driver's initial release.

Real-world Case: Samsung Exynos SPI controller

The SPI controller driver for the Exynos family had preliminary PM code for 25 months after its first commit [22]. The preliminary PM, as shown in Listing 1, is very coarse-grained and saves no energy at run time. It simply enables the SPI controller in `probe()`, which is invoked during system boot, and disables it in `remove()`, which is invoked during system shutdown, keeping the SPI controller on as long as the CPU is on. The much delayed patch shown in Listing 2 fixed the problem with a finer-grained PM, which only keeps the controller enabled for configuration and transmission tasks. More cases are listed in Table 1.

Complex drivers make it hard to do runtime PM. Many devices on modern SoCs are complex. So are the drivers that harness the hardware. The complexity makes it hard for developers to reason about how a driver works and write correct PM code accordingly.

Real-world Case: OMAP4 display controller driver

The display controller in modern SoC is notoriously complex. The OMAP4 technical reference manual [2] dedi-

```
1  int s3c64xx_spi_probe(platform_device
       *pdev)
2  {
3    /* allocate controller resources...*/
4    pm_runtime_get_sync(dev);
5    /*initialize the controller...*/
6  }
7
8  int s3c64xx_spi_remove(platform_device
       *pdev)
9  {
10   /* deinitialize controller...*/
11   pm_runtime_put(dev);
12   /* free controller resources...*/
13 }
```

Listing 1: Preliminary PM that has existed in the Exynos SPI driver for more than two years.

```
1  void s3c64xx_spi_work(work_struct *work)
2  {
3    pm_runtime_get_sync(dev);
4    while (!list_empty(queue)) {
5      /* transmitting message... */
6    }
7    pm_runtime_put(dev);
8  }
9
10 int s3c64xx_spi_setup(spi_device *spi)
11 {
12   pm_runtime_get_sync(dev);
13   /* set up SPI, like tx rate... */
14   pm_runtime_put(dev);
15 }
```

Listing 2: Hand-tuned PM in the Exynos SPI driver after patching [22].

cates 565 pages to the display controller. The corresponding Linux driver consists of 22K SLoC, featuring extensive asynchronous execution (e.g., bottom-halves for completing frame composition) and tens of callbacks. Not surprisingly, the Linux driver only has preliminary power management, leaving the controller ENABLED as long as the screen is on. In our own attempts to patch the driver with finer-grained PM, we found it very difficult, if not impossible, to manually identify where in the driver source code to add runtime PM calls while still keeping the reference counter balanced in various interleavings of execution paths.

Inter-device dependence amplifies the impact of bad drivers Because the state of a domain depends on the state of all its encompassed devices (§2.1), one device that is mistakenly left on will prevent the entire domain from entering a low-power state, ruining the PM efforts of other drivers.

Real-world Case: OMAP4 UART controller

During one entire year after its first release, the OMAP4 UART controller driver did no power management, keeping the whole L4_PER power domain on as long as the CPU was on [13]. Although the ENABLED UART controller consumes a relatively small amount of power, the L4_PER domain drains 17 mW more power for not being able to enter the default low-power state RETENTION, leading to significant runtime inefficiency.

4. Fundamental PM Information

To fix the problems caused by driver-directed PM, we identify information that is essential to device runtime PM. Note that much research focuses on optimizing PM policy by exploiting complex workloads information [23–25]. Unlike these research works, the policy of Linux PM is simple, as has been discussed in §2.2. It requires the following three pieces of information:

(*i*) The QoS requirements supplied by users, such as wakeup latency.

(*ii*) Specifications of power states, including their power consumption, and the latency and energy consumption for transition between states.

(*iii*) Whether a device has pending tasks: only after a device has finished *all* pending tasks that are buffered in the driver and in the device, can the device be DISABLED. When there is a new pending task for a disabled device, the device needs to be enabled. Note that the functionality of a device is not broken if it is disabled after finishing *one* task and enabled before handling the next one, as long as the driver preserves and then restores the device context, respectively. However, this incurs unnecessary overhead due to power-state transition. Therefore, a good PM policy keeps the device enabled until it finishes *all* pending tasks.

We observe that (*i*) is provided by the user of the device, e.g., user-space software or other dependent device drivers. (*ii*) is static information and is available offline, either from the vendor or by profiling.

The Linux PM infers (*iii*) from the value of the reference counter (whether it is zero), and relies on the driver to properly invoke runtime PM API to maintain the reference counter. This approach incurs high development burden [26] and is error-prone as has been discussed in §3.

Our key insight is that information (*iii*) can be made available without device drivers' efforts. We next demonstrate that it can be inferred by monitoring memory access, or with small modification to device hardware.

5. Pending Task Inference

As discussed above, knowing if a device has pending tasks is key to its runtime power management. In this section, we present two alternative ways to acquire this knowledge outside device drivers, i.e., without device drivers' support: (*i*) a software-only approach that infers the knowledge by monitor memory access, and (*ii*) a small modification, a register bit, to device hardware that exposes whether the device is busy or idle.

5.1 Software-based Inference

The software approach infers if there are pending tasks by monitoring device register access initiated by the CPU. More specifically, it is based on the following two insights:

1. If CPU has not accessed a device's registers for a period of time longer than a threshold $T_{threshold}$, it is safe to assert that the device has no pending task.

   Taking I2C transmission as an example: the device registers are frequently accessed until the task is completed. In preparing sending a message, the CPU writes to the controller's registers for configuring speed mode, message length, etc. When the message is being transmitted, the controller frequently interrupts the CPU to provide updates, e.g., FIFO status, and the CPU accesses the controller's registers to examine and clear the interrupts.

2. On ARM-based SoCs, the memory protection mechanism can be used to effectively detect device register access. This is because ARM maps all device registers in the global physical address space. Therefore, CPU's accesses to device registers will go through MMU, and can be captured as memory exceptions when required.

The choice of $T_{threshold}$ We face a trade off in choosing $T_{threshold}$: with a smaller value of $T_{threshold}$, the software approach can infer in a more timely manner if a device has pending tasks and can capture shorter periods in which a device has no pending task. Thus, a smaller $T_{threshold}$ leads to more aggressive PM; however, to avoid false report of no pending task, $T_{threshold}$ needs to be greater than the largest possible interval between register accesses when a device has pending tasks. According to our observation of mobile SoCs, the largest interval appears when a DMA transaction is in progress. Given that the typical mobile SoC memory bandwidth is a few hundred MB/s to a few GB/s, we choose $T_{threshold}$=100 ms which is much larger than the time for a 4 MB transfer, a typical maximum size of contiguous memory allocation in Linux. We will evaluate our choice in §7.1.

