RICE UNIVERSITY

**Automated OS-level Device Runtime Power
Management**

by
Chao Xu

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE
Master of Science

APPROVED, THESIS COMMITTEE:

Dr. Lin Zhong, Chair
Associate Professor,
Electrical and Computer Engineering and
Computer Science

Dr. Joseph R. Cavallaro
Professor,
Electrical and Computer Engineering and
Computer Science

Dr. Moshe Y. Vardi,
Karen Ostrum George Distinguished
Service Professor in Computational
Engineering

Dr. Dan S. Wallach
Professor,
Computer Science and Electrical and
Computer Engineering

Houston, Texas
September, 2014

ABSTRACT

Automated OS-level Device Runtime Power Management

by

Chao Xu

Device drivers are responsible for disabling and enabling devices on a System-on-Chip (SoC) at runtime, a practice known as device runtime Power Management (PM). Based on the observations of drivers and their evolution, we consider it harmful to rely on drivers for device runtime PM.

We identify information essential to device runtime PM and show that it can be obtained without involving drivers, either by using a software inference approach atop existing ARM-based SoC, or more efficiently by adding one register bit to each device. We thus propose a structural change to the Linux runtime PM framework, replacing PM code in each driver with one kernel module called the central PM agent. We experimentally show that the central PM agent is as effective as hand-tuned driver PM code.

We also present a tool called PowerAdvisor, which analyzes traces generated from execution histories and suggests where to add PM code in driver source code. PowerAdvisor not only reproduces PM code that exists in stock drivers, but also correctly suggests PM code never known before.

Acknowledgment

My sincere gratitude goes to my advisor, Dr. Lin Zhong, for his steadfast support of my research. His enthusiasm and guidance has helped me become a better researcher.

I will forever be thankful to Dr. Felix Xiaozhu Lin, for showing me how to work with full passion in the face of adversity, for encouraging me to think boldly, and for teaching me bits and bytes about research. This thesis will not be possible without him.

I am also grateful to Yuyang Wang, for his contribution in the modification of hardware IPs. My thanks also go to Aaron Carroll and Jie Liao for helping improve the evaluation part of this thesis.

I also thank Kevin Boos for patiently revising my thesis and remarkably improving my technical writing skills. My presentation skills have significantly benefited from the inspiring guidance of Robert LiKamWa.

Last but not least, I would like to thank my parents, my group members and friends. I cannot imagine succeeding without their support.

Contents

Abstract	ii
List of Illustrations	vi
List of Tables	vii
1 Introduction	1
2 Background	5
2.1 Hardware PM Infrastructure	5
2.2 Linux Support for Device Runtime PM	7
3 Driver-directed PM Considered Harmful	9
4 Fundamental PM Information	13
5 Pending Task Inference	15
5.1 Software-based Inference	15
5.2 Hardware-assisted Inference	17
6 Central PM Agent	19
6.1 Overall Structure	19
6.2 Software Pending Task Monitor	19
6.3 Hardware-assisted Pending Task Monitor	22
7 Evaluation	25
7.1 Software Central PM	25

7.1.1	Methodology	25
7.1.2	Effectiveness of Central PM Agent	26
7.1.3	Estimated Overall Energy Saving	28
7.1.4	Overhead of Central PM Agent	29
7.2	Hardware-assisted Central PM	30
8	PowerAdvisor for PM Code Suggestion	32
8.1	Division of Responsibility between PowerAdvisor and Developers . .	32
8.2	PowerAdvisor Internals	34
8.3	Evaluation of PowerAdvisor	38
9	Related Work	40
10	Concluding Remarks	43
	Bibliography	46

Illustrations

1.1	An overview of this work	2
2.1	Hardware PM infrastructure in SoC (as exemplified by OMAP4) and the Linux PM frameworks	6
6.1	The structure of a concrete design of central PM agent	20
6.2	An example timeline of central PM agent execution	21
6.3	An example implementation of a busy/idle register	23
7.1	Device DISABLED time with different $T_{threshold}$	27
7.2	Performance of devices managed by central PM agent under stress tests, as compared to the stock Linux.	30
8.1	The workflow of PowerAdvisor	34

Tables

3.1	SoC device drivers and the delay (in months) between their initial release and the implementation of device runtime PM support. Asterisks indicate the absence of PM support even at the time of writing. To confirm the validity of these findings, we checked both the source code released by device vendors and SoC vendors. For example, we checked Amazon’s source code for Kindle Paperwhite which uses i.MX SoC, and Freescale’s source code for i.MX SoC. . . .	12
6.1	The cost of adding a busy/idle register	23
8.1	Traces used in evaluating PowerAdvisor	38

