Thermal and QoS-Aware Embedded Systems

by

Youngmoon Lee

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2019

Doctoral Committee:

Professor Kang G. Shin, Chair
Assistant Professor Mosharaf Chowdhury
Professor Wei Lu
Professor Thomas F. Wenisch

Youngmoon Lee

ymoonlee@umich.edu

ORCID iD: 0000-0002-6393-2994

To my wife Jinwoo and my parents Taeho and Hyesook

for their unwavering love and support

ACKNOWLEDGEMENTS

When I look back on the 5 years of my Ph.D. program, I now realize how much I have been given. I was given an endless time to sit down and think that made me an independent thinker. Furthermore, I was given another chance upon failures, which made me never give up on challenges. My most sincere gratitude goes to my advisor Kang G. Shin, who has been extremely patient in fostering my independent thinking. I would also like to thank my collaborator Mosharaf Chowdhury who has been my hands-on mentor. I will never forget his patience and never giving up spirit.

I am indebted to all the members of Real-Time Computing Lab (RTCL) where I learned a lot from each of these independent thinkers. I would especially like to acknowledge my colleagues Sunmin, Liang, and Hoonsung. We spent an enormous amount of time shaping ideas and refining them into research problems and solutions. I would also like to thank Karen, Xiaoen, Krishina, Huan, Kassem, Kyongtak, Yuchih, Seunghyun, Arun, Dongyao, Taeju, Juncheng, Chunyu, Mert, Duc, Tim, Hsunwei, Youssef, Suining, Hamed, Sunhyun, Daehan, Jiaqing, and Haichuan.

I would like to express my gratitude to my family, especially my wife, Jinwoo. She has been always with me when I was struggling the most. Thank you for trusting me for the 7 years we have spent together. In addition, I appreciate my parents - Taeho and Hyesook - for their unwavering love and support.

My acknowledgments also go to all the people who helped me during my graduate studies. Finally, I wish to acknowledge the support of the National Science Foundation and the South Korean government with my graduate fellowship.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

While embedded systems such as smartphones and smart cars become essential parts of our lives, they face urgent thermal challenges. Extreme thermal conditions (i.e., both high and low temperatures) degrade system reliability, even risking safety; devices in the cold environments unexpectedly go offline, whereas extremely high device temperatures can cause device failures or battery explosions. These thermal limits become close to the norm because of ever-increasing chip power densities and application complexities. Embedded systems in the wild, however, lack adaptive and effective solutions to overcome such thermal challenges. An adaptive thermal management solution must cope with various runtime thermal scenarios under a changing ambient temperature. An effective solution requires the understanding of the dynamic thermal behaviors of underlying hardware and application workloads to ensure thermal and application quality-of-service (QoS) requirements. This thesis proposes a suite of adaptive and effective thermal management solutions to address different aspects of real-world thermal challenges faced by modern embedded systems.

First, we present `BPM`, a battery-aware power management framework for mobile devices to address the unexpected device shutoffs in cold environments. We develop `BPM` as a background service that characterizes and controls real-time battery behaviors to maintain operable conditions even in cold environments. We then propose `eTEC`, building on the thermoelectric cooling solution, which adaptively controls cooling and computational power to avoid mobile devices overheating. For the real-time embedded systems such as cars, we present `RT-TRM`, a thermal-aware resource management framework that monitors changing ambient temperatures and allocates system resources to individual tasks. Next, we target in-vehicle vision systems running on CPUs–GPU system-on-chips and develop CPU–GPU co-scheduling to tackle thermal imbalance across CPUs caused by GPU heat. We evaluate all of these solutions using representative mobile/automotive platforms and workloads, demonstrating their effectiveness in meeting thermal and QoS requirements.

CHAPTER I

Introduction

Smartphones are becomeing essential to daily life; people use these devices not only to connect with others but also to navigate while driving, and even make mobile payments and conduct banking. Beyond the smartphone era, smart cars with advanced driver assistant systems (or even automated driving) are growing rapidly; both the European Union and United States have mandated that by 2020, all vehicles must be equipped with autonomous emergency-braking systems and forward-collision warning systems [34]. With the number of smartphones exceeding the world's population [49] and smart cars expected to hit 1.2 billion by 2025 [91], these embedded systems will further improve the quality and safety of life.

These and other smart applications/systems are enabled by (i) the increasing computational power of embedded systems and (ii) advances in artificial intelligence – a trend that will not slow with the rise of the edge computing paradigm, where interconnected smart/edge devices will perform major computations for end users [108]. Future embedded systems with advanced functionalities and increasing computational capabilities will soon be integrated into every aspect of life, rendering them indispensable.

Such embedded systems, however, face urgent thermal challenges in both extreme thermal conditions (i.e., high and low temperatures) where their reliability is compromised, such as in the following cases:

- Device overheating shortens the lifetime of a device and severely degrades its reliability, even risking safety (e.g., vehicle breakdowns or smartphone explosion).
- Extremely low-temperatures cause unpredictable shutoffs of battery-powered devices; for example, devices might shut off even when their batteries are shown to have 20% remaining capacity.

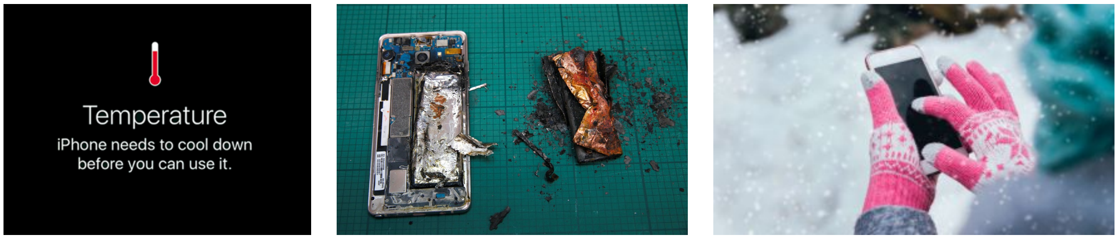
Such systems must cope with changing environmental temperature to overcome these thermal challenges. Dynamic thermal behaviors of application workloads and underlying platforms must also be accounted to meet the thermal requirements. This thesis identifies the new thermal challenges in modern embedded systems (§1.1), highlights dynamic thermal characteristics therein (§1.2) along with state-of-the-art (§1.3), and proposes a set of thermal management solutions to address those deficiencies (§1.4).

1.1 Thermal Challenges

As embedded systems evolve, thermal limits are close to the norm where system reliability is compromised, as is evident in Fig. 1.1.

1.1.1 High Temperature

Dangerously high temperatures severely degrade a system's reliability, and even risk safety. Device overheating often makes the system unavailable (Fig. 1.1a) and can cause battery explosion (Fig. 1.1b), such as in the example of Samsung Note 7 explosions [93]. These thermal emergencies not only cause monetary loss but can also lead to catastrophic consequences for driving, medical and wearable applications. When a given temperature threshold is reached, these devices are often



Temperature

iPhone needs to cool down before you can use it.

(a) Device Unavailable (b) Battery Explosion (c) Unexpected Device Shutoff

Figure 1.1: Thermal challenges in embedded systems threatening system reliability and user's safety.

cooled by stopping/slowing their operations, and these applications therein experience significant lagging and quality-of-service (QoS) degradation.

1.1.2 Low Temperature

Cold environments also limit the availability of battery-powered embedded systems (Fig. 1.1c). This is because the power-supply capability of batteries severely degrades in low temperatures. Devices unpredictably shut off even when the battery is shown to have plenty of capacity remaining, and such unmanageable device shutoffs may lead to disastrous situations for mission-critical applications. In 2017, Apple attempted to avoid such premature/unexpected device shutoffs. They limited the maximum allowed discharge current (through regulating the maximum speed of the processors) on iPhones in cold environments. This solution, however, caused noticeable degradation in the QoS perceived by device users, leading to multiple lawsuits against Apple [92].

Furthermore, these thermal challenges will become worse with increasing application complexities [47] and chip-power densities but a fixed thermal limit [110]. Simultaneously, as people's lives increasingly rely on these devices, their thermal reliability will become even more significant.

1.2 Dynamic Thermal Behaviors

Thermal behaviors of embedded systems are often dynamic, making their thermal management challenging. Three major dynamic factors dictate device temperature: ambient temperature, hardware thermal characteristics and dynamic application behaviors.

1.2.1 Changing Ambient Temperature

Embedded systems, unlike desktops or data-centers, experience a wide range of environmental variations especially ambient temperature during their operation/life. Ambient temperature changes dynamically, and their seasonal and locational variations are very wide. Because a device's temperature depends on the ambient temperature, changing ambient temperature under a fixed thermal limit indicates a changing thermal budget.

1.2.2 Hardware Thermal Variabilities

Other issues pertaining to thermal management are the unique thermal characteristics of underlying hardware components. To manage the thermal behaviors of computing platforms, cooling devices must be captured to manage system temperatures effectively. To enable reliable system operation, the temperature-dependent characteristics of batteries must also be captured, which are the dominant power-supply hardware in embedded systems.

1.2.3 Dynamic Application Workloads

Finally, application workloads fluctuate widely, causing high temperature fluctuations and peak temperatures. In embedded systems, workloads in response to sporadic user activities exhibit a bursty pattern for user-interactive applications. Different application contexts (e.g., driving contexts such as highway/urban driving

Table 1.1: Selected state-of-the-art solutions.

System	Ambient Temperature	Hardware Variabilities	Dynamic Workloads	Deployment
[26, 30, 31, 51, 77, 119]	-	-	-	Simulation
[13, 68, 70, 78, 127]	-	-	✓	Simulation
[43, 63, 99, 107, 112]	-	✓	-	OS change
[28, 54, 55, 87]	✓	-	-	Simulation

or parking) also create large variations in workloads. Such dynamic workloads must be captured to effectively predict/regulate peak temperatures, thereby meeting thermal requirements.

1.3 State of the Art

Extensive studies have been conducted on thermal management [76] at both the hardware- (e.g., architecture design, floorplan, and hardware throttling) and software-level (e.g., task assignment, scheduling, and idle insertion). Thermal-aware dynamic voltage/frequency scaling (DVFS) and task scheduling have been active subjects of research attempting to meet timing and thermal constraints. DVFS scheduling determines the voltage and frequency of a processor to minimize power consumption [15, 129] and the peak temperature subject to timing constraints on single-core [30, 31, 119] or multi-core platforms [26, 51]. These solutions, however, do not deal with the problems of a changing environment, hardware thermal variabilities, and dynamic application workloads.

To address the thermal impact of dynamic workload, researchers have focused on different task-level power dissipations to reduce the peak temperature [13, 70] or maximize throughput [68, 127] through interleaving the execution of hot and cold tasks. Through analyzing such task-level power variations, the peak temperature was derived to meet the thermal constraint [13, 78].

To capture unique thermal characteristics/variabilities of the underlying hardware, researchers have analyzed a platform's temperature imbalance across CPUs [79] and CPUs–GPU [96, 107] platforms. CPUs–GPU thermal coupling is known to limit a core's maximum frequency, which is greatly affected by the GPU's heat dissipation [96,99]. Maestro [107] focused on characterizing a thermally efficient core to control the frequency in heterogeneous systems. The infrared imaging was used to characterize the CPUs–GPU thermal coupling, demonstrating thermal challenges in an integrated CPU and GPU [43]. Singla *et al.* [112] proposed a thermal modeling methodology through system identification for a heterogeneous mobile platform. A few researchers have considered battery characteristics from the system perspective. Xie *et al.* analyzed thermal coupling in smartphones between the system and battery [120]. B-MODS [63] used battery-aware intermittent discharge patterns on mobile devices to exploit the battery relaxation effect.

Several researchers have considered dynamic environments and developed adaptive thermal management to meet both thermal and QoS requirements. Feedback controller approaches were proposed to regulate the processor temperature by adjusting the processor utilization [54] or operating frequency [55] subject to timing constraints. Furthermore, a few online scheduling algorithms have been proposed to enhance the thermal reliability of homogeneous [28, 87] and heterogeneous [86] multiprocessor platforms.

Unfortunately, state-of-the-art solutions fall short in adaptively managing dynamic thermal behaviors, providing a deployable solution and/or satisfying both QoS and thermal requirements. Many of the existing approaches, however, have been evaluated using thermal simulations but have neither been implemented nor tested with realistic platforms and workloads.

Table 1.2: Summary of this thesis' contributions.

System	Target System	Ambient Temperature	Hardware Variabilities	Dynamic Workloads	Deployment
BPM [1]	Mobile	✓	✓	✓	User-level service
eTEC [2]	Mobile	✓	✓	✓	User-level service + cooling device
RT-TRM [3]	Automotive	✓	-	✓	User-level service + OS change
RT-TAS [4]	Automotive	-	✓	✓	User-level APIs

1.4 Thesis Statement and Contributions

Although existing research work considered thermal challenges, embedded systems in the wild still lack effective and adaptive solutions to cope with extreme thermal conditions. An effective thermal management solution requires understanding the dynamic thermal behaviors of application workloads and the underlying hardware to ensure thermal and application QoS requirements, and moreover, an adaptive one must cope with various runtime thermal scenarios under the changing ambient temperature.

Thesis Statement: The thermal management systems for embedded applications developed in this thesis meet both thermal and QoS requirements under (i) changing ambient temperature, (ii) platform thermal variabilities and (iii) dynamic application workloads.

In this thesis, we improve on the state of the art by proposing a set of effective and adaptive thermal management systems span various environment and applications (Table 1.2) that target mobile devices: BPM and eTEC; and automotive systems: RT-TRM and RT-TAS. Each of these thermal management systems advances the state of the art in dynamic thermal management by addressing different aspects of dynamic thermal behaviors. These thermal management systems meet thermal and QoS requirements, and thus provide a practical and deployable solution.