Applicability of Insight 1 Insight 1 applies to most, but not all types of devices. The deciding factor is whether, for a given device, the length of a processing period without register access is bounded such that Insight 1 holds. For I/O devices the length of such periods is bounded by the longest DMA transaction as discussed above. For accelerators with periodic tasks, the length of such periods is bounded as well: for example, the OMAP4 face detection device interrupts the CPU upon finishing each frame, which happens every 33ms. However, this is not the case for complex computational units, such as GPU or DSP, whose processing duration can be unbounded in theory.

Limitations Although effective and immediately deployable, we recognize that the software approach has the following limitations: (*i*) it is less aggressive than the PM code in device drivers, as the device always lingers in ENABLED after there has been no pending task for $T_{threshold}$, which is lower bounded by the largest register access interval; (*ii*) it incurs memory exceptions, which comes with overhead, though small, as evaluated in §7.1.

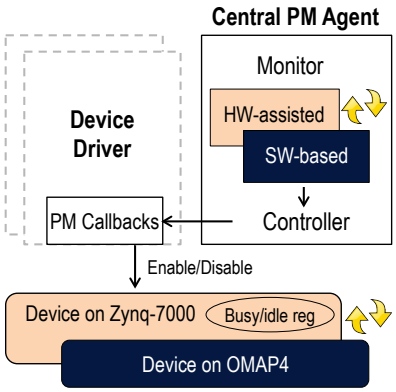


Figure 3: The structure of the Central PM agent, a concrete design following the conceptual model shown in Figure 1(b). The monitor employs one of the two ways for pending task inference, which are prototyped on two respective SoCs.

5.2 Hardware-assisted Inference

In order to avoid the limitations of the software approach, we find that small hardware modifications will suffice: a busy/idle register bit per device.

By polling this register bit periodically, the OS can estimate how long a device has been idle and then infer whether a device has pending tasks. Note that even if a device has been idle for only a few milliseconds, the OS can be certain that the device has no pending task because a device always starts processing pending tasks immediately after finishing the current one. Hence, by applying a small polling period, the OS can infer if a device has pending tasks in a timely manner, which enables more aggressive PM than the software approach in §5.1 does. Moreover, the polling does not incur memory exceptions and is thus more lightweight.

The modifications can be easily implemented on modern SoC hardware. A device already possesses the knowledge on whether it is busy processing a task or not: in its implementation, a device hardware is typically designed as a state machine using a hardware description language. Among all possible states, a subset represent the device being idle. Therefore, the busy/idle register is essentially a mapping from these states to a single bit.

6. Central PM Agent

With the pending task information inferred, we can realize device runtime PM of all applicable devices in a single kernel module, called *central PM agent*. The central PM agent relieves driver developers from reasoning where to insert runtime PM calls in driver source code (§2.2). In this section, we first describe the overall structure of the central

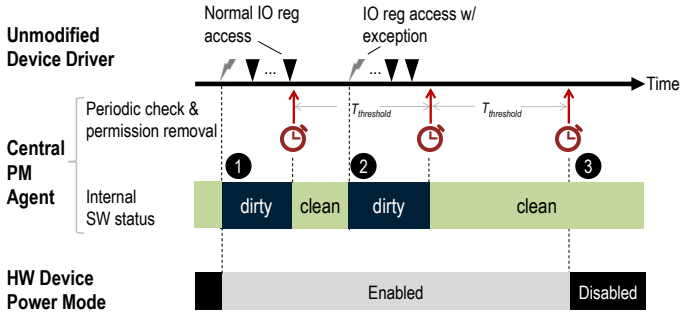
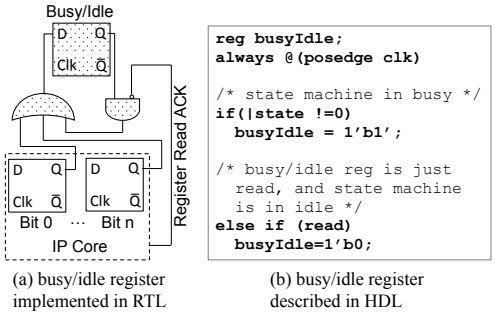


Figure 4: An example timeline of central PM agent execution, which infers the existence of pending tasks using the software-based approach.



(a) busy/idle register implemented in RTL

(b) busy/idle register described in HDL

Figure 5: Example implementation of the one-bit busy/idle register, (a) in several tens of gates and (b) in several lines of code in Verilog.

PM agent and then how the software and hardware-assisted solutions for pending task inference fit in the design.

6.1 Overall Structure

As shown in Figure 3, the central PM agent consists of two major components: the monitor and the controller. The monitor infers whether a device has pending tasks, which can be implemented in software on top of the existing hardware or with the assistance of the busy/idle register bit. Based on the monitor output, the controller calls the PM callbacks provided by the driver to set the device to ENABLED/DISABLED mode.

6.2 Software Pending Task Monitor

The software pending task monitor is based on the insights presented in §5.1: a device has no pending task if CPU has not accessed its registers for $T_{threshold}$. Every $T_{threshold}$ period, the monitor checks if a device's registers have been accessed since the last check, i.e., in the past $T_{threshold}$. In order to do so, the monitor maintains a per-device software status: `dirty` means that CPU has accessed the device's registers since the last check; `clean` otherwise.

Figure 4 illustrates how the central PM agent works for a device through an example timeline. During initialization, the monitor sets the software status to `clean` and sets up a check timer that will fire every $T_{threshold}$ interval.

Every time the timer fires, the monitor does two things. First, it checks the software status: `clean` (❸) indicates that none of the device's registers have been accessed in the past $T_{threshold}$. Based on Insight 1 in §5.1, the monitor concludes that the device has no pending task, and notifies the controller to disable the device. If the status is `dirty`, the monitor simply sets it to `clean`.

Second, the monitor invalidates the page table entries that correspond to the device's registers, so the next access to any of these registers will trigger memory exception, which will be handled by the central PM agent. This is based on Insight 2 in §5.1.