Chapter 1

Introduction

A system-on-chip (SoC) is often the heart of a wide spectrum of embedded systems [1–4]. It is complex, incorporating not only multiple general purpose cores (CPU) but also dozens of IP (intellectual property) modules, most notably IO controllers such as I2C and display controller, and accelerators such as multimedia codec, DSP, and GPU. These IP modules not only bring specialized functions to a SoC efficiently, but also account for the majority of the SoC chip area [5]. In this thesis, we refer to non-CPU IP modules on a SoC as *devices*, as in “device drivers”. On-chip clocks and power supplies are distributed to the devices in a hierarchical way. Often, a commodity Operating System (OS) runs on the CPU and manages/abstracts the devices with corresponding device drivers.

In a system that employs a SoC, power management (PM) is a central design issue that requires software and hardware collaboration. The end 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 thesis focuses on *device runtime PM*: when the CPU is on, the OS should enable/disable individual devices properly and timely. Such runtime PM decisions are bubbled up to the hardware clock and power hierarchy, finally leading to clock (un)gating, power supply switching, or voltage change. Note that device runtime PM is not concerned with the CPU, which is an orthogonal research problem [6] addressed

Figure 1.1 : 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.

by other OS subsystems, e.g., `cpufreq` in Linux.

Aside from runtime PM, system suspension is another major approach to PM, which is exemplified by Android. Android opportunistically suspends the entire SoC, including the CPU, when there is no user interaction. Many prior works focus on fixing buggy user applications that prevent Android from system suspension [7–9], or optimizing the scheduling of user applications to extend the suspension time [10]. Nevertheless, Android also applies runtime PM, which saves power when the CPU is on and is orthogonal to system suspension.

We choose to focus on device runtime PM because it enables finer-grained, per-device PM that is not possible with system suspension. In fact, if implemented correctly, runtime PM can replace system suspension [11]. Moreover, device runtime PM has been recognized as vital by the Linux community [12] due to the emergence

of always-on workloads.

Despite the importance of device runtime PM, it is poorly implemented in commodity OSes such as Linux. In this thesis, we highlight the shortcomings of Linux device runtime PM: as SoCs appear in the market, their device PM support is often long overdue, and is likely to be minimal when finally implemented.

The root cause of such difficulties is that today’s commodity OSes hold drivers responsible for PM. To understand this problem, we identify key information 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 for tracking the number of concurrent software that are using the device, as shown in Figure 1.1(a). This reliance effectively makes the 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 1.1(b): introducing a central OS component, or central PM agent, to relieve device drivers from their runtime PM responsibilities. To build 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 the information: using software to monitor device register access or adding one extra hardware register 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 state 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% less. The central PM agent incurs very low overhead: in our FPGA-based implementation, the added hardware register introduces at most 34(+1.6 %) gates as estimated by the synthesis tool; the device performance loss is negligible.

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 offers suggestions to developers on where to add runtime PM calls in source code. We show that PowerAdvisor is effective: it discovers previously unknown locations for adding PM in a complicated, 22K-SLoC display controller driver.

In summary, this thesis makes three major contributions:

- 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 extra hardware support.
- We present a complete design and implementation of the central PM agent, whose performance is comparable with manual runtime PM code in Linux device drivers.
- We present PowerAdvisor, a tool that effectively simplifies the development of driver PM under the current Linux framework.

Chapter 2

Background

A modern SoC consists of tens of hardware devices from computational devices like CPU, GPU, to I/O controllers like I2C bus controller and display controller. These devices are connected together via on-chip interconnects, and expose functions to the CPU via their registers.

2.1 Hardware PM Infrastructure

We next sketch the hardware support for power management available on mobile 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, 13–17].

A Hierarchy of Domains. Hardware devices on a SoC may share clock and power supplies. This effectively organizes all devices into a hierarchy of domains. 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 the domains discussed below.

- A *clock domain* is a group of devices sharing the same clock sources. Domain-level clock gating stops the clock sources, such as a digital phase-locked loop,

Figure 2.1 : Hardware PM infrastructure in SoC (as exemplified by OMAP4) and the Linux PM frameworks

that are used by the domain. It sets the clock domain as INACTIVE and results 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 switches. 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).

The three types of domains are the basic units of clocks management, power management, and voltage management, respectively. For them, software only chooses the target low-power states (e.g., RETENTION, OFF); hardware decides when to perform the actual state transitions, as discussed below.