1.4.1 BPM

Many users have reported experiencing unexpected shutoffs of their mobile devices, such as smartphones and tablets, even when the device battery is shown to have >30% remaining capacity. After examining the problem from both the user and device sides, we have discovered the cause of such unexpected shutoffs to be the large and dynamic voltage drop across the device battery's internal impedance, which, in turn, is caused by the dynamics of mobile devices' power supply and demand: (i) a battery's dynamic internal impedance, which varies with the state-of-charge (SoC), temperature, and age, together with the device's bursty discharge current, cause a voltage drop across the battery's impedance to fluctuate; (ii) this drop reduces the voltage supplied to the device, and if the reduction is too large, this shuts off the device even before the battery is fully drained.

To fix such unexpected shutoffs, we designed a novel battery-aware power management (BPM) middleware for mobile devices. This middleware accounts for the dual-dynamics of device operation — capturing the dynamic battery impedance and adaptively controlling the device's dynamic runtime discharge current — thereby regulating the battery's voltage drop and achieving reliable and extended device operation. Specifically, BPM profiles the battery impedance at different SoCs and temperatures using a novel duty-cycled charging method, and then regulates, at runtime, the discharge current based on the thus-constructed battery profile. We implemented and evaluated BPM on four commodity smartphones from different original equipment manufacturers (OEMs), demonstrating that BPM prevents unexpected device shutoffs and extends the device operation time by 1.16–2.03×.

1.4.2 eTEC

In this project, we investigated device overheating, where underlying processors were throttled to cool devices, causing mobile apps to suffer significant degradation in performance. Fans or heat sinks are not a viable option for mobile devices, which calls for a new portable cooling solution. *Thermoelectric coolers* (TECs) are scalable and controllable cooling devices that can be embedded into mobile devices on the chip surface. This project presents a thermoelectric cooling solution that enables efficient thermal management of processors in mobile devices. Our goal was to minimize performance loss from thermal throttling by efficiently using thermoelectric cooling. Because mobile devices experience large variations in workload and ambient temperature, our solution adaptively controls cooling power at runtime. Our evaluation on a smartphone using mobile benchmarks demonstrated that the performance loss from the maximum speed was only 1.8% with a TEC compared to 19.2% without the TEC.

1.4.3 `RT-TRM`

Whereas `BPM` and `eTEC` focus on mobile devices, we subsequently focused on embedded real-time applications such as automotive systems. For real-time automotive systems, we demonstrated the importance of accounting for *dynamic ambient temperature* and *task-level power dissipation* in resource management to meet both thermal and timing constraints. To address this problem, we proposed a real-time thermal-aware resource management (`RT-TRM`) framework. We first introduced a task-level dynamic power model that could capture different power dissipations with a simple task-level parameter called the *activity factor*. We then developed two new mechanisms, *adaptive parameter assignment* and *online idle-time scheduling*. The former adjusts voltage/frequency levels and task periods according to the varying ambient temperatures while preserving feasibility. The latter generates a schedule by allocating idle times efficiently without missing any task or job deadlines. By tightly integrating the solutions of these two mechanisms, we could guarantee both thermal and timing constraints in the presence of dynamic ambient temperature variations. We implemented RT-TRM on an automotive microcontroller to demonstrate its effectiveness, improving resource utilization by 18.2% over other runtime approaches while simultaneously meeting both thermal and timing constraints.

1.4.4 RT-TAS

Because modern cars perform real-time vision processing using high-performance CPUs–GPU system-on-chip (SoC), they face greater thermal problems than before, which in turn cause higher failure rates and cooling costs. We used a representative vision platform to demonstrate the importance of scheduling the CPU and GPU while accounting for their thermal coupling, which incurred significant temperature imbalance on the platform. To address this problem, we proposed RT-TAS, a real-time thermal-aware CPUs–GPU scheduling framework. We first developed a CPUs–GPU thermal coupling model that could capture the different CPU temperatures caused by GPU power dissipation. We then used the model for *thermally-balanced* task-to-core assignment and *CPU–GPU co-scheduling*. The former addresses the platform's temperature imbalance by assigning tasks efficiently to cores while preserving scheduling feasibility. The latter, building on the thermally-balanced assignment, co-schedules the CPU and GPU to mitigate peak total power dissipation without missing any task deadlines. We implemented and evaluated RT-TAS on a representative automotive vision platform to demonstrate its effectiveness, achieving a 1.52× improvement in platform lifetime or savings on cooling cost of US$ 16.4 over other approaches by reducing the maximum temperature while meeting timing constraints.

CHAPTER II

BPM: Battery-Aware Power Management

2.1 Introduction

Many users are reported to have suffered the unexpected shutoffs of their mobile devices — even when the device batteries are shown to have >30% remaining capacity — on both Android [66] and iOS platforms [115], especially in cold environments. These unexpected shutoffs prevent users from making and receiving important calls and texts even when the device batteries have sufficient remaining capacity. Apple introduced an update to iOS 10.2.1 to remedy the unexpected shutoff of iPhones with aged batteries. This update, however, (i) did not fully address the underlying issue [115], and (ii) slowed the phone noticeably [6], leading to multiple lawsuits against Apple [92].

Causes: Large and Dynamic Internal Voltage Drop. Our experiments showed the cause of such unexpected device shutoffs to be a large voltage drop across the battery's internal impedance, causing an insufficient voltage supplied to the device[1], and thus shutting off the device.[2] The internal voltage drop is determined by the battery's impedance and the device's discharge current, both of which vary

[1] Mobile devices require a minimum voltage (e.g., V_{bat}>3.4V) to operate.

[2] Mobile devices may shut off when their batteries/chips are overheated. These shutoffs, however, are intentionally-triggered (for safety) and well-tracked by both Android and iOS, and are thus not considered "unexpected".

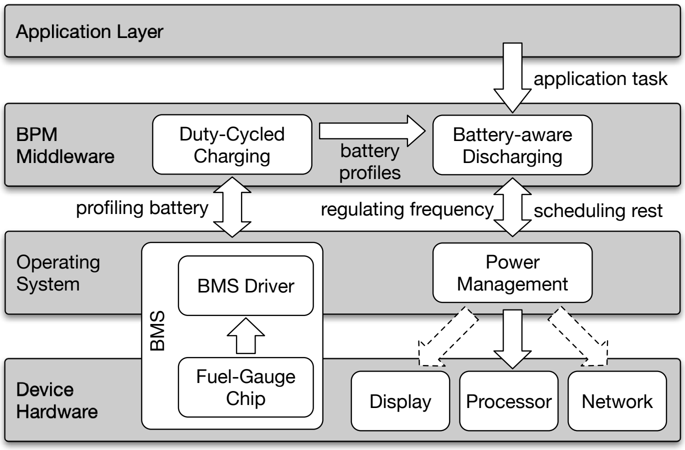


Figure 2.1: Battery-aware power management middleware.

over device operation: (i) the battery's impedance varies with the state-of-charge (SoC)[3] and rises as the battery ages or temperature falls [62], and (ii) a mobile device's discharge current is often bursty in response to user activities [102]. Such "dual-dynamics" of battery impedance and discharge current magnify the uncertainty of the battery's internal voltage drop, making it difficult to predict/regulate the battery's voltage output. Notably, the dependency of a battery's impedance on its age or the environmental temperature exacerbates the unexpected shutoffs as it ages or when it operates in a cold environment (§2.3).

Fixes: Battery-Aware Power Management (BPM). To mitigate unexpected device shutoffs, we present a novel BPMmiddleware that is compatible with commodity mobile devices and require no additional hardware — except for a typical charger — or OS modifications. BPM captures the dynamic battery impedance at different SoCs and temperatures, updates it as the battery ages, and regulates the device's runtime discharge current to ensure a sufficient voltage supply to operate the device whenever possible, thereby achieving a reliable and extended device operation. Thus, users can use their phones for longer and in a more predictable manner (§2.4.1). Specifically, BPM fixes unexpected device shutoffs with the joint management of battery charging and

[3] SoC is the percentage of remaining capacity relative to the total usable capacity when the battery is fully charged.

discharging via close interactions with the (lower) OS layer and (upper) application layer (see Fig. 2.1):

- *Duty-Cycled Charging.* BPM becomes aware of the device battery by profiling (and updating) the battery impedance at different SoC levels with novel *duty-cycled charging*: the battery is rested after being charged to a set of discretized SoC levels, and the battery is characterized at a specific SoC level based on the battery voltage during the corresponding rest period. BPM further compensates the temperature-dependency of the thus-profiled battery impedance according to the environmental temperature. This duty-cycled charging, however, requires more time to fully charge the battery. To ensure no perceivable degradation of user experience caused by the prolonged charging, BPM applies the duty-cycled charging only when the device is charging overnight (as mobile device users commonly do), and furthermore, it allows a sufficient time to fully charge/characterize the battery [64] (§2.4.2).

- *Battery-Aware Discharging.* BPM adaptively regulates the device's operation, and hence battery discharging, based on this battery-awareness. Specifically, BPM (i) estimates the battery impedance based on the above-captured impedance–SoC relationship; (ii) identifies the maximum allowed discharge current; (iii) regulates the device's discharge current below the allowed maximum by limiting the maximum processor frequency; and then (iv) schedules rest periods between consecutive device operations to restore the battery voltage before executing the next operation. BPM uses the processor frequency and scheduling as control knobs to regulate the discharge current, instead of the operation of other device modules, such as display and networking (see Fig. 2.1). This was based on our empirical observation that the major dynamics of a device's discharge current are attributed to the processor (§2.4.3). Moreover, BPM limits the maximum processor frequency only to the necessary degree, thereby minimizing degradation in user-perceived experience. Our experiments showed that BPM prevented unexpected device shutoffs at a cost of only a 1.1% reduction in processor frequency, whereas the iOS 10.2.1 update reduces the processor frequency by 9.1% according to the empirical measurements in [6].

We implemented and evaluated BPM on the following smartphones: two Nexus 5X, one Nexus 6P, and one Pixel (§2.5). Our experimental results demonstrated BPM to (i) prevent unexpected device shutoffs, and (ii) extend device operation by 1.16–2.03× compared with the default battery saver mode of these devices.

Contribution : . This chapter makes the following main contributions:

- Demonstrations of the root causes of unexpected mobile device shutoffs and the importance of accounting for devices' dynamic power supply and demand together to avoid unexpected shutoffs.
- Design of BPM, a novel battery-aware power management middleware that fixes unexpected device shutoffs and extends device operation (§2.4).
- Implementation (§2.5) and evaluation of BPM on four commodity mobile devices from different OEMs (§2.6).

2.2 Background

Presented in the following subsection is the necessary background information of mobile device batteries and their management.

2.2.1 Mobile Devices and Their Batteries

Batteries are used to power the hardware components of a mobile device, such as processors, displays, and communication modules, with the typical architecture

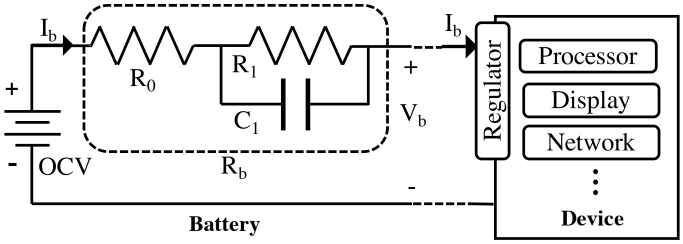


Figure 2.2: Equivalent circuit model of a mobile device: the device will shut off when the battery voltage V_b is lower than the minimum level required by the device.

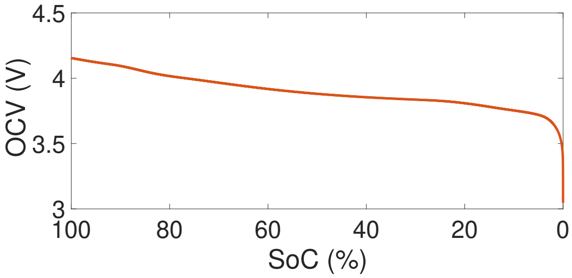


Figure 2.3: The OCV-SoC relationship of batteries.

illustrated in Fig. 2.2. The device battery is abstracted by an equivalent circuit model (the left part of Fig. 2.2) consisting of [123]:

1. An ideal voltage source, providing the battery's *open-circuit voltage* (OCV), defined as the voltage between its terminals without loads/charger connected. A battery's OCV has a monotonic relationship with the battery's remaining capacity (see Fig. 2.3, which presents the example of the battery of a Nexus 5X), which is the basis for SoC estimation commodity mobile devices [59].