A memory exception occurs upon the first access to any of the device's registers since last check. The monitor handles the exception and sets the software status to `dirty`, indicating that an access has occurred. Then it checks the device mode: if DISABLED (❶), the monitor notifies the controller to enable the device; otherwise, if the device is already ENABLED (❷), the monitor takes no action. Finally, it validates the page table entries to allow further register access to pass through without incurring any exception.

Implementation on OMAP4 SoC We implement the central PM agent based on the above monitor design as a kernel module on the OMAP4 SoC. We choose OMAP4 because it has abundant public information.

6.3 Hardware-assisted Pending Task Monitor

We add the busy/idle register bit to existing devices, so that (*i*) the runtime PM can be more aggressive and (*ii*) the design of the monitor is simplified and incurs less overhead. We next describe how we design the busy/idle register and revise the design of the monitor accordingly.

Mapping device state machine to busy/idle register As discussed in §5.2, device hardware is described as state machines, with a subset of the states representing the device being idle. We design the semantics of the busy/idle register as follows:

The busy/idle register is set to busy when the state machine leaves the idle states; it resets to idle immediately after a read if the state machine is in an idle state when the read happens. Thus, if the register shows busy, it does not necessarily mean the device is busy at the moment. Instead, it indicates the device has been busy at least once since the last read of the busy/idle register. On the other hand, if the register shows idle, the device must have been idle since the last read of the busy/idle register.

Because the logic of a busy/idle register re-uses the state machine in the hardware, it requires very little hardware resources and development effort to implement. As the conceptual design in Figure 5(a) shows, aside from a busy/idle

Table 2: The cost of adding the busy/idle register. Development time is in man-hours. *unknown because Xilinx specific FPGA primitives cannot be synthesized to ASIC.

Module	Development Effort		FPGA Resources		ASIC Resources
	LoC	Time	LUTs	Registers	Gates
Xilinx I2C	93 (+1.2%)	12	16 (+3.8%)	8 (+2.4%)	N/A*
Opencores SPI	15 (+6%)	5	1 (+1.3%)	1 (+1.5%)	15 (+1.1%)
Opencores I2C	20 (+2%)	10	4 (1.8%)	1 (+0.6%)	34 (+1.6%)

register bit, the design merely requires several gates to extract the needed information from the state machine. Figure 5(b) shows that several lines of Verilog code suffice to describe the logic of the busy/idle register.

Using the busy/idle register Instead of triggering memory exceptions, the monitor polls the busy/idle register periodically to estimate how much time has elapsed since the completion of the most recent task. Based on this information, the monitor infers whether the device has pending tasks. As discussed in §5.2, a smaller polling period enables more aggressive PM. However, frequently polling the busy/idle register wastes CPU cycles and energy. In practice, we expect that a polling interval of tens of milliseconds is aggressive enough.

Implementation on Zynq-7000 SoC We choose Zynq [27] for using its on-chip FPGA to prototype the busy/idle register. Moreover, the FPGA has dedicated clocks that can be gated in software without affecting the rest of the system. We add a busy/idle register to an Xilinx I2C controller [28], and instantiate the modified I2C controller in the FPGA. We run the central PM agent on Zynq's CPU. As shown in Table 2, the development effort in modifying the I2C controller is small. The extra FPGA resources and the estimated number of extra gates required for ASIC implementation are also small. Note that our modification is largely unoptimized and the resource usage should be read as upper bounds.

In order to further experiment with the busy/idle register, we implement it by modifying an Opencores I2C controller [29] and an Opencores SPI controller [30] written in Verilog. As shown in Table 2, the involved development effort and extra resources are similarly small.

7. Evaluation

In the evaluation, we experimentally answer the following questions:

1. Can the central PM agent effectively save power, as compared to manual PM?
2. What is the associated runtime overhead introduced by the central PM agent?
3. Can the central PM agent correctly use the added busy/idle register?

7.1 Software Central PM

We evaluate the software central PM agent on Pandaboard Rev B2, which employs OMAP4460 SoC. It runs Linaro Android release 13.10. The kernel version is 3.2, the latest kernel that supports Android on OMAP SoC. Its driver code base receives contributions regularly from TI and Google.

7.1.1 Methodology

We use the central PM agent to automatically control four widely used and representative devices: the multimedia card (MMC) controller, the secure digital input/output (SDIO) controller, the I2C controller, and the display controller (DISPC). We use a trace collected from interacting with the Pandaboard to test the central PM agent. The trace is of 600 seconds and contains 48 user input events, including reading emails with the Android email application and browsing the web with the default Android browser. These activities exercise the drivers of the four devices extensively.

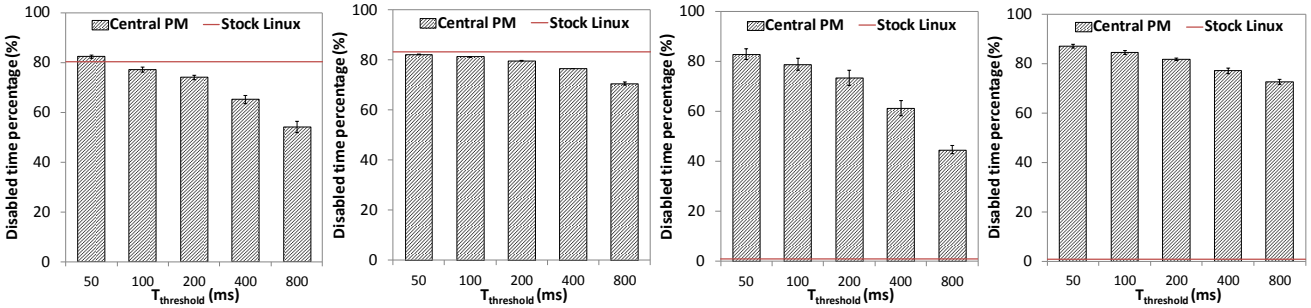
In running the benchmark for each device, we measure the total length of DISABLED periods. We run the benchmark with the central PM agent, and with the existing PM code if there is any.

7.1.2 Effectiveness of Central PM Agent

We show that the central PM agent is able to manage devices as effectively as the hand-tuned PM code in the MMC controller and I2C controller drivers: the difference in the disabled time is within 3.3 percentage points (with $T_{threshold}$=100 ms). Moreover, the central PM agent provides PM to the SDIO controller and DISPC drivers that lack PM code. We next describe the results for the four devices in detail.