Global Power Manager. Global power manager is a special on-chip hardware device that coordinates various devices and domains in performing power state transitions. The manager sets a domain to a low-power state if all encompassed devices of the domain have been configured by software as `DISABLED`; the manager brings the domain back to 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. First, it decides the mode of a device: `ENABLED` or `DISABLED`. Second, it decides the target low-power state of a domain so that QoS requirements are met despite power management. We next discuss how Linux fulfills these two roles and highlight the responsibility of device drivers.

Note that the Linux community often abbreviate device runtime PM as *runtime PM*. We will respect the convention in describing the related Linux API.

Linux Runtime PM Framework The Linux runtime PM is based on the fact that a device can only be safely set to `DISABLED` when no software is using the device. In order to track the number of concurrent use of the device, Linux provides a generic runtime PM framework. The framework keeps a per-device reference counter, whose value, by design, should equal the number of concurrent software using the device. When the reference counter drops to zero, the framework sets the device to `DISABLED`.

Built under the runtime PM 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.

Linux PM QoS Framework Linux provides the PM QoS framework to allow users to express QoS requirements (e.g., wakeup latency) for the OS to meet. 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.

Chapter 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 chapter, we provide three reasons and elaborate upon each with a real-world case.

PM as Afterthought 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 with preliminary, coarse-grained power management for a long time. Proper PM support, if any, appears much later than the driver’s initial release.

Real-world Case: Delayed PM for Samsung Exynos SPI

The Exynos SPI driver had preliminary PM code for 25 months after its first commit [18]. The preliminary PM, as shown in Listing 3.1, is very coarse-grained and saves no energy at run time. It simply enables the SPI controller in `probe()`, which is only 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 3.2 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 3.1.


```

1  int s3c64xx_spi_probe(platform_device *pdev)
2  {
3      /* allocate resource for controller...*/
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 resource...*/
13 }

```

Listing 3.1: Preliminary PM that has existed in the Exynos SPI driver for more than two years.

Complex Device Drivers Modern SoC hardware is 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: The complex OMAP4 display driver

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

```

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 3.2: Hand-tuned PM in the Exynos SPI driver after patching [18].

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.

Hierarchical PM Hardware Infrastructure The hierarchical PM infrastructure magnifies the power waste resulting from improper PM of individual device 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.

Driver	Delay	Driver	Delay
OMAP UART	12 ^[19]	Exynos Keypad	18 ^[20]
OMAP Wdt	22 ^[21]	Exynos USB	14 ^[22]
OMAP USB	45 ^[23]	OMAP GPIO	32 ^[24]
i.MX SD Ctrl	37 ^[25]	i.MX I2C	9 ^[26]
Tegra SD Ctrl	44 *	i.MX SPI	31 ^[27]

Table 3.1 : SoC device drivers and the delay (in months) between their initial release and the implementation of device runtime PM support. Asterisks indicate the absence of PM support even at the time of writing. To confirm the validity of these findings, we checked both the source code released by device vendors and SoC vendors. For example, we checked Amazon’s source code for Kindle Paperwhite which uses i.MX SoC, and Freescale’s source code for i.MX SoC.

Real-world Case: UART lacking PM keeps an entire power domain on

During one entire year after its first release, the OMAP4 UART driver did no power management, keeping the whole L4_PER power domain on as long as the CPU is on [19]. Although the ENABLED UART 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.

Chapter 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 [28–30]. 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 different power states, including power consumption in all power states, the latency and energy consumption for transition between two 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, the device can 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 resumes the device context accordingly. However, this incurs unnecessary power state transition overhead. Therefore, a good power management policy keeps the device enabled until it finishes *all* pending tasks.

For a given device, 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 [31] 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.

Chapter 5

Pending Task Inference