2. A resistor–capacitor network (i.e., R_0, R_1, C_1), which we call the battery's *internal impedance* R_b.

When the battery discharges current I_b, the serial resistance R_0 causes an instant voltage drop

$$\Delta V_{inst.} = I_b \cdot R_0. \tag{2.1}$$

The parallel connection of R_1 and C_1 further triggers a gradual voltage drop of

$$\Delta V_{trans.}(t) = I_b \cdot R_1 - R_1 \cdot C_1 \frac{dV_b(t)}{dt}, \tag{2.2}$$

which converges (i.e., when $\frac{dV_b(t)}{dt}$=0) at

$$\Delta V_{trans.} = I_b \cdot R_1. \tag{2.3}$$

A combination of Eqs. (2.1) and (2.2) shows that the battery's output voltage $V_b(t)$ can be described as

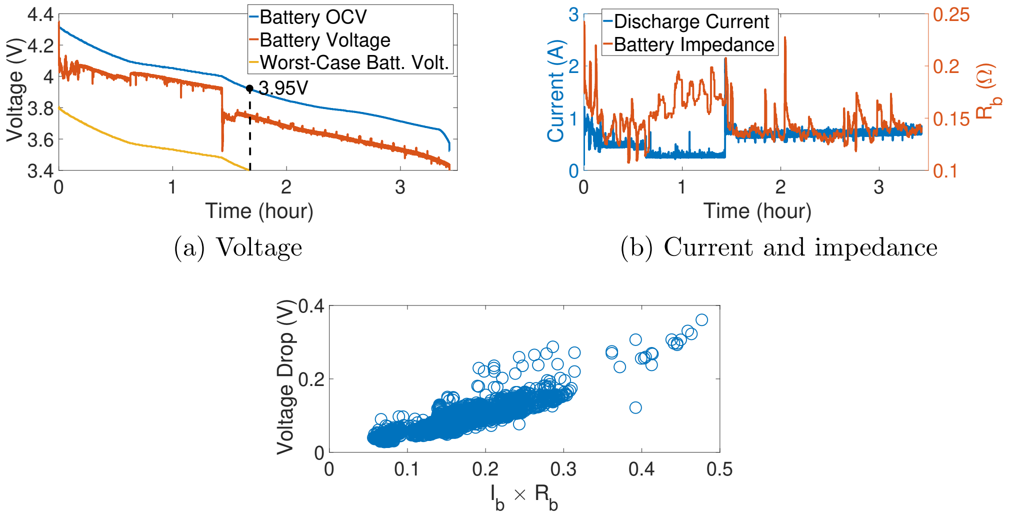
$$\begin{aligned} V_b(t) &= OCV(SoC) - \Delta V_{inst.} - \Delta V_{trans.}(t) \\ &= OCV(SoC) - (R_0 + R_1) \cdot I_b + R_1 \cdot C_1 \cdot \frac{dV_b(t)}{dt}. \end{aligned} \tag{2.4}$$

Note that by defining the discharge/charge current as positive/negative values, Eqs. (2.1)–(2.4) hold for charging devices as well.

The mobile device on the right of Fig. 2.2 requires the minimum input voltage V_b^{cutff} to operate (e.g., 3.3–3.4V for mobile phones [63]), otherwise the device will shut off. This cutoff voltage — usually implemented using voltage regulators [62] — ensures a sufficient voltage to power hardware components and avoids the deep discharging of the battery, which accelerates battery degradation.

2.2.2 Battery Management of Mobile Devices

The battery management system (BMS) of commodity mobile devices consists of a fuel-gauge chip and the BMS driver/firmware in the OS (see Fig. 2.1). The fuel-gauge chip monitors the battery information in real time, such as the voltage, current, and temperature. The BMS driver/firmware then estimates advanced battery information such as SoC and battery health using this raw information [17]. The OS displays



(a) Voltage

(b) Current and impedance

(c) Voltage drop

Figure 2.4: Operating a Nexus 5X phone: playing a video, idling, and then playing a game until shutoff.

this battery information to users and takes coarse-grained actions (e.g., enabling the battery saver mode or disabling the camera) when the battery's remaining capacity is low. The OS also maintains device/battery usage statistics to calculate the power usage of each app or device module, and uses them to adjust the processor frequency through dynamic voltage frequency scaling (DVFS).

2.3 Causes of Unexpected Device Shutoffs

With the understanding of the power architecture of mobile devices, this section analyzes and validates the causes of unexpected device shutoffs.

Device Operation and Shutoff. We first use the empirical traces shown in Fig. 2.4 to illustrate how mobile devices operate,[4] from which we make the following three key observations.

O1. The battery voltage decreases during the phone's operation until it reaches

[4] These traces were collected with a Nexus 5X phone in a room temperature, during which the phone was used to play a Youtube video (i.e., the first 38 minutes), kept idle (i.e., 38–86 minutes), and then play a game until shutoff (i.e., 86–206 minutes).

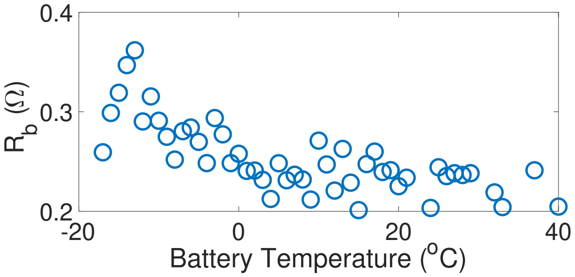


Figure 2.5: Battery impedance rises as temperature falls.

approximately 3.4V, at which point the phone shuts off, as shown in Fig. 2.4(a).

O2. Both the discharge current and battery impedance vary during device operation (see Fig. 2.4(b)).

O3. The internal voltage drop of the battery — that is the difference between the "Battery OCV" and "Battery Voltage" in Fig. 2.4(a) — depends on both the discharge current and impedance. This can be observed in Fig. 2.4(c), where the voltage drop (i.e., y-axis) and the term $I_b{\cdot}R_b$ (i.e., x-axis) are calculated from Fig. 2.4(a) and (b), respectively. This can also be derived from Eq. (2.4): the battery's output voltage V_b is determined by its internal resistance and capacitance (i.e., R_0, R_1 and C_1) and discharge current I_b. Note the markers in Fig. 2.4(c) are below the line of $y = x$ because of the insufficient time for the battery voltage to be stabilized during this measurement (i.e., $dV_b(t)/dt{>}0$ in Eq. (2.2)) — the collected voltage drop has not yet reached its maximum.

These observations led to our conjecture that *a large voltage drop over a battery's internal impedance may reduce the battery voltage too much to power the device, thereby causing unexpected device shutoffs.* This large voltage drop is likely to occur in practice because of a dynamically changing battery impedance and discharge current, especially in view of the fact that the battery impedance also varies with temperature T_b; that is, the impedance rises as the temperature falls, as shown in Fig. 2.5 with the Nexus 5X.

Assuming this conjecture holds, the "Worst-Case Battery Voltage" in Fig. 2.4(a) plots the lower-bound of the battery voltage, meaning the lowest possible voltage without shutting the device off. This is derived using

$$V_{\text{worst}} = OCV(SoC) - \max\{I_b\} \cdot \max\{R_b\}, \tag{2.5}$$

where $\max\{I_b\}$ and $\max\{R_b\}$ are extracted from Fig. 2.4(b), showing that the phone may shut off with an OCV as high as 3.95V, which maps to (according to Fig. 2.3) an SoC of nearly 70%.

Case Studies of Unexpected Shutoffs. To corroborate this conjecture, we conducted case studies to trigger unexpected shutoffs of a Nexus 5X phone by magnifying the voltage drop across its battery's impedance (i.e., $I_b{\cdot}R_b$). Specifically, we operated a fully-charged Nexus 5X phone in a freezer (-5°C) with the User Interaction (UI) exerciser [8] on until it shut off (thus with increasing R_b and $I_b{\cdot}R_b$), warmed it in room temperature (thus R_b and $I_b{\cdot}R_b$ decreased), and then attempt to turn it on and operate it further without charging. Fig. 2.6 plots the discharge current, battery impedance, voltage drop, and battery voltage supplied to the phone during this measurement, showing that,

- the discharge current was highly dynamic/bursty;
- the battery's internal impedance rose as the temperature fell;
- the phone shut off when the voltage dropped to approximately 3.4V, but then it was successfully turned back on after being warmed in room temperature — delivering another 330mAh capacity or operating for an additional 18 minutes — without having its battery charged. The battery's voltage drop before the unexpected shutoff was 0.49V, which reduced to 0.14V after being warmed in room temperature (e.g., at the 88th minute) and thus we were able to turn the

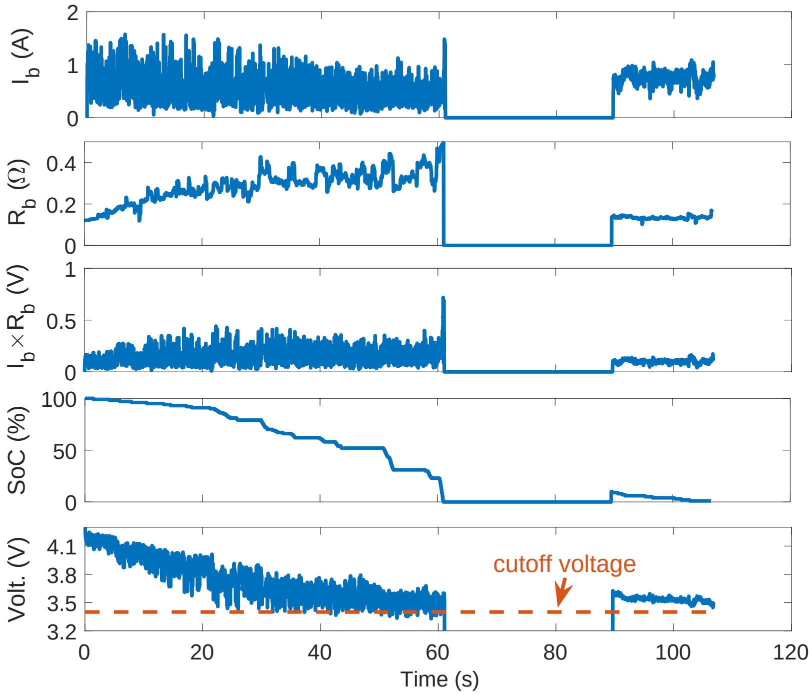
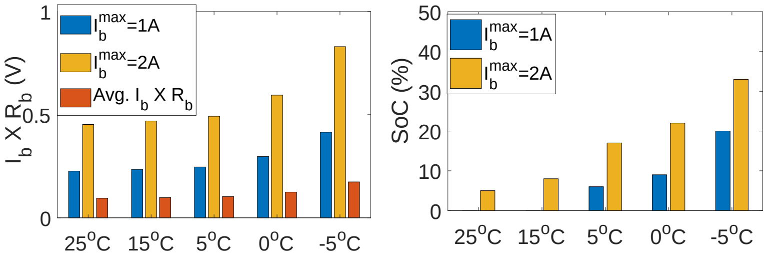


Figure 2.6: Unexpected shutoff of a Nexus 5X smartphone in a cold ambient temperature.

phone on again.

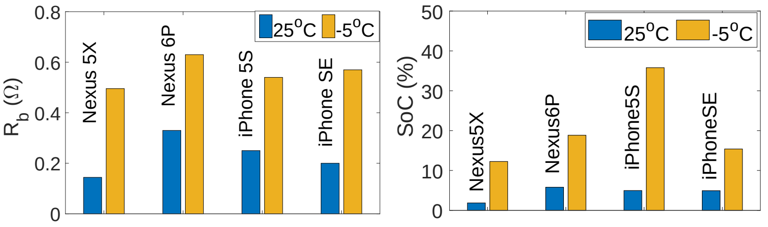
Next, we repeated the experiments while varying the ambient temperature from -5 to 25°C and the maximum discharge current from 1 to 2A. The results are plotted in Fig. 2.7, and show that (i) unexpected shutoffs were observed at all explored temperatures, and (ii) the voltage drop increased and the phone shut off with up to 33% SoC when discharging with a large current in cold ambient temperatures. We have conducted similar experiments with a Nexus 6P and iPhone 5S and SE[5] and made similar observations, as summarized in Fig. 2.8. Note the iPhone SE had the iOS 10.2.1 update to prevent unexpected shutoffs, whereas the iPhone 5S did not. Although this update reduced the unexpected shutoffs at -5°C (i.e., from 35% SoC on iPhone 5S to 15% SoC on iPhone SE), the problem still persisted. More importantly, the update degraded the phone performance significantly, for example, it reduced the

[5] All these devices are within their battery warranty (e.g., 500 complete charge/discharge cycles).



(a) Voltage drop (b) SoC at shutoff

Figure 2.7: Voltage drop at shutoff vs. the average voltage drop and SoC at shutoff of a Nexus 5X battery.



(a) Impedance at shutoff (b) SoC at shutoff

Figure 2.8: Battery impedance and SoC when different mobile devices shut off.

average processor frequency by 9.1% [6].

These case studies confirmed our conjecture that *a large voltage drop across the battery's internal impedance (i.e., $I_b{\cdot}R_b$) causes unexpected device shutoffs*, which are prevalent across Android and iOS devices.

2.4 Fixes for Unexpected Device Shutoffs

These causes of unexpected device shutoffs also inspire their remedy, i.e., regulating the voltage drop across battery impedance $I_b{\cdot}R_b$, where both I_b and R_b vary.

2.4.1 Overview

As mobile devices have little control over their battery's internal impedance R_b, `BPM` regulates $I_b{\cdot}R_b$ by actively limiting I_b based on the real-time estimation of R_b.

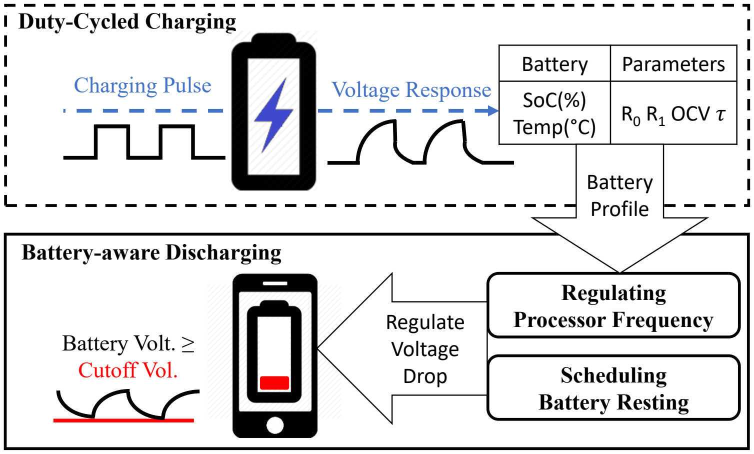
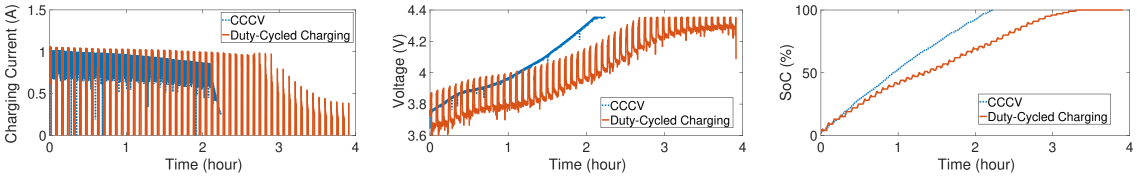


Figure 2.9: BPM profiles the device battery during charging and regulates the voltage drop during discharging.