MMC controller. On many embedded systems, the CPU accesses the root filesystem through a MMC controller. The Linux MMC controller driver comes with hand-tuned PM code. To examine how well the central PM agent works for it, we disable the PM code and run the benchmark, which generates extensive file operations (e.g., saving email attachment, application launch) to exercise the MMC controller driver. As shown in Figure 6(a), when $T_{threshold}$ is 100 ms, the central PM agent keeps the MMC controller in DISABLED for 77.2% of the time, which is only 3.3 percentage points less than the hand-tuned PM. If we aggressively reduce $T_{threshold}$ to 50 ms, the disabled time under the control of the central PM agent is actually 1.9 percentage points more, as the hand-tuned PM chooses to disable the controller after a 100 ms idle timeout.

I2C and SDIO controllers. The I2C and SDIO controllers on Pandaboard together bridge the CPU with the Wi-Fi interface. The I2C controller driver comes with hand-tuned PM; the SDIO driver does not, leaving the SDIO controller always on as long as the Wi-Fi interface is enabled, even when no data is being transmitted. The benchmark exercises



(a) MMC Controller (b) I2C Controller (c) SDIO Controller (d) Display Controller

Figure 6: Device DISABLED time with different $T_{threshold}$, with the default being 100 ms in this paper. The Linux stock drivers in (a) and (b) come with hand-tuned PM code while that of (c) and (d) does not. In (c) and (d), we have slightly shifted up the lines for Linux stock drivers to make them visible since they are 0. With $T_{threshold}$=100 ms, the central PM agent saves at most 17 mW by disabling (c) the SDIO controller, and 10 mW by disabling (d) the display controller.

the two drivers with rich network activities generated from web browsing and email fetch. The results are shown in Figure 6(b) and Figure 6(c). When $T_{threshold}$ is 100 ms, the central PM agent keeps I2C controller DISABLED for 81.2% of the time , which is only 2.1 percentage points less than the hand-tuned PM; the central PM agent keeps SDIO controller DISABLED for 78.8% of the time.

DISPC. The display controller, or DISPC, overlays multiple rendered frames into a final buffer that is co-located with the video output interface, HDMI in our setup. Its driver comes with no runtime PM, keeping DISPC ENABLED as long as the display is on, missing the power-saving opportunity when the displayed image is still and DISPC has no task. The results are shown in Figure 6(d). When $T_{threshold}$ is 100 ms, the central PM agent keeps the DISPC DISABLED for 84.5% of the time.

Impact of $T_{threshold}$. When $T_{threshold}$ is larger than 100 ms, the time in which the central PM agent keeps the four devices in DISABLED state decreases noticeably. With a larger $T_{threshold}$, a device not only lingers in a high-power state for longer time after the last task is finished, but also misses power-saving opportunities for periods without pending tasks but shorter than $T_{threshold}$. We have also tried a reduced $T_{threshold}$ of 50 ms, which at most leads to a trivial 5.2% increase of device DISABLED time. We believe this is because most periods without pending tasks are longer than 50 ms. Hence, we choose $T_{threshold}$=100 ms as the default value, which is safe as stated in §5.1.

7.1.3 Estimated Overall Energy Saving

We have shown that the central PM agent provides effective runtime PM to drivers that lack PM code, i.e., the SDIO controller and DISPC. We next show that the resulting energy savings are significant.

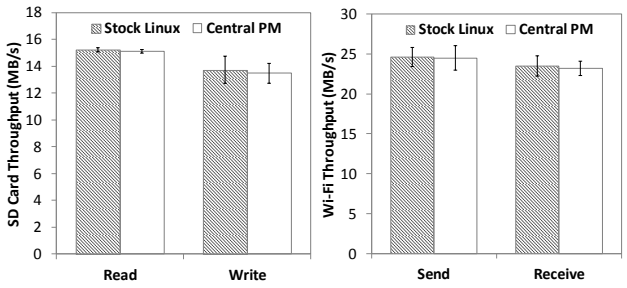
We measure the power consumed by the two devices by physically sampling current on power rails [31]. By disabling the DISPC, the central PM agent saves 10 mW. Although the SDIO controller itself consumes very little power, disabling it saves 17 mW because an enabled SDIO controller blocks the encompassing power domain from entering RETENTION.

The resulting energy savings are significant, estimated from smartphone daily usage reported in LiveLab [32]. For the SDIO controller, the stock driver keeps it always ENABLED as long as the user has not manually switched off the Wi-Fi; compared to the stock driver, the central PM agent saves up to 71.4 mWh daily which extends the standby time by 2.4 hours. For the DISPC, the stock driver keeps it always ENABLED as long as the screen is on; compared to the stock driver, the central PM agent saves around 18.5 mWh daily, which extends the standby time by 0.6 hours. Given that a smartphone usually lasts for 0.7 to 2 days with average use [33], extending the standby time by a total of 3 hours is a significant gain. We note the LiveLab data was collected about four years ago; the gain would be more if the user uses the smartphone more because runtime PM saves power when the system is in use. We further estimate that the relative gain may be considerably higher on systems like Google Glass that use a similar SoC but have lower overall power consumption due to smaller displays.

7.1.4 Overhead of Central PM Agent

The central PM agent incurs very small overhead at run time, which mainly comes from the memory exceptions introduced by software inference. By using CPU performance counters, we measure that each memory exception takes around 2500 cycles, which is 8 µs if the CPU is at the lowest frequency 300 MHz. Note this overhead occurs only once for each device every $T_{threshold}$, which is typically 100 ms.

To understand the impact of this overhead, we stress the SD card (backed by the MMC controller) and the Wi-Fi interface (backed by the SDIO and I2C controllers) and measure the performance loss due to the central PM agent.



(a) SD Card Throughput (b) Wi-Fi Throughput

Figure 7: Performance of devices managed by central PM agent under stress tests, as compared to the stock Linux.

All results are averaged over 10 iterations. To stress the SD card, we invoke `dd` to read/write a 100MB file. We force dropping pagecache before each read test and use the `fdatasync` option in the write test. To stress the Wi-Fi interface, we use *iperf* [34] to measure the maximum TCP throughput. We run each iteration for 10 seconds, and manually interleave two types of runs (with and without the central PM agent) to exclude the impact of possible wireless signal fluctuation. As shown in Figure 7, the throughput loss due to the central PM agent is smaller than the standard deviations and thus negligible, for both the SD card and the Wi-Fi interface.