As discussed above, knowing if the target device has pending tasks is key to power management. In this chapter, we propose two alternative ways to acquire this knowledge outside device drivers, i.e., without device drivers' support: *(i)* a software approach that infers the knowledge without extra hardware support and *(ii)* a small hardware modification to device hardware for exposing whether the device is busy or idle through an additional register bit.

5.1 Software-based Inference

The software approach infers whether 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 $T_{threshold}$, the device is likely to have no pending task.

Taking I2C transmission as an example: the device registers are frequently accessed until the task is completely done. 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 if a device has pending tasks more timely 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 SoC, the largest interval appears when a DMA transaction is in progress. Given 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 The software inference applies to most, but not all types of devices. The deciding factor is that for a given device, whether the length of processing periods without register access is bounded so that the insight 1 holds. For IO 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 is practically

unbounded.

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.

5.2 Hardware-assisted Inference

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

By polling these registers 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 SoC 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 represents the device being idle. Therefore, the busy/idle register is essentially a mapping from the

subset of states to a single bit, which is an easy modification.

Chapter 6

Central PM Agent

With the pending task information inferred, we can realize device runtime PM of all 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 chapter, we first describe the overall structure of the central PM agent; then we describe how the software and hardware-assisted pending task inference fit in the design.

6.1 Overall Structure

As shown in Figure 6.1, 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 hardware busy/idle register. Based on the monitor output, the controller calls the PM callbacks provided by the driver to set the device to ENABLED/DISABLED.

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

Figure 6.1 : The structure of central PM agent, a concrete design following the conceptual model shown in Figure 1.1(b). The monitor employs two alternative approaches for pending task inference, which are prototyped on two respective SoCs.

$T_{threshold}$ 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, being either **dirty** or **clean**:

- **dirty** means that CPU has accessed the device's registers since the last check;
- **clean** means otherwise.

Figure 6.2 illustrates how central PM agent works for a given device with 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 jobs. First, it checks the software status: a **clean** status (⊙) indicates that the device's registers have not been accessed

Figure 6.2 : An example timeline of central PM agent execution, which infers the existence of pending tasks using the software-based approach.

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, which will then safely disable the device. If the software 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 our Insight 2 in §5.1.

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

Implementation on OMAP4 SoC We implement the central PM agent based on the above monitor design as a kernel module on OMAP4 SoC. We choose OMAP4 because it has abundant public information and has full fledged power management hardware infrastructure.

6.3 Hardware-assisted Pending Task Monitor

We add the one-bit busy/idle registers to existing devices, so that *(i)* the power management can be more aggressive and *(ii)* the design of the monitor is simplified and causes less overhead. We next describe how we design the busy/idle register and change the design of the monitor accordingly.

Mapping device state machine to busy/idle register As discussed in 5.2, manufacturers describe a device hardware as a state machine, with a subset of the states representing the device being idle. We design the semantics of the busy/idle register this way:

The busy/idle register is set to busy when the state machine leaves the idle states; it is reset to idle on the CPU read if the state machine is in the idle states 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 at least once been busy since CPU's last read of the busy/idle register. On the other hand, if the register shows idle, it indicates the device has always been idle since CPU's last read of the register.

Because the logic of a busy/idle register re-uses existing state machine in the hardware, the required hardware resources and development efforts to implement it is small. As the conceptual design in Figure 6.3(a) shows, aside from a busy/idle

(a) busy/idle register implemented in RTL