Specifically, BPM uses (i) duty-cycled charging management to profile the dynamic battery characteristics thereby facilitating the real-time estimation of R_b, and (ii) battery-aware discharging management to regulate the device's discharge current I_b at runtime (see Fig. 2.9). During battery charging (dotted line), BPM charges the device with a duty-cycled current followed by a rest period, and then determines the battery parameters — i.e., $<OCV, R_0, R_1, C_1>$ — at each SoC based on the voltage observed during the rest period. At runtime, BPM further compensates these battery parameters based on the environmental temperature using a temperature dependency model (§2.4.2). During discharging (solid line), BPM (i) estimates the runtime battery impedance, (ii) identifies in real time the maximum allowed discharge current based on battery impedance, (iii) determines the thus-allowed maximum processor frequency, and (iv) allocates a rest period between operations to restore the battery voltage — using the recovery effect of batteries [63] — before executing the next operation (§2.4.3).

Note that the BPM middleware implements the duty-cycled charging and discharging management by leveraging readily available BMS and the DVFS drivers of commodity mobile devices, and thus requires no special hardware — except a



(a) Discharge Current (b) Voltage (c) SoC

Figure 2.10: BPM's duty-cycled charging vs. standard CCCV charging.

typical charger — or OS modifications (§2.5).

2.4.2 Profiling Batteries During Charging

BPM profiles the battery's parameters as functions of battery SoC and temperature, and then stores them as lookup tables.

Duty-Cycled Charging. BPM constructs and updates these lookup tables by charging the devices with a customized duty-cycle: in each cycle, the battery is charged with a current I_c for a duration of t_c, and then it is rested the battery for a duration of t_r. BPM implements this duty-cycled charging by enabling/disabling the device's charging,[6] which also simplifies BPM because the charging current I_c will automatically be determined by the device's charging chip — BPM only needs to control t_c and t_r. Note that when the device's charging is disabled with the charger connected, the device's operation will be powered by the charger, thereby resting the battery.

Fig. 2.10 depicts the duty-cycled charging current, battery voltage, and SoC during BPM's charging of a Nexus 5X phone, and compares them with constant-current constant-voltage (CCCV) charging, which is commonly used in mobile devices [64]. Clearly, BPM's duty-cycled charging prolongs the time required to fully charge the battery, for example, Fig. 2.10 shows that BPM requires approximately 1.4 hours longer to fully charge the battery compared with CCCV. To preserve the user-perceived QoS,

[6] This charging control can be achieved, e.g., by configuring */sys/class/power_supply/bms/battery_charging_enable* in Android devices.

BPM applies duty-cycled charging only when mobile devices are charged overnight, which provides sufficient time to fully charge the battery and is very common for most mobile device users [64]. Moreover, resting the battery after each charging cycle slows battery aging [59].

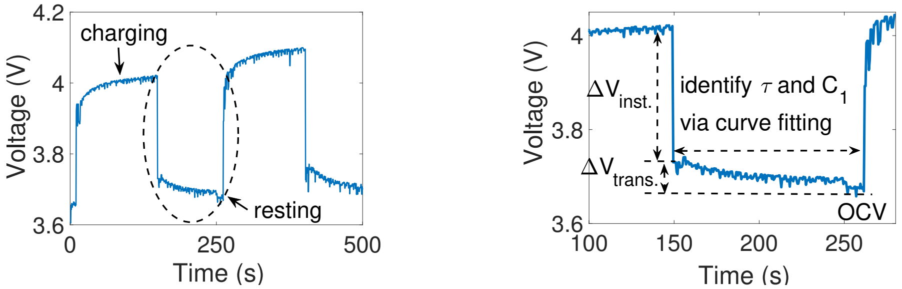
In addition, note that BPM's duty-cycled charging differs from existing pulsed charging; that is, BPM exploits the rest periods to profile the battery, as we explain next.

Battery Voltage During Resting. BPM uses the battery voltage during rest periods to estimate battery parameters at specific SoC levels. According to Eqs. (2.1)–(2.4), resting battery at time 0 after charging it with current I_c yields:

$$\begin{aligned} V_b(0^-) &= OCV - I_c \cdot (R_0 + R_1), && (2.6)\\ V_b(0^+) &= OCV - I_c \cdot R_1, && (2.7)\\ V_b(t) &= OCV - I_c \cdot R_1 \cdot e^{-\frac{t}{R_1 \cdot C_1}} \qquad (t > 0), && (2.8) \end{aligned}$$

showing the battery voltage to (i) drop instantly by $\Delta V_{inst.} = I_c{\cdot}R_0$ because of the ohmic voltage drop across R_0 (i.e., Eq. (2.6) to Eq. (2.7)), and (ii) drop gradually afterwards according to Eq. (2.8) until converged to the steady-state voltage of OCV.The term $\tau{=}R_1{\cdot}C_1$ in Eq. (2.8) is the time-constant of the R_1 & C_1 parallel network in Fig. 2.2, which describes how quickly the battery voltage stabilizes. Eqs. (2.6)–(2.8) are the basis for BPM to estimate the battery parameters $<OCV,\ R_0,\ R_1,\ C_1>$ from the battery voltage, as we describe next.

Estimating Battery Parameters Using Voltage. BPM profiles the battery parameters at a set of discretized SoC levels: $\{0\%, \Delta\%, 2\Delta\%, \cdots, 100\%\}$. BPM charges the battery with current I_c until the next SoC level is reached, rests the battery by disabling the charging for t_r, and then estimates the battery parameters at the current SoC level using the battery voltage during resting, as illustrated in



(a) Duty-Cycled Charging

(b) Zoom-in of Resting Voltage

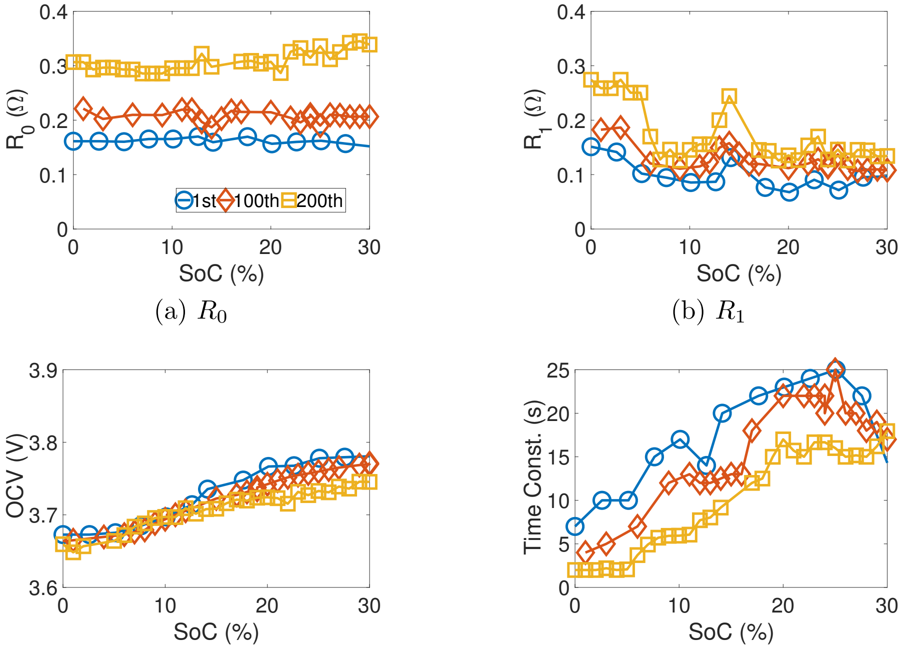
Figure 2.11: Estimating battery parameters using battery voltage during resting.

Fig. 2.11 with $\Delta = 2$ and a resting period of $t_l = 100$s. Specifically, BPM estimates the battery parameters using the resting voltage based on Eqs. (2.6)–(2.8) as follows:

- it estimates R_0 from the instantaneous voltage drop according to $R_0 = \Delta V_{inst.}/I_c$;
- it estimates R_1 based on the transient voltage drop to the steady-state voltage $R_1 = \Delta V_{trans.}/I_c$;
- it estimates C_1 from the time constant ($\tau{=}R_1{\cdot}C_1$) of the voltage response via least-square curve-fitting; and
- it estimates the OCV as the steady-state voltage.

Note that the 100-s rest period in Fig. 2.11 is determined based on Eq. (2.8): the battery voltage converges to OCV at a rate of $1 - e^{\frac{t}{\tau}}$. For example, with the maximum τ of approximately 25s observed in Fig. 2.12, a 100s rest allows the voltage to converge to OCV $1 - e^{100s/25s} \approx 98\%$.

Fig. 2.12 plots the thus-estimated parameters of a battery used by a Nexus 5X, for the SoC range of $[0, 30]\%$, at the $\{$1st, 100th, 200th$\}$ charging cycles. Unlike R_0 which is relatively stable across a given charging cycle, R_1 and τ vary significantly with the SoC because of phase transitions [22], causing different voltage drops at different SoC levels even with the same discharge current. Moreover, these battery parameters change significantly over charging cycles: battery impedances increase



(a) R_0

(b) R_1

(c) Battery OCV

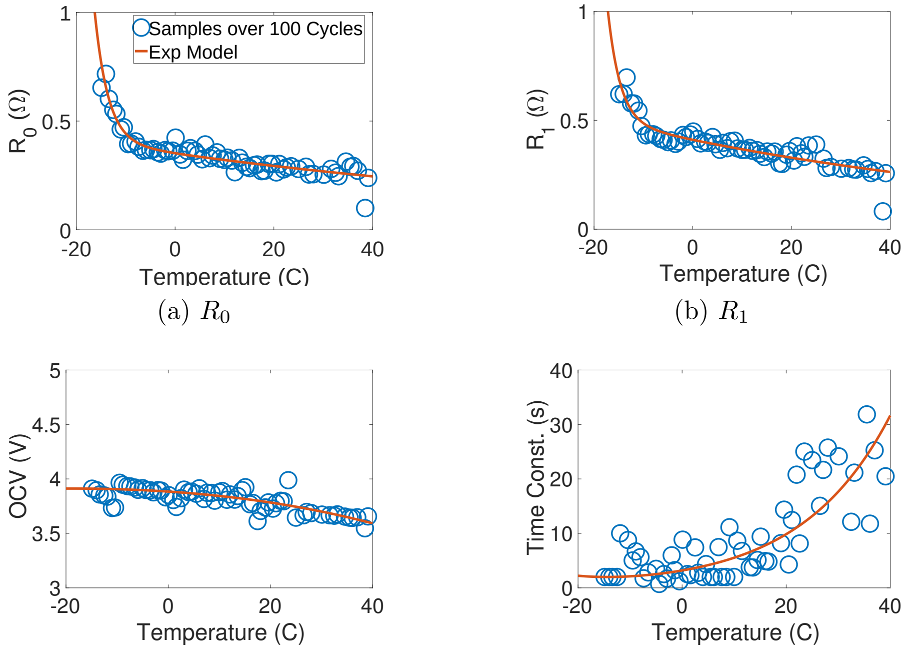
(d) Time Constant (τ)

Figure 2.12: Battery parameters estimated at different SoC levels for the 1st, 100th, and 200th charging cycles.

while the time-constant (i.e., τ in Eq. (2.8)) decreases, thereby reducing the battery power capacity over time. This explains why devices with aged batteries are likely to experience more unexpected shutoffs.

Capturing Batteries' Temperature-Dependency. Battery parameters also vary with temperature, which `BPM` must capture to facilitate the compensation of battery parameters based on the runtime environment temperature. Clearly, empirically capturing battery parameters at all potential temperature is impractical; moreover, overnight device charging usually occurs at room temperature. Thus, asking users are required to assist in profiling the parameters at different temperatures, which requests too much effort from them. To overcome this challenge, `BPM` estimates the battery parameters at various runtime temperatures based on those at room temperature using an offline constructed temperature-dependency model.[7] Specifically, we profiled

[7] Accurate offline temperature-dependency model is known to generate reliable battery parameter estimation over battery aging [116].



(a) R_0

(b) R_1

(c) Battery OCV

(d) Time Constant (τ)

Figure 2.13: Validating the temperature-dependency model over 100 cycles with Nexus 5X battery.

Table 2.1: Summary of the regression model.