7.2 Hardware-assisted Central PM

We prototype the hardware-assisted central PM agent on the Zynq SoC. As described in §6.3, we instantiate the modified I2C controller on Zynq's FPGA. We then develop test software running on Zynq's CPU to validate that the busy/idle register has the expected behavior. We further connect Zynq with an external accelerometer [35] over the I2C bus. Managed by the central PM agent, Zynq can communicate with the accelerometer correctly, showing that the central PM agent supported by our busy/idle register is working without breaking any device functionality. In addition, our effort in bringing up the central PM agent on Zynq is small.

8. PowerAdvisor for PM Code Suggestion

So far we have argued for an overhaul to the Linux PM. Can we simplify driver development under the *current* runtime PM framework without such an overhaul? To answer this question, we build PowerAdvisor, a software tool that suggests where to add runtime PM calls in existing driver source code. Consisting of an instrumenter and an offline analyzer, the tool analyzes historical execution trace and makes suggestions accordingly.

Figure 8 shows the workflow of PowerAdvisor. A developer first instruments the driver with the tool. In test runs, the developer exercises the instrumented driver with various user workloads. The instrumented driver will generate a trace; the analyzer then examines the trace and outputs a list of source locations where `pm_get()` or `pm_put()` shall be inserted.

8.1 Division of Responsibility between PowerAdvisor and Developers

PowerAdvisor is best-effort. It greatly simplifies, but does not completely relieve driver developers from, PM efforts. It provides the following guarantees to the developer. If the driver were patched with the suggested runtime PM calls, in the test run from which the trace has been generated:

G1 During any no-pending-task period (that is, any period longer than $T_{threshold}$ during which no device register access occurs, as described in Insight 2 in §5.1), the PM reference counter would remain as zero, implying that the hardware device remains DISABLED.

G2 At the moment of any device register access, the reference counter would be above zero, implying that the hardware device is ENABLED at that moment.

Given the guarantees, the developers need to further reason about the following two questions:

Will the added PM code break device functionality? PowerAdvisor cannot guarantee that the suggested PM calls will not break device functionality. This is due to the incompleteness of the tool's knowledge about device internals: although the tool can safely assume that no task is being processed in a device during no-pending-task periods, at an arbitrary moment *outside* of such periods, it has no visibility into the device to decide if a task is being processed there. Note that the tool cannot assume that the device has to be always ENABLED outside of no-pending-task periods, a constraint so strong that it suppresses useful suggestions.

Will the added runtime PM calls be effective in future executions? PowerAdvisor makes suggestions purely based on historical observations. The above two guarantees only apply to the trace it has observed; developers need to reason if the PM code is effective under different execution paths.

We believe it is feasible to reason about the two questions above in practice. For the first question, as the number of suggested PM calls is usually moderate, reasoning about whether these calls break device functionality is often tractable. For the second question, moderately extending the length of test runs and increasing workload variation is effective. For instance, a test run longer than a few minutes is often sufficient for the tool to reproduce hand-tuned PM. Furthermore, symbolic execution tools, such as KLEE [36], are able to produce high-coverage test runs. Because of these, we believe PowerAdvisor as a best-effort tool is useful in practice. We will show experimental evidences in §8.3.

8.2 PowerAdvisor Internals

For suggesting runtime PM calls, PowerAdvisor considers the following candidate locations:

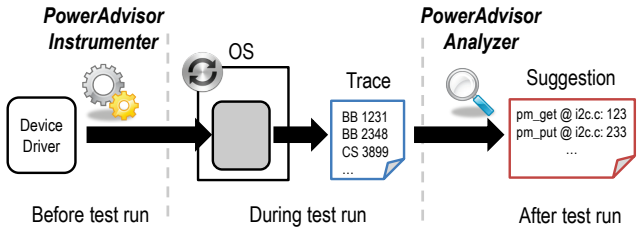


Figure 8: The workflow of PowerAdvisor

- `pm_get()`: the *start* of any basic block that contains device register access; the location *before* call sites that may lead to device register access.
- `pm_put()`: the *end* of any basic block that contains device register access; the location *after* call sites that may lead to device register access.

The choice of these candidate locations is based on the following rationale. Intuitively, the tool should consider the start and the end of basic blocks that contain device register access to fulfill G1 and G2 in §8.1. Furthermore, as one of such basic blocks may be included in various execution paths, we want the tool to be able to insert runtime PM calls that only affect a subset of these paths. On the other hand, we need to limit the number of candidate locations and make the resulting problem tractable. As a trade-off, we choose to instrument all call sites that may lead to such basic blocks.

8.2.1 Inserting Tracepoints

At compile time, the driver is firstly compiled as Intermediate Representation (IR). The instrumenter's job is to insert trace points into the driver IR to collect runtime information about candidate locations for PM calls.

To do so, the instrumenter first marks basic blocks that contain device register access, and call sites that may lead to the functions that contain device register access. This is a fairly easy job, as Linux kernel uses dedicated macros to encapsulate device register access.

With the markings, the instrumenter inserts trace points at four types of code locations: the start and the end of each marked basic block, and the locations before and after each marked call site. The instrumenter statically assigns each tracepoint a unique identifier that can be retrieved at run time.

8.2.2 Collecting Trace

As shown in Figure 8, in a test run, whenever a tracepoint is reached by the control flow, it appends its identifier into an global trace buffer. In addition, an entry for a no-pending-task period is inserted in the same buffer when the interval between two consecutive tracepoints is larger than $T_{threshold}$. This is done by a small piece of code added to the kernel.

After the test run, the resulting trace is a sequence of tracepoint identifiers delimited by no-pending-task periods.

8.2.3 Analyzing Trace

Overall, the PowerAdvisor analyzer looks for legal PM calls that, when applied to the collected trace, satisfy the guarantees in §8.1. The analyzer translates the problem into a Satisfiability Modulo Theories (SMT) problem, with the linear integer arithmetic as the background theory. It establishes SMT constraints based on the trace, and invokes the Z3 SMT solver [37] to identify the PM calls that should be inserted.