```

reg busyIdle;
always @(posedge clk)

/* state machine in busy */
if(!state !=0)
    busyIdle = 1'b1';

/* busy/idle reg is just read,
and state machine is in idle*/
else if (read)
    busyIdle=1'b0;

```

(b) busy/idle register described in HDL

Figure 6.3 : Example implementation of a busy/idle register, in (a) several tens of gates and (b) in several lines of HDL code

Module	Development Efforts		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%)

Table 6.1 : The cost of adding a busy/idle register. Development time is in man-hours. *The extra gates required by Xilinx I2C is unknown, because it uses Xilinx specific FPGA primitives, which cannot be synthesized to ASIC.

register, the design merely requires several gates to extract the needed information from the state machine. Figure 6.3(b) shows that several lines of HDL code suffice to describe the logic of the busy/idle register.

Using the busy/idle register Instead of triggering the 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 consumes energy. In practice, we expect that a polling interval of tens of milliseconds is aggressive enough.

Implementation on Zynq-7000 SoC We choose Zynq for using its on-chip FPGA to prototype the busy/idle register. Moreover, the FPGA has dedicate clocks which can be gated in software without affecting the rest of the system. We add a busy/idle register to an Xilinx I2C controller [32], and instantiate the modified I2C on the FPGA. We run the central PM agent on Zynq’s CPU. As shown in Table 6.1, the development efforts in modifying the I2C controller is also small. The extra FPGA resources, i.e., lookup tables (LUT) and FPGA registers, required by the implementation are moderate. The estimated number of extra gates required for ASIC implementation is 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 [33] and an Opencores SPI controller [34] written in Verilog. As shown in Table 6.1, the involved development efforts and FPGA resource usage are similarly small.

Chapter 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 3.2 kernel is the latest kernel that supports Android on OMAP SoC. Its driver code base receives contributions regularly from TI, Google, etc.

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 real-world trace as benchmark to test the central PM agent. In order to do so, we employ a trace of user activity lasting 600 seconds containing 48 user input events, in which the user actively reads emails using the Android email application and browses

the web using the default Android browser. These user 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 device drivers: the difference in the disabled time is within 3.3% (with $T_{threshold}=100$ ms). Moreover, the central PM agent provides PM to drivers lacking PM. We next describe the results for the four devices in detail.

MMC. On many embedded systems, CPU accesses the root filesystem through a MMC controller. The Linux MMC 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 driver. As shown in Figure 7.1(a), when we choose $T_{threshold}$ to be a typical value of 100 ms, the central PM agent keeps the MMC controller in DISABLED for 77.2% of the time, which is only 3.3% 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% more, as the hand-tuned PM chooses to disable the controller after a 100 ms idle timeout.

I2C and SDIO. The I2C and SDIO on Pandaboard are used to serve one common use – to bridge the CPU with the Wi-Fi interface. The I2C driver comes with hand-tuned PM; the SDIO driver does not, leaving the SDIO always on as long as the

Figure 7.1 : Device DISABLED time with different $T_{threshold}$. The $T_{threshold}=100$ ms (with underline) is the default value. 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.

Wi-Fi interface is enabled, even when no data is being transmitted. The benchmark exercises the two drivers with rich network activities generated from web browsing and email fetching. The results are shown in Figure 7.1(b) and (c). When we choose $T_{threshold}$ to be a typical value of 100 ms, the central PM agent keeps I2C DISABLED

for 81.2% of the time, which is only 2.1% less than the hand-tuned PM; the central PM agent keeps SDIO 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, which is HDMI in our setup. Its driver comes with minimal PM, which keeps 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 7.1(d). When we choose $T_{threshold}$ to be 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. Because with a larger $T_{threshold}$, a device not only lingers in 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, leading 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 PM to drivers that lack PM (SDIO 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 [35]. By disabling DISPC, the central PM agent saves 10 mW. Despite the central PM agent only saving a small amount of power by disabling SDIO, it would save 17 mW if SDIO were to block the encompassing power domain from entering RETENTION altogether.

The power reduction is relatively small compared to the typical power consumption of smartphones under active use. For example, Samsung Galaxy S III consumes 1300 mW during web browsing via Wi-Fi [36]. Nevertheless, the resulting energy savings are significant, in terms of the extended standby time estimated from smartphone daily usage reported in LiveLab [37]. For SDIO, the stock driver keeps SDIO always ENABLED as long as the user has not manually switched off Wi-Fi; compared to it, the central PM agent saves up to 71.4 mWh daily, which extends the standby time by 2.4 hours. For DISPC, the stock driver keeps DISPC always ENABLED as long as the screen is on; compared to it, the central PM agent saves around 18.5 mWh daily, which extends the standby time by 0.6 hour.

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 CPU cycles, which is at most 8 μ s when 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 SD card (backed by MMC) and Wi-Fi interface (backed by SDIO and I2C) and measure the performance loss caused by the central PM agent. All results are averaged over 10 iterations.

Figure 7.2 : Performance of devices managed by central PM agent under stress tests, as compared to the stock Linux.