Parameters	$[a_0, b_0, c_0, d_0]$	RMSE	Adj. R^2
$R_0(\Omega)$	[1.2e-4, -5.7, 1.1, -0.14]	0.051	0.979
$R_1(\Omega)$	[0.94, -0.17, 2.3e-5, -6.2]	0.044	0.968
$OCV(V)$	[1.0, -2.4e-2, -3.9e-5, -4.8]	0.095	0.946
Time Constant $\tau(s)$	[0.15, -0.47, 0.47, 0.37]	2.11	0.884

battery parameters at different temperatures, using a thermal chamber. For example, `BPM` compensates the temperature's impact on R_0 using

$$R_0(T_b) = (a_0 \cdot e^{b_0 \cdot T_b} + c_0 \cdot e^{d_0 \cdot T_b}) \cdot R_0^r(SoC\%). \tag{2.9}$$

where $R_0^r(SoC\%)$ is the R_0 at room temperature for the current SoC level ($SoC\%$) and a_0, b_0, c_0, d_0 are the regression coefficients. We then empirically examined the accuracy of the temperature-dependency model over extended battery discharging cycles. Fig. 2.13 plots the measured battery parameters (circle labels) over 100 cycles at different battery temperatures, justifying the use of a set of exponential regression

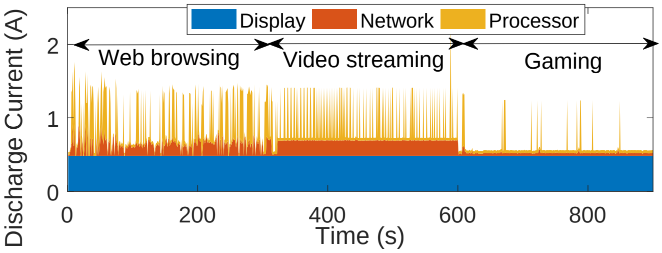


Figure 2.14: The processor incurs a burstier discharge current than other components.

models (solid line) to capture the battery's temperature-dependency. Table 2.1 summarizes these regression models and the corresponding model validation errors.

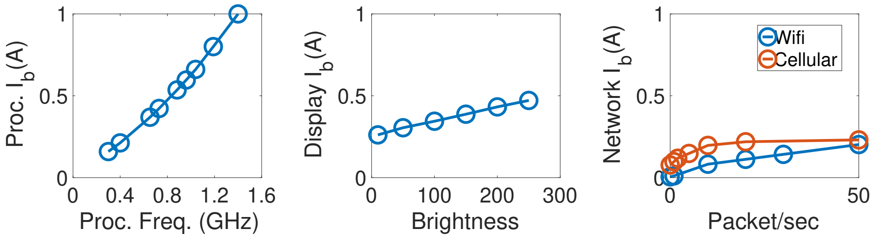
In summary, during the over-night charging process, BPM first estimates the parameters at each SoC, and then, using the temperature-dependency model, it estimates the parameters at different battery temperatures to construct a set of lookup table for battery parameters at different SoC and temperature levels.[8]

2.4.3 Regulating Battery Voltage During Discharging

BPM uses the above-constructed battery profile to mitigate unexpected shutoffs of mobile devices and extend their operation, by (i) regulating the discharge current based on real-time battery impedance through adjusting the maximum processor frequency, and (ii) restoring the battery voltage to a safe level by resting the battery before performing the next operation. BPM employs the processor frequency and scheduling as control knobs for regulating the device's discharge current because the processor dominates the dynamics thereof, as we explain below.

Modeling a Device's Discharge Current. Processor, network and display modules are the dominant energy consumers in a mobile device [124, 126]. Fig. 2.14 plots the discharge current required to run these modules on a Nexus 5X, collected with PowerTutor [126] during web browsing, video streaming, and 3D gaming.

[8] The lookup table on Nexus 5X contains the battery parameters from -20°C to 40°C battery temperature with a 0.4°C interval equal to the temperature sensor precision, and from 0% to 100% SoC with a 2% interval. The space overhead is only 0.03MB or 0.0015% of total memory.



(a) Processor (b) Display (c) Network

Figure 2.15: Component-specific discharge currents.

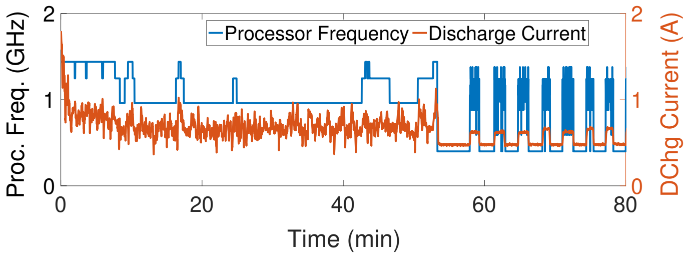


Figure 2.16: Processor frequency variations during the phone's real-life usage.

Whereas the currents drawn by the display and network modules are relatively stable, the processor's current draw varies greatly, implying that the processor dominates the dynamics of the device's discharge current. We further examined the discharge current of each module with different configurations. Specifically, Fig. 2.15 plots the collected processor discharge current at different frequencies (Fig. 2.15a); the display's discharge current at different levels of brightness (Fig. 2.15b); and the network module's discharge current at different packet transmission rates (Fig. 2.15c), showing that the processor discharge current is much more sensitive to its configuration (i.e., frequency) than those of the display and network modules. To further examine whether such dynamic processor frequencies exist in practice, we plotted the processor frequency and discharge current in Fig. 2.16 during the phone's real-life usage, confirming the dynamics of processor frequency and the thus-caused dynamics of discharge current.

Inspired by the above mentioned empirical observations, we abstracted the discharge current of mobile devices with two components: a stable background current I_{bg} and dynamic current I_{dyn}. The background current I_{bg} is contributed by components other than the processor and the idle processor leakage, and the dynamic current I_{dyn} is drawn by the active processor performing the computation. Thus, the discharge current during the busy period I_b^{busy} and the idle period I_b^{idle} can be captured using

$$I_b^{busy} = I_{dyn} + I_{bg} \qquad \text{and} \qquad I_b^{idle} = I_{bg}. \tag{2.10}$$

Furthermore, the dynamic current I_{dyn} is usually described by the dynamic power model [121, 126]:

$$I_{dyn} = V_p^2 \cdot f_p \cdot \alpha, \tag{2.11}$$

where V_p and f_p are the processor voltage and frequency,[9] and α is a scaling factor that can be empirically identified based on the relationship between discharge current and processor frequency (e.g., as shown in Fig. 2.15a) [90, 124]. Through this, we can obtain the average discharge current using the processor utilization U_p:

$$I_b = I_{dyn}(U_p) + I_{bg}. \tag{2.12}$$

Controlling Maximum Processor Frequency. `BPM` regulates the processor frequency to control the dynamic discharge current, without incurring noticeable impact on user experience (e.g., dimming of the screen in battery saver mode). `BPM` checks the constructed battery profile with the current SoC/temperature every control period to determine the maximum allowed discharge current (i.e., the cutoff current I_{cutoff}), and then determines the maximum feasible processor frequency based on I_{cutoff}.

The cutoff current is determined using Eq. (2.4) to maintain the battery voltage

[9] On commodity mobile devices, the processor voltage V_p is set based on a given frequency f_p in a pre-defined DVFS table, i.e., there is a one-to-one mapping between voltage and frequency.

above V_b^{cutff}; that is,

$$V_b(t) = OCV - (R_0 + R_1)I_b + R_1C_1\frac{dV_b(t)}{dt} \geq V_b^{cutff}. \tag{2.13}$$

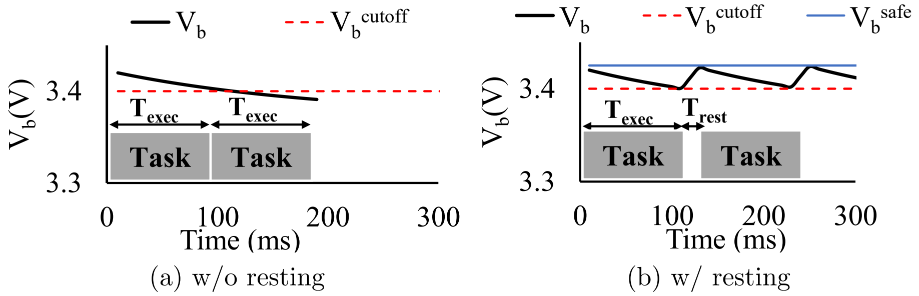
To meet the constraint, in the extreme case of $\tau\rightarrow0$ (e.g., in the low SoC levels as in Fig. 2.12(d)), we obtain

$$I_b \leq \frac{OCV - V_b^{cutff}}{R_0 + R_1} = I_{cutoff}. \tag{2.14}$$

Note that both R_0 and R_1 depend on battery SoC and temperature, making I_{cutoff} SoC/temperature-dependent. In every control period, `BPM` first identifies the dynamic and background current (i.e., I_{dyn} and I_{bg}): I_{dyn} is determined based on the current processor frequency using Eq. (2.11), and then, by sampling the processor utilization U_p and discharge current I_b, `BPM` estimates I_{bg} based on $\{I_b, I_{dyn}, U_p\}$ using Eq. (2.12). `BPM` then identifies the maximum processor frequency that regulates the discharge current below I_{cutoff}, by plugging the thus-obtained I_{bg} and U_p into Eq. (2.12). This way, `BPM` allows the processor to run at the maximum available frequency when the battery voltage is high, and adaptively reduces the maximum processor frequency to the required degree when the battery is low. Additionally, `BPM` is compatible with existing low-power DVFS schemes because it only limits the maximum processor frequency, within which the processor frequency can still be dynamically adapted to the workload.

Moreover, `BPM` must determine its control period. Inspired by the fact that the battery voltage changes gradually with the time-constant $\tau{=}R_1{\cdot}C_1$ in Eq. (2.8), we used the time-constant for the current SoC level as the control period.

Resting the Battery to Restore Voltage. On top of regulating the OS-layer processor frequency, `BPM` further captures application-level task executions and schedules a rest period between consecutive executions to restore the battery



(a) w/o resting

(b) w/ resting

Figure 2.17: Battery voltage with and without inserting rest periods between task executions.

voltage. Specifically, BPM schedules an idling thread with the highest priority, which is triggered upon the completion of every task to insert a rest period. Fig. 2.17 compares the battery voltage with and without rest periods inserted between task executions. While both cases have the same average discharge current, (i) a continuous workload without resting reduces the battery voltage below the operable level (see Fig. 2.17a), and (ii) by efficiently inserting rest periods (see Fig. 2.17b), the battery voltage is restored during rest periods and thus stays above the operable level.

To efficiently schedule battery resting, we need to determine when and for how long to insert such rest periods. According to Eqs. (2.4) and (2.10), we obtain two voltage levels: (i) when the processor is busy and drawing I_b^{busy}, the stable-state battery voltage is

$$V_b^{busy} = OCV - (R_0 + R_1) \cdot (I_b^{dyn} + I_b^{idle}), \tag{2.15}$$

and (ii) when the processor is idle, the battery voltage recovers to

$$V_b^{idle} = OCV - (R_0 + R_1) \cdot I_b^{idle}. \tag{2.16}$$

Clearly, no rest time distribution is required if $V_b^{busy} \geq V_b^{cutoff}$. Let T_{exec} be a

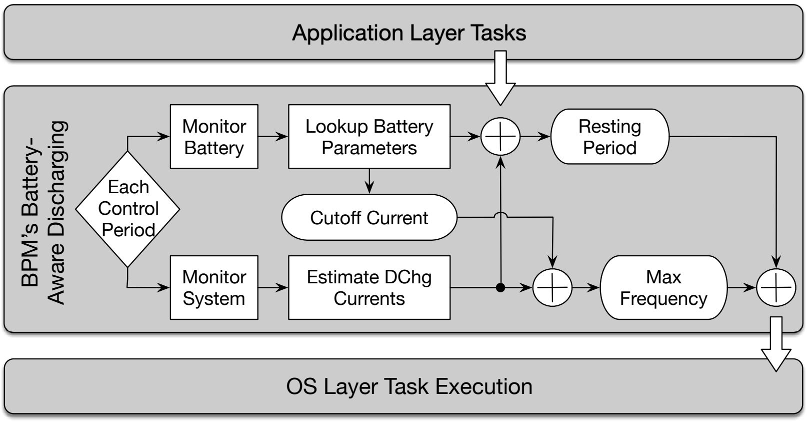


Figure 2.18: Control flow of BPM's battery-aware discharging management.

task's execution time.[10] BPM first identifies the safe voltage V_{safe} that allows the task execution without dropping the voltage below V_b^{cutoff}, as illustrated in Fig. 2.17,

$$V_b(T_{exec}) = (V_{safe} - V_b^{busy}) \cdot e^{\frac{-T_{exec}}{R_1 \cdot C_1}} + V_b^{busy} = V_b^{cutff}. \tag{2.17}$$

Then, we can find the rest period that can recover the battery voltage to V_{safe}:

$$V_b(T_{rest}) = (V_b^{cutff} - V_b^{idle}) \cdot e^{\frac{-T_{rest}}{R_1 \cdot C_1}} + V_b^{idle} = V_b^{safe}. \tag{2.18}$$

In this manner, BPM determines the rest period T_{rest} based on T_{exec}, and inserts it before executing the next task. Taking the task of user touch interaction as an example — including initiating user input and the corresponding processing/communication — BPM inserts the rest period between UI tasks by calculating the rest period using Eq. (2.18). It then inserts such a rest period by scheduling an idling thread before executing each task. With a 108ms median execution time of UI tasks (as we will see in Sec. 2.6.3), the rest period calculated by Eq. (2.18) ranges from 4.7 to 15.8ms depending on the battery impedance.

Summary. Fig. 2.18 illustrates the control flow of BPM's battery-aware discharging.

[10] The execution time of each task can be acquired from app log.

BPM collects the battery information at the beginning of each control period, identifies the cutoff current based on this information, and regulates the processor frequency in the OS layer accordingly. Furthermore, BPM encapsulates an app task by appending a rest period before the task, before passing the encapsulated task to the OS layer for execution.

2.5 BPM Implementation

We implemented the BPM middleware as a user-level background service on an unmodified Android kernel, which automatically starts upon the device being turned on. We summarize a few implementation details as follows. Specifically, BPM:

- monitors and records battery voltage, current, SoC, and temperature from *voltage_now, current_now, capacity*, located at */sys/class/power_supply/bms/*;
- generates charging pulses by disabling/enabling the charging flag *charging_enable*, located at */sys/class/power_supply/battery/*;
- limits the maximum CPU frequency at */sys/devices/system/ cpu/cpufreq/scaling_max_freq*; and
- inserts a rest period by scheduling an idling thread with the highest priority, using the priority-based scheduling policy *sched_setscheduler*$(SCHED\_FIFO)$.