We next briefly describe the SMT problem in an informal way. All variables in the SMT problem correspond to the candidate PM call locations that have appeared in the trace, which are the union of four subsets: the start (S_{BB_GET}) and the end (S_{BB_PUT}) of the marked basic blocks, before (S_{CS_GET}) and after (S_{CS_PUT}) the marked call sites:

$$V = S_{BB\_GET} \cup S_{BB\_PUT} \cup S_{CS\_GET} \cup S_{CS\_PUT} \quad (1)$$

We define the possible values of the SMT variables as the delta applied to the reference counter by the corresponding PM calls.

$$\begin{aligned} \forall v \in S_{BB\_GET} \cup S_{CS\_GET}, v \in \{0, 1\} \\ \forall v \in S_{BB\_PUT} \cup S_{CS\_PUT}, v \in \{0, -1\} \end{aligned} \quad (2)$$

The analyzer further translates the collected trace into a set of SMT constraints. In order to do so, the analyzer first splits the trace into multiple subsequences delimited by no-pending-task periods, and generates constraints from each subsequence.

We next zoom in on one subsequence $Q = \{q_i\}_{i=1..N}$. According to our discussion of trace collection, each element of the subsequence is an appearance of one SMT variable in V. A SMT variable from V might appear one or multiple times in Q.

Based on Q, the analyzer asserts the following two sets of constraints that are directly mapped to the two guarantees G1 and G2 in §8.1. First, G1 requires the reference counter to be zero in the no-pending-task period following the subsequence. That is, all `pm_get()` and `pm_put()` in the subsequence are balanced.

$$\sum_{q_i \in Q} q_i = 0 \quad (3)$$

Second, G2 requires the device to be enabled when the control flow enters any basic block containing device register access, i.e., the PM reference counter to be larger than zero.

$$\forall j \in \{1...N\}, \text{if } q_j \in S_{BB\_GET}, \text{then } \sum_{i=1}^{j} q_i > 0 \quad (4)$$

Note that the analyzer generates the constraints (3) and (4) for all subsequences in the trace.

At last, the analyzer minimizes the number of introduced PM calls $C = \sum_{v_i \in V} |v_i|$.

In order to do so, the analyzer assigns different constants to C, from the smallest possible value $C = 2$ upward, and invokes the SMT solver on each resulting SMT problem. It stops when a solution is found.

Table 3: Driver traces used in evaluating PowerAdvisor. *#NPT stands for the number of no-pending-task periods.

Device	Trace			SMT Problem	
	Time (s)	#Entries	#NPT*	#Vars	#Constraints
MMC ctrl.	50.2	20,000	19	58	4,930
I2C ctrl.	33.9	5,000	13	32	1,480
SDIO ctrl.	24.4	5,000	78	14	1,708
DISPC	183.2	679,915	275	294	126,274

With the found solution, the analyzer maps the SMT variables with non-zero values to the corresponding source lines, and suggests that the developer insert PM calls there.

We note that the above SMT problem can be expressed as a Integer Linear Programming problem and solved more efficiently by an linear programming solver like CPLEX [38]. In our prototype of PowerAdvisor, we use the Z3 SMT solver because we are more familiar with it.

8.3 Evaluation of PowerAdvisor

We experimentally show that the PowerAdvisor can offer useful and correct suggestions. To do so, we study the drivers of the four devices used in §7.1. We generate driver traces by using the Android system to browse the web and navigate among multiple applications; for validating the suggested PM code, we replay the user trace described in §7.1 on the Android system.

Statistics of collected driver traces and resulting SMT problems are shown in Table 3. Note that the driver trace for the DISPC is much longer as its driver is significantly more complex than others.

As mentioned in §7.1, the drivers of the MMC and I2C controllers come with hand-tuned PM code, which has been removed by us before the test runs. Based on the driver traces, PowerAdvisor suggests PM calls identical to the hand-tuned code.

The drivers of the SDIO controller and DISPC come with no PM code. For the SDIO controller driver, PowerAdvisor suggests two pairs of `pm_get()` and `pm_put()`, which disable the SDIO controller for more than 85% of the time in the validation run. For the DISPC driver, whose complexity defeated our attempts of manually adding PM code, PowerAdvisor discovers a pair of `pm_get()`/`pm_put()` to be invoked when any interrupt handler is registered and unregistered, respectively. With the suggested PM code applied, the driver is able to disable the display controller for 90.5% of the time during the validation run. We further stress-test the driver with continuous animation on the screen. The driver runs for more than 48 hours without breaking the device functionality, and correctly disables the device in no-pending-task periods.

9. Related Work

Power management (PM) is critical not only to system efficiency but also to performance as we enter the dark silicon era [39]. We roughly categorize prior PM works into system suspension management and runtime PM. We also discuss model checking and code synthesis tools that are related to PowerAdvisor.

System suspension management. In system suspension, most SoC components (including CPU) are off, leaving on only the necessary circuitry for wakeup. Although effective and popular, system suspension challenges software to properly and timely drive hardware in and out of the suspension state. Android opportunistically suspends the entire SoC if no user interaction has occurred recently. In addition, it allows applications to override this policy by holding wakelocks. This relies on application developers for correct PM, leading to many power bugs, e.g., no-sleep [40] and sleep conflict [41], and has stirred up long debates in the Linux community [42]. Many have suggested ways to detect power bugs due to wakelock misuse, e.g., by detecting abnormal energy-hungry phases in applications [43], monitoring I/O bus traffic [41], and static analysis of source code [40]. Notably, the static analysis techniques used by Pathak et al. [40] only work well for programs having single entry/exit point. Unfortunately, device drivers usually have multiple entry/exit points, e.g., callbacks registered with various Linux frameworks and interrupt handlers. Moreover, the static analysis techniques only handle wakelocks that are not reference counted, while the Linux runtime PM framework indeed relies on reference counters. In contrast, our PowerAdvisor is based on execution trace generated from test runs and thus is not subject to the limitations discussed above.

Runtime power management. The goal of runtime PM is to reduce power consumption when the system is still in use.

Runtime PM for Computational Unit: A rich set of works focus on matching the performance of computational units, e.g., CPU or GPU, with the expectation of given workloads. Vertigo[6] uses a hierarchy of performance-setting algorithms to choose the CPU operating point for interactive applications. The Process Cruise Control framework [44] and the Koala framework [45] adjust the CPU performance for given workloads according to the workloads characteristics captured with performance counters. Pathania *et al.* adjust the performance point of CPU and GPU together according to 3D game workloads history to save energy [46]. Gupta *et al.* point out that the power consumed by uncore components, e.g., memory controller, should be considered in making runtime PM decisions for the CPU [47].

Runtime PM for Device: Device runtime PM aims at disabling individual devices that are not in use when the system is in use. Compared to system suspension, device runtime PM is finer-grained and can leverage more power-saving opportunities. If done properly, device runtime PM can bring

the power consumption by an SoC to as low as system suspension does [42] while allowing the system to remain responsive to external events. It is recognized by the Linux kernel community that device runtime PM is becoming increasingly important as more always-on applications emerge on personal computing devices [7].