To stress 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. As shown in Figure 7.2(a), the throughput loss due to central PM agent is negligible, even smaller than the standard deviations.

To stress Wi-Fi interface, we use *iperf* [38] 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 7.2(b), the central PM agent introduces negligible throughput loss in both sending and receiving. The loss is even smaller than the standard deviations.

7.2 Hardware-assisted Central PM

We prototype the hardware-assisted central PM agent on 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 [39] 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 added busy/idle register is working without breaking any device functionality. In addition, our efforts in bringing up the central PM agent on Zynq are small.

Chapter 8

PowerAdvisor for PM Code Suggestion

The central PM agent presented in § 6.2 argues 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 runtime PM calls added to existing driver source code. Consisting of an instrumenter and an offline analyzer, the tool analyzes historic execution trace and identifies runtime PM calls that work for the historic execution.

Figure 8.1 shows the PowerAdvisor workflow. A developer first instruments the driver with the tool. In test runs, the developer runs various user workloads to exercise the instrumented driver. The instrumented driver will generate trace, which is fed into the analyzer after the test run is finished. After the test runs, the analyzer examines the trace, and finally 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 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 device to decide if a task is being processed there. Note that the tool cannot assume that device has to be always ENABLED outside of no-pending-task periods, a too strong constraint that suppresses useful suggestions.

Will the added runtime PM calls be effective in future executions? PowerAdvisor makes suggestions purely based on historic observations. The above two guarantees only apply to the historic execution trace; developers need to reason to ensure that the PM code works in the future.

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

Figure 8.1 : The workflow of PowerAdvisor

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 [40], 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:

- `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 one or 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.

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

Collecting Trace

As shown in Figure 8.1, in a test run, whenever a tracepoint is reached by the control flow, it appends its identifier into an global trace buffer. In addition, any no-pending-task period is logged to the same trace buffer, which is detected if 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.

Analyzing Trace

Overall, the PowerAdvisor analyzer looks for legal PM calls, when applied to the collected trace, can satisfy the guarantees in §8.1. The analyzer translates the problem into a Satisfiability Modulo Theories (SMT) problem, with the linear integer arithmetic (LIA) as the background theory. It establishes SMT constraints based on the trace, and invokes the Z3 SMT solver [41] to identify 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_PUT}) and after (S_{CS_PUT}) the marked call sites:

$$V = S_{BB_GET} \cup S_{BB_PUT} \cup S_{CS_GET} \cup S_{CS_PUT} \quad (8.1)$$

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

$$\begin{aligned} \forall v \in S_{BB_GET} \cup S_{CS_GET}, v \in \{0, 1\} \\ \forall v \in S_{BB_PUT} \cup S_{CS_PUT}, v \in \{0, -1\} \end{aligned} \quad (8.2)$$

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 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.

$$\sum_{q_i \in Q} q_i = 0 \quad (8.3)$$

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

$$\forall j \in \{1 \dots N\}, \text{ if } q_j \in S_{BB_GET}, \text{ then } \sum_{i=1}^j q_i > 0 \quad (8.4)$$

Note that the analyzer generates the constraints (8.3) and (8.4) for all subsequences in the trace.

At last, the analyzer has to minimize the number of introduced PM calls C , which is

$$\sum_{v_i \in V} |v_i| = C \quad (8.5)$$

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.

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

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

Table 8.1 : Traces used in evaluating PowerAdvisor

8.3 Evaluation of PowerAdvisor

We experimentally show that the PowerAdvisor can offer useful and correct suggestions. In order to do so, we study four drivers: MMC, I2C, SDIO, and DISPC. For test runs, we ask a user to use the Android system to browse 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 traces and resulting SMT problems are shown in Table 8.1. Note that we choose to extend the test run for DISPC as DISPC is significantly more complex than the others.

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

The drivers of SDIO and DISPC come with no PM code. For the SDIO 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()` and `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 animations on the screen. The driver runs for more than 48 hours without breaking the device functionality, and correctly disables the device when it has no pending task.

Chapter 9

Related Work

Power management has always been a multi-faceted problem. It is critical not only to system efficiency but also to performance as we enter the dark silicon era [42]. We roughly categorize prior PM works as that applies to when system is suspended and when system is running. We also discuss model checking and code synthesis tools that are related to PowerAdvisor.

System suspension management. In system suspension, most SoC hardware (including CPU) is off, leaving on only the necessary circuits for wakeup. Although effective and popular, it 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 [8] and sleep conflict [9], and stirs up long debates in the Linux community [11]. Many research works have been proposed as fixes: finding the battery draining culprit application by identifying its new energy-hungry phase [7], detecting power bugs by monitoring IO bus traffic [9], or using static analysis approach to detect power bugs in the source code [8].