In addition, BPM stores the constructed battery profiles as a set of lookup tables. Our implementation/evaluation showed that the battery-aware discharging management incurs a runtime overhead per core of only 0.16% on average.

2.6 Evaluation

We evaluated BPM on mobile devices with various battery cycles: two Nexus 5X at the 143rd and 263rd cycle, respectively; one Nexus 6P at the 414th cycle; and one

Pixel at the 15th cycle. All these batteries were within the typical warranty (e.g., 500 cycles) and replacement period (e.g., 2–3 years). The obtained results are highlighted as follows.

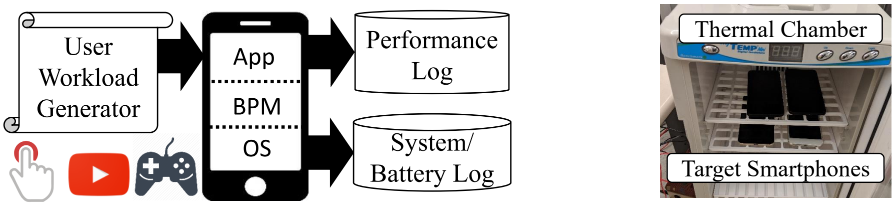
- With BPM, the devices shut off when showing an SoC close-to-0%, validating BPM's effectiveness at preventing unexpected device shutoffs (§2.6.2).
- BPM facilitates devices extracting more of their battery capacities, thereby achieving extended device operation, especially for devices powered by aged batteries (§2.6.3).
- BPM's advantage is more pronounced at low temperatures or on aged devices (§2.6.4).

2.6.1 Methodology

To evaluate BPM in various real-life scenarios, we emulated realistic user activities using representative mobile apps. Specifically, we considered three typical mobile apps:

- **UI Exerciser (UI):** generating a sequence of events emulating user operations, such as touch events and app launching [8];
- **YouTube video streaming (Video):** playing a video using *YouTube*;
- **3D gaming (Game):** playing a 3D game called *FarmVille*, which has 10M+ downloads.

Fig. 2.19 shows the overview and setup of our experiments using a thermal chamber. We emulated the user workload using UI/App Exerciser [8] and logged the app performance and system/battery information, to compare the battery operation and system/app performance with and without BPM. Without BPM, the phone's default battery saver mode is used when the battery is low, in which case (i) the interactive DVFS lowers the processor frequency to the minimum level and only raises it in response to user activities [102]; and (ii) the location service and



(a) Overview (b) Setup

Figure 2.19: Experimental setup.

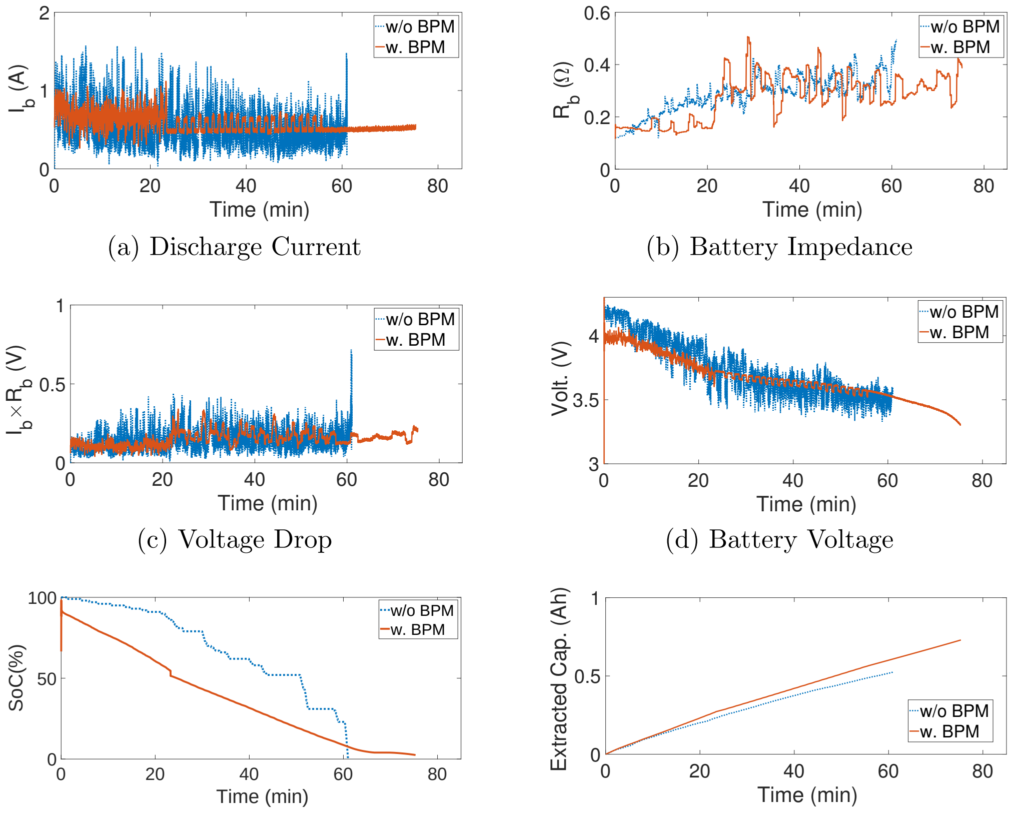
background sync are disabled, and the phone waits until the user activates an app (e.g., email or news) to refresh its content. Unless otherwise specified, we ran a full discharging cycle from 100% SoC to device shutoff while executing one of the above apps at a constant ambient temperature.

2.6.2 Preventing Unexpected Device Shutoffs

We first validated `BPM`'s effectiveness at preventing unexpected device shutoffs. Specifically, we repeated the experiments in Fig. 2.6; that is running UI exerciser on a Nexus 5X in a cold ambient temperature, with and without `BPM`. Again, this cold ambient temperature was to facilitate triggering unexpected shutoffs without `BPM` enabled. Fig. 2.20 plots the (a) discharge current, (b) battery impedance, (c) voltage drop across battery impedance, (d) battery voltage supplied to the phone, (e) battery SoC, and (f) discharged capacity during a full discharge cycle, from which two observations were obtained.

First, without `BPM`, the discharge current fluctuated significantly due to OS-level power management, because the processor frequency increased as the workload rose without awareness of battery impedances. The peak current at approximately 61min caused an excessive voltage drop across the battery impedance, reducing the battery voltage to below the cutoff level and thus shutting off the phone when the battery had an SoC of 23%.

Second, `BPM` adaptively regulated the discharge current based on the increasing



(a) Discharge Current (b) Battery Impedance

(c) Voltage Drop (d) Battery Voltage

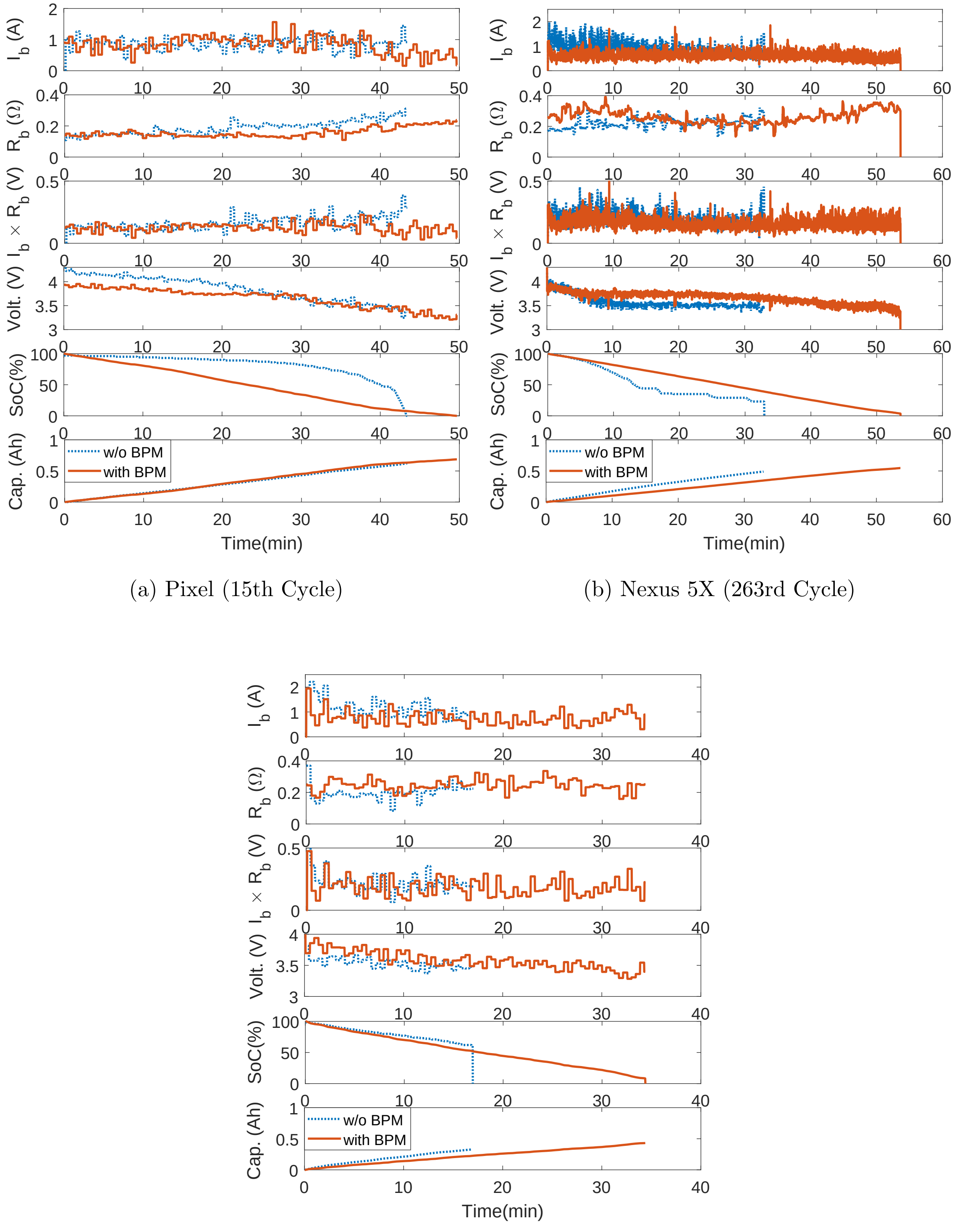
(e) Battery SoC (f) Extracted Capacity

Figure 2.20: Operating a Nexus 5X (143rd cycle) until it shuts off, with and without BPM.

impedance of the battery (caused by a cold temperature), thereby mitigating the sudden and significant voltage drops. Specifically, with BPM, the device:

- shut off when the battery SoC reduced steadily to 0%, thereby preventing the unexpected shutoff;
- extracted nearly 730mAh more capacity from the battery to support its operation, a $730/1897 = 38.4\%$ improvement over the case of without BPM;
- operated 79min before it shut off; that is, $79/61 \approx 1.3\times$ of that without BPM.

To further corroborate BPM's effectiveness for different phones, we repeated similar experiments with a Google Pixel with a battery at the 15th cycle, a (second) Nexus 5X with a battery at the 263th cycle, and a Nexus 6P with a battery at the

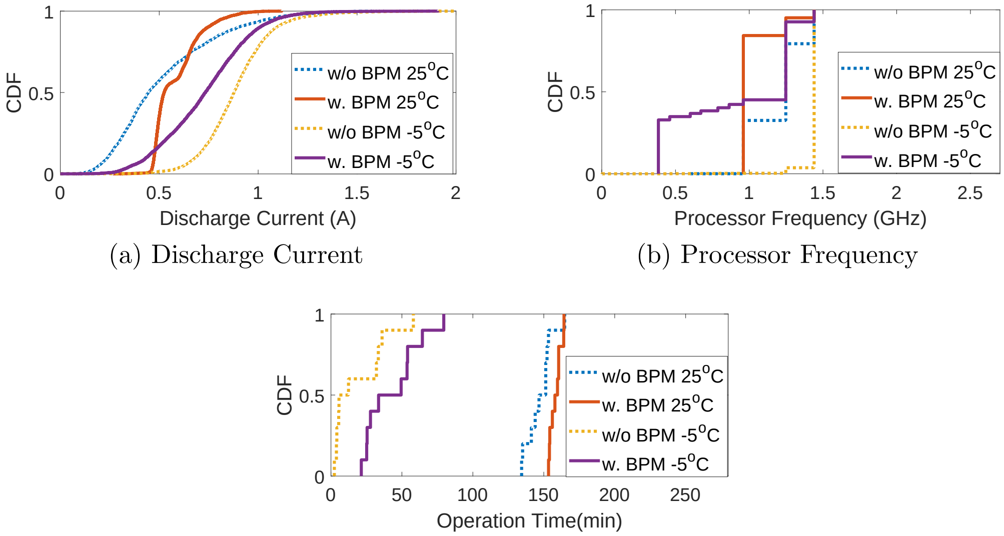


(a) Pixel (15th Cycle)

(b) Nexus 5X (263rd Cycle)

(c) Nexus 6P (414th Cycle)

Figure 2.21: BPM prevents unexpected device shutoffs and extends device operation, especially for aged devices.