The central challenge to device runtime PM is to turn off unused devices in a timely manner. This is difficult as the OS and the SoC hardware are complicated. Intel engineers report great engineering efforts in implementing runtime PM for their Medfield SoC. They also advocate hardware-supplied information for detecting pending tasks [26]. ICEM [48] integrates PM code into locks for device drivers. However, the approach only focuses on a specific class of device drivers ("shared drivers") which is uncommon in Linux for modern SoCs, according to our observation. Anand *et al.*[49] propose a new I/O API for applications to disclose hints for better device PM. Although sharing the similar goal of device runtime PM with such prior work, we seek to maintain the existing Linux API and are not restricted to a subset of devices. In a previous paper [50], we present a preliminary implementation and the results of the software central PM agent.

Formal methods and synthesis tools. PowerAdvisor (§8) is related to software tools that use formal methods for bug finding or code synthesis. However, these tools, when applied to PM, cannot make the developer's task much easier. Model checking tools like CBMC [51] and symbolic execution tools like KLEE [36] can be used to verify if a device is disabled when the control flow reaches given source code locations. However, it is up to the developers to manually identify those locations where the device must have no pending task. In theory, automatic device driver synthesis tools like Termite [52] can synthesize PM code given formal device specifications, either derived from the documentation or RTL (register transfer level) description of a device. However, it is unclear whether deriving such formal specifications is any easier, if not harder, than writing driver PM code.

10. Concluding Remarks

How far can hardware PM support go? We have shown that a small hardware modification (§5.2) can go a long way. However, shifting the runtime PM responsibility entirely to hardware is unwise as it may result in much higher hardware complexity: (*i*) each device must fully preserve its hardware context before hitting off, possibly by implementing a RETENTION state, instead of relying on the software to preserve the hardware context; (*ii*) each device must be capable of interpreting various PM QoS requests expressed by users.

How general is the central PM agent? As mentioned in §5.1, the central PM agent is applicable to an SoC device as long as it is (*i*) memory-mapped, (*ii*) with bounded register-access interval. Additionally, if an SoC device serves as a slave in a communication protocol, e.g., an I2C slave, then it has to be (*iii*) interruptible when in a low-power state. These conditions are true for most devices on ARM-based SoCs. In particular, to satisfy (*iii*), most devices rely on platform-specific hardware mechanisms, e.g., asynchronous wakeup or re-routing interrupt signal through GPIO. However, a device such as an Ethernet controller on a TV SoC lacks such hardware support as of now, making its runtime PM difficult in general.

Conclusion Device runtime PM requires the OS to disable devices that have no pending task in a timely manner. Currently, Linux places this burden on driver developers, who unfortunately implement runtime PM poorly in most cases. To address this issue, we relieve drivers from PM by building a central PM agent that automatically performs runtime PM, according to its observation and inference on whether there are pending tasks for the device. We propose two alternatives for performing the inference, one with existing hardware and the other with added hardware support. In addition, we demonstrate it is possible and useful to have a best-effort software tool that suggests where to add PM calls in driver source code, an approach reducing (but not eliminating) developers' efforts in reasoning about PM without structural changes to the OS.

Acknowledgments

The work was supported in part by NSF Awards #1054693, #1065506, and #1218041. The authors thank Aaron Carroll for injectevents [53], Jie Liao for helping process the LiveLab data, and Kevin Boos for helping revise the paper. The authors also thank anonymous reviewers for their useful feedbacks.

References

[1] NVIDIA, "NVIDIA Tegra K1 mobile processor technical reference manual," 2014.

[2] Texas Instruments, "OMAP4460 multimedia device technical reference manual," 2011.

[3] Samsung, "Samsung Exynos 5 Quad (Exynos 5250) RISC microprocessor user's manual," 2012.

[4] Texas Instruments, "Multicore DSP+ARM KeyStone II System-on-Chip (SoC) technical reference manual," 2013.

[5] M. Mehendale, S. Das, M. Sharma, M. Mody, R. Reddy, J. Meehan, H. Tamama, B. Carlson, and M. Polley, "A true multistandard, programmable, low-power, full HD video-codec engine for smartphone SoC," in *Digest of Technical Papers of IEEE Int. Solid-State Circuits Conf. (ISSCC)*, pp. 226–228, 2012.

[6] K. Flautner and T. Mudge, "Vertigo: automatic performance-setting for Linux," in *Proc. USENIX Symp. Operating Systems Design and Implementation (OSDI)*, pp. 105–116, 2002.

[7] R. J. Wysocki, "Power management in the Linux kernel: current status and future." http://events.linuxfoundation.org/sites/events/files/slides/kernel_PM_plain.pdf, 2013.

[8] NVDIA, "NVIDIA Tegra 4 4-plus-1 quad-core processors technical reference manual," 2013.

[9] Freescale, "i.MX 6Dual/6Quad applications processor reference manual," 2013.

[10] Intel, "Intel Atom processor Z36xxx and Z37xxx series datasheet," 2013.

[11] P. Choudhary and D. Marculescu, "Power management of voltage/frequency island-based systems using hardware-based methods," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 427–438, 2009.

[12] T. Hattori, T. lrita, M. Ito, E. Yamamoto, H. Kato, G. Sado, Y. Yamada, K. Nishiyama, H. Yagi, T. Koike, Y. Tsuchihashi, M. Higashida, H. Asano, I. Hayashibara, K. Tatezawa, Y. Shimazaki, N. Morino, K. Hirose, S. Tamaki, S. Yoshioka, R. Tsuchihashi, N. Arai, T. Akiyama, and K. Ohno, "A power management scheme controlling 20 power domains for a single-chip mobile processor," in *Digest of Technical Papers of IEEE Int. Solid-State Circuits Conf. (ISSCC)*, pp. 2210–2219, 2006.

[13] G. Raja, "OMAP2+: UART: add runtime pm support for OMAP-serial driver." http://www.spinics.net/lists/linux-omap/msg58443.html, 2011.

[14] M. Brown, "Input: samsung-keypad: implement runtime power management support." http://www.spinics.net/lists/linux-input/msg18796.html, 2011.

[15] P. Walmsley, "Watchdog: OMAP_wdt: add fine grain runtime PM." http://permalink.gmane.org/gmane.linux.ports.arm.omap/54608, 2011.