In the recent Android 4.4 update, Android extends the period of time the system stays in suspension by batching the alarms set by applications [10]. Since this update,

Android by default treats the trigger time of all alarms as inexact to allow the batching. This alarm batching technique is not limited to system suspension management. Other systems such as Linux and OS X also apply similar techniques to extend CPU idle time [43, 44].

Runtime power management. The goal of runtime PM is to reduce power consumption when system is functional and the CPU is on (including idle).

Runtime PM for Computational Unit: A rich set of works focuses on matching the computational unit (e.g., CPU or GPU) performance with the performance expectation of given workloads. Vertigo [6] uses a hierarchy of performance-setting algorithms to choose CPU operating point for interactive applications. The Process Cruise Control framework [45] and the Koala framework [46] adjust CPU performance for given workloads according to the workloads characteristics captured with performance counters. Pathania *et al.* adjusts CPU and GPU performance point together according to 3D game workloads history to save energy [47]. 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 CPU [48].

Runtime PM for Device: Device runtime PM aims at disabling individual SoC modules that are not in use when the CPU is on. Compared to suspending the whole system, device runtime PM is finer-grained and can leverage more power saving opportunities. If done properly, device runtime PM enables the SoC to consume similar power as system-wide suspension does [11] and to still remain responsive to external user inputs. 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 [12].

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 [31]. ICEM [49] integrates power management 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. [50] proposes a new IO API for applications to disclose hints for better device power management. 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.

Formal methods and synthesis tools. PowerAdvisor (§8) is essentially related to software tools that use formal methods for bug finding or code synthesis. However, these tools, when applied to PM, cannot make developer’s task much easier.

Model checking tools like CBMC [51] and symbolic execution tools like KLEE [40] can be used to verify if a module is disabled when control flow reaches given source code locations. However, it is up to the developers to manually identify those locations where the module must have no pending task. Automatic device driver synthesis tools like Termite [52] are expected to be capable of synthesizing PM code given formal device specifications, either derived from device documentation or device RTL description. However, it is unclear whether deriving such formal specification is any easier, if not harder, than writing driver PM code.

Chapter 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 powering off, possibly by implementing a RETENTION state, instead of relying on the software to preserving the hardware context; *(ii)* each device must be capable of interpreting various PM QoS requests expressed by users, possibly by adding a microcontroller. Therefore, we advocate keeping software responsible for the context saving and PM QoS so that they can be implemented in a cheaper and more flexible way.

Is the Z3 SMT solver the most efficient tool for the PowerAdvisor? Because the SMT problem defined in § 8.2 is the conjunction of constraints 8.1 through 8.5, the SMT problem can be rewritten as an Integer Linear Programming (ILP) problem, with 8.1 through 8.4 as constraints, and 8.5 as the objective function. An ILP problem can be solved by linear programming solver like CPLEX [53] and MOSEK [54]. Compared to these linear programming solvers, Z3 is less optimized in solving this specific problem because, before calling its linear arithmetic solver, Z3 has to go through an extra step of calling the DPLL-based SAT solver to solve the boolean satisfiability problem abstracted from the SMT problem [41]. Nonetheless, the overhead caused by this extra step is small, because the SAT solver will quickly

determine that all the constraints in conjunction have to be satisfied. A more thorough discussion on the relationship between SMT and linear programming can be found in literature on SMT [55, 56].

Is the central PM agent applicable to high-performance SoCs? Because high-performance SoCs, such as Freescale QorIQ Qonverge [57] and TI Keystone [4], also group on-chip devices into clock domains and power domains, it is possible to do fine-grained runtime power management for them. These high-performance SoCs may contain power-hungry IO devices that are not present on mobile SoCs, such as a PCIe host or a SATA host adapter. In principle, the central PM agent presented in § 6.2 is applicable to these IO devices. Furthermore, high-performance SoCs may contain extra computational devices such as baseband accelerators and network processors. It is unclear how these devices are used; if they are similar to the OMAP4 face detection accelerator discussed in § 5.1, i.e. with bounded register access interval, then the central PM agent is applicable to them.

Conclusion Device runtime PM requires the OS to disable devices that has 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.

Bibliography

- [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 *Solid-State Circuits Conference (ISSCC)*, pp. 226–228, 2012.
- [6] K. Flautner and T. Mudge, “Vertigo: automatic performance-setting for Linux,” in *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 105–116, 2002.
- [7] 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. of the 10th USENIX Conference on Networked*

Systems Design and Implementation (NSDI), pp. 57–70, 2013.

- [8] 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. of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pp. 267–280, 2012.
- [9] A. Jindal, A. Pathak, Y. C. Hu, and S. Midkiff, “Hypnos: understanding and treating sleep conflicts in smartphones,” in *Proc. of the 8th ACM European Conference on Computer System (EuroSys)*, pp. 253–266, 2013.