(a) Discharge Current

(b) Processor Frequency

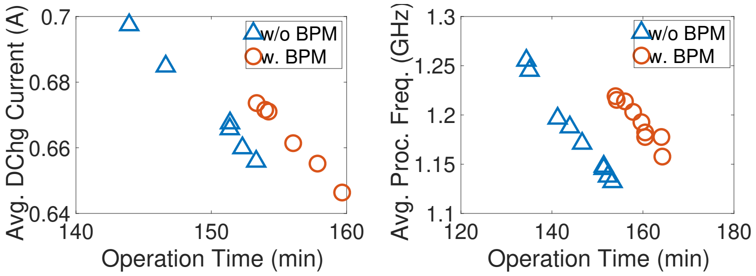
(c) Operation Time

Figure 2.22: BPM (a) reduces the peak discharge current by (b) limiting the processor frequency, thus (c) achieving extended device operation.

414th cycle. Fig. 2.21 summarizes the discharging processes, showing that BPM (i) prevented unexpected shutoffs, as demonstrated by device shutoffs when the SoC reduced steadily to approximately 0%, and (ii) extended the device operation from 43 to 50min for the Google Pixel, 33 to 54min for the Nexus 5X, and 17 to 34min for the Nexus 6P. This meant an increase up to 2.03× in device operation time, especially for those powered by aged batteries (e.g., the Nexus 6P).

2.6.3 Performance–Operation Time Tradeoff

BPM achieves the above mentioned reliable and extended device operation by limiting the processor frequency (and thus the discharge current) trading the device's computation power with its operation time. To examine this tradeoff closely, we repeated the full discharging cycle experiment 10 times on the Nexus5X running UI exerciser (in Fig. 2.20) with 25°C and a -5°C ambient temperatures, respectively. Fig. 2.22 plots the cumulative distribution functions (CDFs) of the discharge current, processor frequency, and operation time during/of these experiments. BPM reduced the



(a) Current vs. Operation Time (b) Freq. vs. Operation Time

Figure 2.23: Performance and operation time tradeoff.

peak discharge current (Fig. 2.22a) by limiting the processor frequency (Fig. 2.22b), thereby extending the device operation by up to 30 min and 17 min on average (Fig. 2.22c). In particular, BPM reduced the unpredictability of operation time by decreasing its variation by 19.8%, as well as extended the minimum operation time by 19 min. BPM achieved this at the cost of reducing the average process frequency by only 1.1% (Fig. 2.22b), whereas iOS was shown to reduce the processor frequency by 9.1% on average [6].

Fig. 2.23a plots BPM's tradeoff between the average discharge current and operation time, and then compares it with the case without BPM. Note that multiplying the average discharge current (y-axis) with the operation time (x-axis) obtains the extracted capacity. This way, the markers toward the top-right corner of the figure — the results with BPM do — indicate greater effectiveness in extracting battery power to operate the device. Fig. 2.23b plots a (similar) tradeoff between the average processor frequency and operation time, where the markers at the top-right corner again with BPM indicate a greater overall computation ability of the device before shutoff.

We further investigated whether this tradeoff causes noticeable degradation in user-perceived app performance. To examine its impact on application-level performance, we used *response latency* to quantify the user experience when running the UI exerciser (i.e., the latency for the phone to respond to user actions, such as

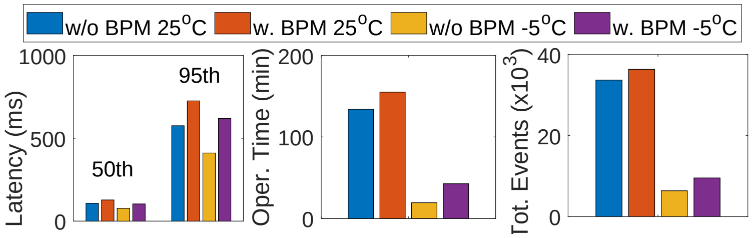


Figure 2.24: UI latency/operation time/total events.

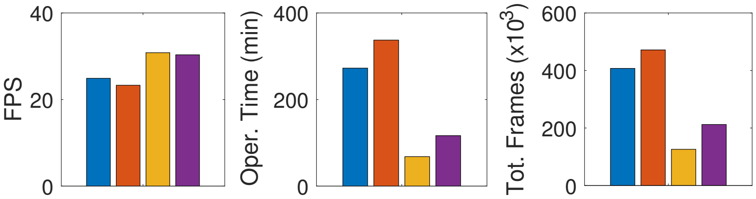


Figure 2.25: Video FPS/operation time/total frames.

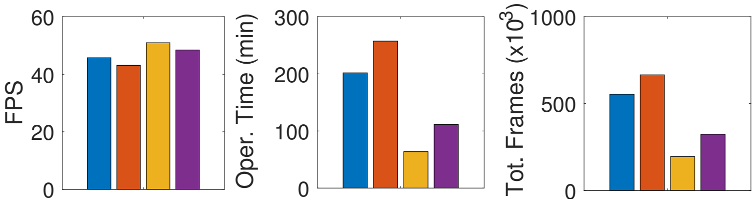
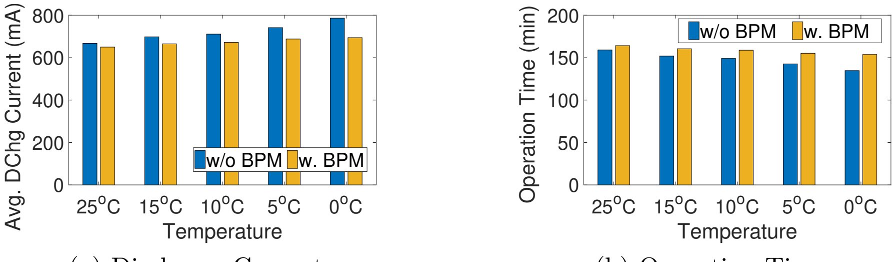


Figure 2.26: Game FPS/operation time/total frames.

touching the screen), and used *frames per second* (FPS) as the metric to evaluate user experience during video streaming and gaming. We repeated full discharging cycles 10 times while running each app at ambient temperatures of 25°C and -5°C, respectively.

Fig. 2.24 compares the {50th, 95th}-percentiles of the response latency, operation time, and total number of processed UI events, when running the UI/App exerciser on a fully charged Nexus 5X until it shut off. `BPM` increased the median latency from $108ms$ to $119ms$ at the 25^oC and from $77ms$ to $104ms$ at the -5^oC because of a lower processor frequency, but such an increase is only equal to approximately $11ms$ and $27ms$ per action, which are still below the average response time that human can perceive [98]. Moreover, `BPM`, by increasing the operation time by 1.15× at 25^oC and 2.2× at the -5^oC, enabled the device to perform 1.07× and 1.49× more user actions before the device shut off. For video streaming in Fig. 2.25, `BPM` slightly reduced the



(a) Discharge Current

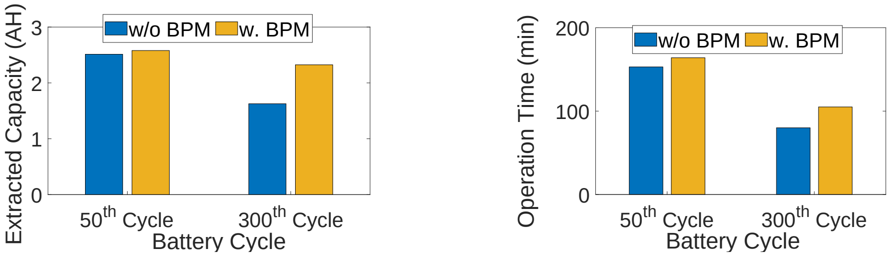
(b) Operation Time

Figure 2.27: Average discharge current and operation time with different temperatures.

FPS by 0.94× at 25^oC and 0.98× at -5^oC, but the device was able to stream 1.23× and 1.71× longer. As a result, the phone processed $64.3K$ and $86.2K$ more frames with BPM before it shut off, which are 1.16× and 1.68× more than DVFS. Similar observations were made with the gaming app shown in Fig. 2.26. Note that BPM's improvements to the operation time and total computation ability at 25^oC — an ideal temperature for battery operation — were not as significant as those at -5^oC, i.e., $\{1.15\times, 1.23\times,\ 1.27\times\}$ v.s. $\{2.2\times, 1.71\times,\ 1.74\times\}$ in terms of improving the phone's operation time, as shown in Figs. 2.24–2.26. This is because the phone's performance at 25°C was already close to optimal, and hence little space for further improvement existed, even without BPM.

2.6.4 BPM with Different Temperature and Battery Cycles

To obtain a clearer view on the performance of BPM in different runtime thermal scenarios, we ran the UI exerciser as the workload on a Nexus5X until it shut off, in different ambient temperatures ranging from room temperature (25°C) to freezing temperature (0°C). Fig. 2.27 summarizes the discharge current and operation time, averaged over 10 experimental runs and shows BPM to have extended the device operation by $(154 - 135)/135 = 14.1\%$ in an ambient temperature of 0°C compared with the case without BPM. Furthermore, the discharge current increased gradually as the temperature falls. This is because the battery's internal impedance increases



(a) Discharge Current

(b) Operation Time

Figure 2.28: Extracted capacity and operation time with batteries of different ages.

as temperature fell, which, in turn, reduced the battery's output voltage (i.e., V_b=$OCV-I\cdot R_b$). As a result, a larger discharge current was required to supply the same power (i.e., P_b=$V_b\cdot I_b$). Without BPM, the unregulated discharge current shortened the operation time, especially in cold ambient temperatures; the operation time at freezing temperature was shortened by $(159-135)/159 = 15.1\%$ compared with that at room temperature. By contrast, BPM's adaptive regulation of discharge current mitigated the shortening of operation time in the cold temperature; the operation time was reduced from 163min to 154min — only 6.1% — when the ambient temperature fell from 25°C to 0°C.

Last but not least, we compared BPM's effects on a Nexus 5X while powering the phone with two batteries of different ages (i.e., at the 50th and 300th discharging cycles, respectively) at room temperature (25°C). Fig. 2.28 plots the experiment results, demonstrating BPM's magnified advantages with aged batteries/devices — a 42.8% increased capacity delivery and a 26 min longer operation time for the battery at the 300th cycle.

2.7 Related Work

Battery Management of Mobile Devices. Sudden voltage drops and a crowd-sourced approach to the analysis of fading battery capacity have been presented in [35, 65–67]. Inaccurate SoC estimation due to battery temperature was addressed in [62], aiming to provide accurate SoC or full charge capacity monitoring. He et al. designed a method to estimate batteries state-of-health (SoH) using only the battery voltage [60]. These approaches, however, only passively monitor or estimate the battery status, and are not able to proactively operate mobile devices based on the thus-obtained awareness of batteries.

Power Management of Mobile Devices. At the other end of the spectrum, extensive research has been conducted to analyze the sources of energy consumption focusing on system [74, 126], application/network module [124], and user contexts [50,89], to prolong the operation of mobile devices. These analyses have led to various proposals for reducing the energy consumption in systems [40,113], apps [24,122] and networks [101,128]. However, these solutions do not consider batteries, and thus miss a crucial dimension for improving the device operation.

Battery-Aware Power Management. Among the limited explorations that have considered battery dynamics in power management, Benini et al. explored hardware-level power management policies in digital audio recorder system [19]. Another study proposed software approaches using task sequencing and DVFS [104] to optimize the operation time based on an offline battery model. Furthermore, a pulsed discharge pattern for communications in wireless sensor networks was proposed for enhancing the delivered battery capacity [32]. B-MODS [63] used battery-aware intermittent discharge patterns to exploit the battery relaxation effect for mobile data services. Unlike these approaches, BPM investigates the unexpected shutoffs of mobile devices by identifying the causes and designing/implementing mitigations.

2.8 Conclusion

We have presented BPM, a novel battery-aware power management middleware for mobile devices, to mitigate the unexpected device shutoffs frequently experienced by users. The design of BPM was steered by the causes of unexpected device shutoffs that we empirically identified/verified; that is, the dynamic voltage drop across the internal impedance of device batteries. Specifically, BPM regulates such voltage drops by (i) capturing the dynamically-changing battery impedance during device charging, and (ii) adaptively regulating the device's runtime discharge current. We have implemented and evaluated BPM on four commodity smartphones, demonstrating that it prevents unexpected device shutoffs and extends device operation by up to 2.03×.

CHAPTER III

eTEC: Efficient Thermoelectric Cooling

3.1 Introduction

Mobile devices such as smartphones, tablets, and laptops have become a primary computing/communication platform because of their portability and high computational power. In turn, their processor chips must cope with dangerously high temperatures with a limited cooling capability [110]. Because using a fan or heat sink is not a viable solution, mobile devices rely on *thermal throttling*, such as voltage/frequency scaling. When a given temperature threshold is reached, mobile devices are cooled by reducing the processor speed; thus, applications on those devices experience significant lagging. In particular, interactive mobile applications that require real-time responsiveness suffer significant degradation in performance.

Motivation. Thermoelectric coolers (TECs) are compact and controllable cooling devices that actively extract heat by flowing a cooling current via the *Peltier effect* [106]. A recent study suggested that thin-film TECs can meet the cooling demand of modern microprocessors [36]. Because TECs can be built at a micro scale (13mm^3 footprint), they can be embedded into mobile devices, whereas conventional cooling hardware cannot fit the device form factor. Furthermore, unlike cooling fans, solid-state TECs are reliable and noise-free, making them an attractive cooling solution for mobile devices.

Existing proposals for TECs focus on high-power desktop/server processors, optimizing *static* TEC cooling power in conjunction with fan cooling [45, 85]. However, low-power mobile processors dissipate substantially less heat than in previous analyses [46, 69], rendering TECs a feasible cooling solution without fans. Moreover, unlike server systems, mobile devices exhibit large variations not only in operating ambient temperature but also application workloads in response to sporadic user activities [103]. This chapter focuses on *dynamic* control of a TEC adaptive to various runtime thermal scenarios in mobile devices.