[16] V. Gautam, "DWC3/xHCI: enable runtime power management." https://lkml.org/lkml/2013/1/28/229, 2013.

[17] R. Quadros, "USB: implement runtime idling and remote wakeup for OMAP EHCI controller." https://lkml.org/lkml/2013/7/10/355, 2013.

[18] F. Balbi, "GPIO: OMAP: be more aggressive with pm_runtime." http://www.spinics.net/lists/linux-omap/msg64196.html, 2012.

[19] A. Dong, "MMC: sdhci-esdhc-imx: add runtime PM support." http://lists.infradead.org/pipermail/linux-arm-kernel/2013-October/208191.html, 2013.

[20] R. Zhao, "I2C: IMX: disable clock when it's possible to save power." http://lists.infradead.org/pipermail/linux-arm-kernel/2009-October/002455.html, 2009.

[21] S. Huang, "SPI: spi-imx: only enable the clocks when we start to transfer a message." http://lists.infradead.org/pipermail/linux-arm-kernel/2013-October/206797.html, 2013.

[22] M. Brown, "SPI/s3c64xx: implement runtime PM support." http://www.spinics.net/lists/linux-samsung-soc/msg08912.html, 2012.

[23] S. Irani, S. Shukla, and R. Gupta, "Online strategies for dynamic power management in systems with multiple power-saving states," *ACM Transaction on Embedded Computing Systems (TECS)*, vol. 2, no. 3, pp. 325–346, 2003.

[24] T. Simunic, L. Beniani, and G. De Micheli, "Event-driven power management of portable systems," in *Proc. Int. Symp. System Synthesis*, pp. 18–23, 1999.

[25] Q. Qiu and M. Pedram, "Dynamic power management based on continuous-time Markov decision processes," in *Proc. ACM/IEEE Design Automation Conf. (DAC)*, 1999.

[26] R. Muralidhar, H. Seshadri, V. Bhimarao, V. Rudramuni, I. Mansoor, S. Thomas, B. Veera, Y. Singh, and S. Ramachandra, "Experiences with power management enabling on the Intel Medfield phone," in *Proc. Linux Symposium*, 2012.

[27] Xilinx, "Zynq-7000 all programmable soc technical reference manual," 2014.

[28] Xilinx, "LogiCORE IP AXI IIC Bus Interface v2.0." http://japan.xilinx.com/support/documentation/ip_documentation/axi_iic/v2_0/pg090-axi-iic.pdf, 2014.

[29] R. Herveille, "Opencores project: I2C." http://opencores.org/project,i2c.

[30] R. Herveille, "Opencores project: simple SPI." http://opencores.org/project,simple_spi.

[31] eLinux.org, "PandaBoard Power Measurements." http://elinux.org/PandaBoard_Power_Measurements.

[32] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "Livelab: Measuring wireless networks and smartphone users in the field," *SIGMETRICS Performance Evaluation Review*, vol. 38, pp. 15–20, Jan. 2011.

[33] Verizon Wireless, "A smartphone is only as good as its battery life." http://www.verizonwireless.com/mobile-living/tech-smarts/smartphones-with-long-lasting-battery-life/, 2013.

[34] M. Gates, A. Tirumala, J. Ferguson, J. Dugan, F. Qin, K. Gibbs, and J. Estabrook, "Iperf: The TCP/UDP bandwidth measurement tool, version 2.0.5." https://iperf.fr/.

[35] Freescale, "Xtrinsic MMA8452Q 3-axis, 12-bit/8-bit digital accelerometer." http://www.freescale.com/files/sensors/doc/data_sheet/MMA8452Q.pdf, 2014.

[36] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. USENIX Symp. Operating Systems Design and Implementation (OSDI)*, pp. 209–224, 2008.

[37] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[38] IBM, "CPLEX Optimizer." http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[39] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proc. ACM/IEEE Int. Symp. Computer Architecture (ISCA)*, pp. 365–376, 2011.

[40] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proc. Int. Conf. Mobile Systems, Applications, and Services (MobiSys)*, pp. 267–280, 2012.

[41] A. Jindal, A. Pathak, Y. C. Hu, and S. Midkiff, "Hypnos: understanding and treating sleep conflicts in smartphones," in *Proc. ACM European Conf. Computer System (EuroSys)*, pp. 253–266, 2013.

[42] R. J. Wysocki, "Technical background of the Android suspend blockers controversy." lwn.net/images/pdf/suspend_blockers.pdf, 2010.

[43] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker, "eDoctor: automatically diagnosing abnormal battery drain issues on smartphones," in *Proc. USENIX Symp. Networked Systems Design and Implementation (NSDI)*, pp. 57–70, 2013.

[44] A. Weissel and F. Bellosa, "Process cruise control: Event-driven clock scaling for dynamic power management," in *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pp. 238–246, 2002.

[45] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser, "Koala: a platform for OS-level power management," in *Proc. ACM European Conf. Computer Systems (EuroSys)*, pp. 289–302, 2009.

[46] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra, "Integrated CPU-GPU power management for 3D mobile games," in *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pp. 40:1–40:6, 2014.

[47] V. Gupta, P. Brett, D. A. Koufaty, D. Reddy, S. Hahn, K. Schwan, and G. Srinivasa, "The forgotten 'uncore': on the energy-efficiency of heterogeneous cores," in *USENIX Annual Technical Conf. (ATC)*, pp. 367–372, 2012.

[48] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis, "Integrating concurrency control and energy management in device drivers," in *Proc. ACM Symp. Operating Systems Principles (SOSP)*, pp. 251–264, 2007.

[49] M. Anand, E. B. Nightingale, and J. Flinn, "Ghosts in the machine: interfaces for better power management," in *Proc. ACM Int. Conf. Mobile Systems, Applications, and Services (MobiSys)*, pp. 23–35, 2004.

[50] C. Xu, X. Lin, and L. Zhong, "Device drivers should not do power management," in *Proc. ACM SIGOPS Asia-Pacific Wrkshp. Systems (APSYS)*, June 2014.

[51] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176, 2004.

[52] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser, "Automatic device driver synthesis with Termite," in *Proc. ACM Symp. Operating Systems Principles (SOSP)*, pp. 73–86, 2009.

[53] N. FitzRoy-Dale and A. Carroll, *injectevents*. https://github.com/xaaronc/injectevents.