- [10] Google Inc., “Manual of AlarmManager class.” <http://developer.android.com/reference/android/app/AlarmManager.html>, 2014.
- [11] R. J. Wysocki, “Technical background of the Android suspend blockers controversy.” lwn.net/images/pdf/suspend_blockers.pdf, 2010.
- [12] 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.
- [13] NVIDIA, “NVIDIA Tegra 4 4-plus-1 quad-core processors technical reference manual,” 2013.
- [14] Freescale, “i.MX 6Dual/6Quad applications processor reference manual,” 2013.
- [15] Intel, “Intel Atom processor Z36xxx and Z37xxx series datasheet,” 2013.
- [16] 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*, vol. 17, no. 3, pp. 427–438, 2009.

- [17] T. Hattori, T. Irita, 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 *Solid-State Circuits Conference (ISSCC)*, pp. 2210–2219, 2006.
- [18] M. Brown, “SPI/s3c64xx: implement runtime PM support.” <http://www.spinics.net/lists/linux-samsung-soc/msg08912.html>, 2012.
- [19] G. Raja, “OMAP2+: UART: add runtime pm support for OMAP-serial driver.” <http://www.spinics.net/lists/linux-omap/msg58443.html>, 2011.
- [20] M. Brown, “Input: samsung-keypad: implement runtime power management support.” <http://www.spinics.net/lists/linux-input/msg18796.html>, 2011.
- [21] P. Walmsley, “Watchdog: OMAP_wdt: add fine grain runtime PM.” <http://permalink.gmane.org/gmane.linux.ports.arm.omap/54608>, 2011.
- [22] V. Gautam, “DWC3/xHCI: enable runtime power management.” <https://lkml.org/lkml/2013/1/28/229>, 2013.
- [23] R. Quadros, “USB: implement runtime idling and remote wakeup for OMAP EHCI controller.” <https://lkml.org/lkml/2013/7/10/355>, 2013.
- [24] F. Balbi, “GPIO: OMAP: be more aggressive with pm_runtime.” <http://www.spinics.net/lists/linux-omap/msg64196.html>, 2012.

- [25] A. Dong, “MMC: sdhci-esdhc-imx: add runtime PM support.” <http://lists.infradead.org/pipermail/linux-arm-kernel/2013-October/208191.html>, 2013.
- [26] 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.
- [27] 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.
- [28] S. Irani, S. Shukla, and R. Gupta, “Online strategies for dynamic power management in systems with multiple power-saving states,” *ACM Transaction Embedded Computing Systems (TECS)*, vol. 2, no. 3, pp. 325–346, 2003.
- [29] T. Simunic, L. Beniani, and G. De Micheli, “Event-driven power management of portable systems,” in *Proc. of International Symposium on System Synthesis*, pp. 18–23, 1999.
- [30] Q. Qiu and M. Pedram, “Dynamic power management based on continuous-time Markov decision processes,” in *Proc. of the 36th Design Automation Conference (DAC)*, 1999.
- [31] 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. of Linux Symposium*, 2012.

- [32] 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.
- [33] R. Herveille, “Opencores project: I2C.” <http://opencores.org/project,i2c>.
- [34] R. Herveille, “Opencores project: simple SPI.” http://opencores.org/project,simple_spi.
- [35] eLinux.org, “PandaBoard Power Measurements.” http://elinux.org/PandaBoard_Power_Measurements.
- [36] A. Carroll and G. Heiser, “The systems hacker’s guide to the galaxy: energy usage in a modern smartphone,” in *Proc. of the 4th Asia-Pacific Workshop on Systems (APSYS)*, p. 5, ACM, 2013.
- [37] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, “Livellab: Measuring wireless networks and smartphone users in the field,” *SIGMETRICS Performance Evaluation Review*, vol. 38, pp. 15–20, Jan. 2011.
- [38] 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/>.
- [39] Freescale, “Xtrinsic MMA8452Q 3-axis, 12-bit/8-bit digital accelerometer.” http://www.freescale.com/files/sensors/doc/data_sheet/MMA8452Q.pdf, 2014.
- [40] C. Cadar, D. Dunbar, and D. Engler, “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proc. of the 8th*

- USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pp. 209–224, 2008.
- [41] 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.
- [42] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *Proc. of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pp. 365–376, 2011.
- [43] Apple Inc., “Power efficiency in OS X.” https://www.apple.com/media/us/osx/2013/docs/OSX_Power_Efficiency_Technology_Overview.pdf, 2013.
- [44] Linux, “man-page of time(7).” <http://man7.org/linux/man-pages/man7/time.7.html>, 2012.
- [45] A. Weissel and F. Bellosa, “Process cruise control: Event-driven clock scaling for dynamic power management,” in *Proc. of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pp. 238–246, 2002.
- [46] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser, “Koala: a platform for OS-level power management,” in *Proc. of the 4th ACM European Conference on Computer Systems (EuroSys)*, pp. 289–302, 2009.
- [47] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra, “Integrated CPU-GPU power management for 3D mobile games,” in *Proc. of the 51st Annual Design Automation Conference (DAC)*, pp. 40:1–40:6, 2014.

- [48] 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 Conference (ATC)*, pp. 367–372, 2012.
- [49] 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. of Symposium on Operating Systems Principles (SOSP)*, pp. 251–264, 2007.
- [50] M. Anand, E. B. Nightingale, and J. Flinn, “Ghosts in the machine: interfaces for better power management,” in *Proc. of the 2nd International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pp. 23–35, 2004.
- [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. of the 22nd symposium on Operating systems principles (SOSP)*, pp. 73–86, ACM, 2009.
- [53] IBM, “CPLEX Optimizer.” <http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/>.
- [54] MOSEK ApS, “MOSEK optimization software.” <http://www.mosek.com/>.
- [55] Y. Yu and S. Malik, “Lemma learning in SMT on linear constraints,” in *Theory and Applications of Satisfiability Testing (SAT)*, vol. 4121, pp. 142–155, 2006.
- [56] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, “Satisfiability Modulo Theories,” *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.

- [57] Freescale, “BSC9131 QorIQ Qonverge Multicore Baseband Processor.” http://cache.freescale.com/files/32bit/doc/data_sheet/BSC9131.pdf, 2014.