We present an efficient TEC (`eTEC`) solution embedded into mobile devices on the chip surface, which enables efficient processor thermal management. Our goal was to minimize performance degradation caused by thermal throttling through efficiently using the TEC. Our solution controls the TEC cooling power adaptively to the runtime workloads and ambient temperature.

To address this challenge, we first needed to model the thermal characteristics of the TEC and processor chip. Using the system thermal model, we then needed to determine the optimal cooling current and perform adaptive cooling control by tracking runtime workloads and ambient temperature. Chip temperature forms a convex function of the cooling current and processor speed, facilitating mathematical optimization for determining the optimal cooling current. At runtime, we read thermal sensors to learn a *processor activity factor* and adaptively controlled the cooling power.

We evaluated the effectiveness of the TEC cooling solution on a smartphone using representative mobile benchmarks [5]. When running compute-intensive workloads without the TEC, the processor speed was lowered to the minimum level, resulting in significant lagging. Using the TEC, the processor speed could be maintained close to the maximum speed; it only reduced by 1.8% on average compared with 19.2% without the TEC. Our TEC solution achieves the maximum performance at a cost of 0.2W cooling power consumption through adaptively controlling the TEC. Furthermore, we performed thermal simulations to complement the experimental evaluation for extreme ambient temperatures by emulating the TEC with thermal parameters identified from the experiments. The results revealed that significant performance degradation under adverse ambient temperatures can be mitigated using the TEC, but at the cost of higher cooling power consumption.

Contribution. The main contributions of `eTEC` are as follows:

- Feasibility test of thermoelectric cooling for mobile devices;
- Dynamic cooling control that is adaptive to runtime workloads and ambient temperatures; and
- Demonstration and an in-depth evaluation on a smartphone.

3.2 Related Work

To cope with increasing chip temperatures, new cooling techniques have received significant attention in recent years. Chowdhury *et al.* [36] demonstrated that modern thin-film TECs can meet the cooling requirement of on-chip hotspots. Chaparro *et al.* aimed to enhance existing dynamic thermal management using a TEC [29]. Long *et al.* [85] optimized the deployment of TEC devices and the static cooling current. Dousti *et al.* [45,46] optimized the cooling power of a fan cooler and TEC to minimize power consumption. A recent study [69] investigate energy harvesting and cooling with fan cooler on a fully instrumented TEC cooling system.

Unlike existing solutions, we aimed to adaptively control the TEC not only for runtime workloads but also for the surrounding ambient temperature, which is significant for mobile devices. Where previous studies have focused on desktop/server processors [45, 85] rated over 80W with fan coolers [46, 69], we propose the use of

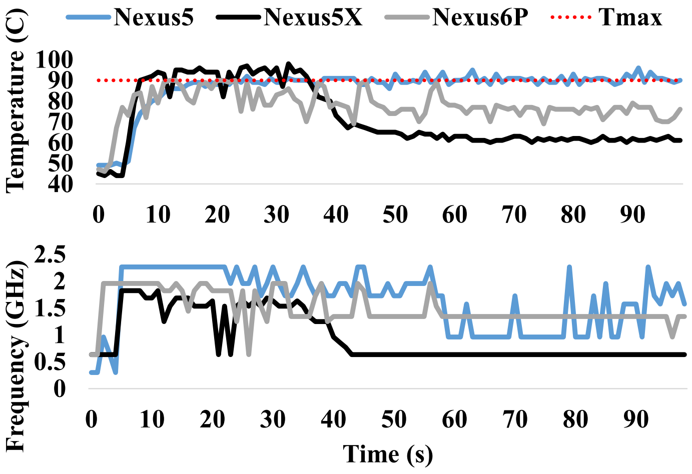


Figure 3.1: Chip temperature and frequency traces from Nexus 5/5X/6P while running Mibench benchmark.

TECs for mobile devices rated at most a power consumption of 3W, which can be efficiently cooled with low cooling power consumption.

3.3 Motivation

Modern mobile devices are powered by state-of-the-art multi-core chips, yet they cannot leverage their computing power because of limited cooling capabilities without fans or heat sinks. Our measurements for Nexus 5/5X/6P smartphones in Fig. 3.1 show that a CPU-intensive benchmark application [57] quickly raised the chip temperature beyond the specified temperature threshold within 20s, throttling the processor frequency thereafter. The processor frequency was reduced to 56/35/68% of the maximum speed on average; thus, applications on the devices suffer large performance degradation. This calls for a new portable cooling technology for mobile devices.

TECs can be used for mobile devices because of their scalable size and solid-state property. While a TEC can instantly extract heat from the cold side to the hot side, consuming active cooling power, the cooling effect is limited when the heat accumulates at the hot side. However, mobile devices only demand cooling power for

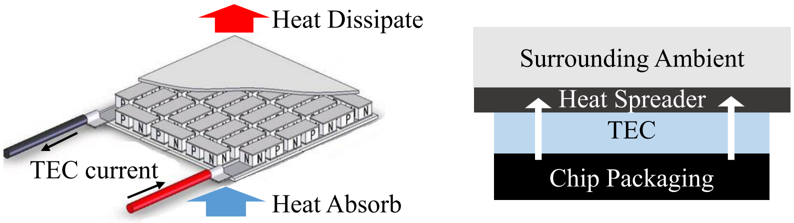


Figure 3.2: TEC device and chip packaging with embedded TEC.

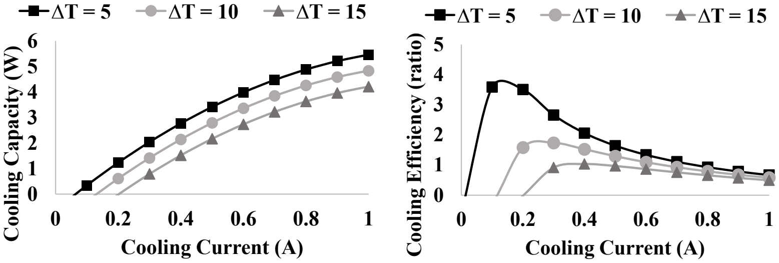
short bursts of user activities and are idle most of the time, and thus, the heat does not continuously accumulate on the hot side. Moreover, mobile processors are designed with orders of magnitude smaller thermal design power (TDP) [110] than desktops/server processors to operate without a fan or heat sink; therefore, the hot-side temperature does not increase much. The instantaneous cooling of TEC devices can efficiently cool their processor in response to sporadic workloads, and it must be dynamically controlled according to the runtime workloads to minimize cooling power consumption.

3.4 System Thermal Model

Fig. 3.2 illustrates a TEC device that can be embedded on top of the chip packaging. We consider a processor chip tiled with TEC modules extracting heat to the external ambient temperature. In this chapter, ambient temperature is the average temperature surrounding the chip packaging affected by heat dissipation from other components, such as the battery, display, and communication modules. This section describes how to model the TEC cooling capacity, processor power consumption, and system-level thermal behavior.

3.4.1 TEC Cooling Model

A TEC is a solid-state device made of arrays of N- and P-type semiconductor pellets. When an electric current flows through the thermoelectric material, heat is



(a) Net Cooling Capacity (b) Cooling efficiency

Figure 3.3: (a) Net cooling capacity and (b) cooling efficiency of a TEC depending on the temperature difference between two sides.

absorbed from the cold side and dissipated to the hot side via the *Peltier effect* [106]. This thermoelectric heat pump can be controlled by the current. The amount of cooling effect on the cold side and heat dissipation on the hot side can be modeled as follows [85]:

$$P_c = -ST_cI_{\text{TEC}} + \frac{1}{2}I_{\text{TEC}}^2 r_{\text{TEC}} \quad P_h = ST_hI_{\text{TEC}} + \frac{1}{2}I_{\text{TEC}}^2 r_{\text{TEC}} \tag{3.1}$$

where S is the *Seebeck* coefficient, I_{TEC} is the cooling current, T_c and T_h are the temperatures on the cold and hot sides, and r_{TEC} is the electrical resistance generating heat on both sides. The cooling power consumption is computed as follows [85]:

$$P_{TEC} = P_h - P_c = I_{\text{TEC}}^2 r_{\text{TEC}} + S(T_{\text{h}} - T_{\text{c}})I_{\text{TEC}}. \tag{3.2}$$

While active TEC cooling extracts heat from the cold side to the hot side, heat dissipates from the hot side to the cold side through heat conduction, thereby limiting the TEC cooling capacity, which is described as follows:

$$P_{\text{net}} = -ST_{\text{chip}}I_{\text{TEC}} + \frac{1}{2}I_{\text{TEC}}^2 r_{TEC} + \frac{T_h - T_c}{R_{\text{TEC}}} \tag{3.3}$$

Fig. 3.3 shows the cooling capacity and efficiency of a TEC for various TEC currents. As the temperature difference (ΔT) increases, more heat is conducted to the cold side, limiting the net cooling capacity. Because the TEC power consumption quadratically increases with the cooling current, the cooling efficiency ($\frac{P_{\text{net}}}{P_{\text{TEC}}}$) is higher with a smaller cooling current. Compared with high-power desktop/server chips, low-power mobile chips demand a small cooling current, where cooling efficiency of the TEC is maximized.

3.4.2 Processor Power Model

Processor heat dissipation can be modeled using dynamic and leakage power consumption [112]. The dynamic power consumed when executing workloads depends on the processor voltage/frequency and activity caused by workloads. Leakage power is statically consumed even when the processor is idle. Leakage power increases with temperature, which can be approximated using a linear model [25]. The processor power consumption is equal to:

$$P_{\text{chip}} = P_{\text{dyn}} + P_{\text{leak}} = CV^2 f\alpha + V(\beta_1 T_{\text{chip}} + \beta_0) \tag{3.4}$$

where C is the constant load capacitance, V, f are processor voltage/frequency, α is the processor activity factor caused by workloads, T_{chip} is the average chip die temperature and β_1, β_0 are leakage parameters. The leakage parameters, β_1, β_0, are platform-dependent constants depending on the technology that can be characterized at design time. By contrast, the activity factor α is a runtime parameter capturing the real-time CPU workloads that must be characterized at runtime.

3.4.3 System Thermal Model

Because the TEC cooling capacity and efficiency greatly depend on the temperature difference between the hot and cold sides of the TEC, as shown in Fig. 3.3, a system thermal model is required to efficiently control the TEC. By

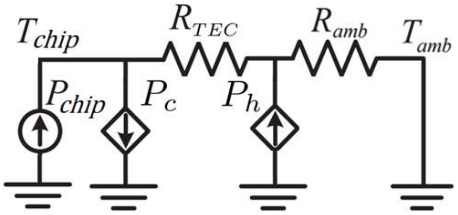


Figure 3.4: Thermal circuit model of the TEC system.

$T_{\text{chip}}, T_{\text{amb}}$	Chip and surrounding ambient temperatures
$P_{\text{chip}}, P_c, P_h$	Chip heat dissipation, TEC heat pump
$R_{\text{TEC}}, R_{\text{amb}}$	Thermal resistances between chip-TEC, TEC-ambient

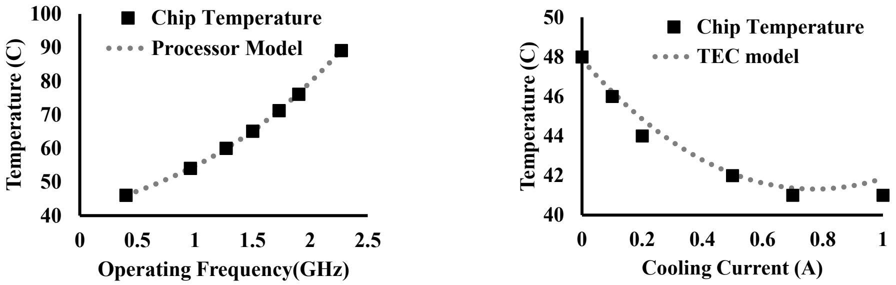
combining TEC and processor models, Fig. 3.4 presents an RC thermal circuit model that corresponds to the chip packaging with the TEC in Fig. 3.2. The TEC extracts heat from the cold side on the chip surface (P_c) to the hot side (P_h), which eventually dissipates into the ambient temperature. In the steady-state, chip temperature can be written as

$$T_{\text{chip}} = T_{\text{amb}} + R_{\text{amb}} \cdot P_h + (R_{\text{TEC}} + R_{\text{amb}})(P_{\text{chip}} - P_c) \tag{3.5}$$

Where thermal resistances and the TEC thermal constants can be determined at the design time through *system identification* [112], the changing ambient temperature T_{amb} must be captured at runtime.

3.5 Processor Thermal Management

Our focus in this chapter is on minimizing performance loss caused by thermal throttling through efficiently controlling the TEC cooling power. We first identified the model parameters required to optimize the TEC cooling current. Then, we dynamically controlled the cooling power according to runtime ambient and workloads.



(a) Varied processor frequency

(b) Varied cooling current

Figure 3.5: Thermal model identification with various (a) processor frequencies and (b) TEC currents.

3.5.1 Thermal Model Identification

To optimize the TEC, we first needed to identify the thermal model parameters of the target platform. By leveraging the thermal sensors available on most mobile devices [110], we could perform *system thermal identification* to learn the thermal model parameters. We ran mobile benchmark workloads [57] for a sufficient length of time and measured the steady-state chip temperature while the ambient temperature remained constant. The steady-state temperature measurement was repeated for the various processor frequency and TEC cooling currents. Fig. 3.5 illustrates the system thermal identification using various processor frequencies and cooling currents. Through plugging the measured chip temperature, processor frequency, and TEC cooling current into Eq. (3.5), we were able to identify the thermal constants and TEC parameters. Sec. **??** provides details on the experimental setups.

Fig. 3.5a presents the measured chip temperature and identified thermal model at different processor frequencies without TEC cooling. The chip temperature increases with processor frequency following the dynamic power model in Eq. (3.4). The measured chip temperature (plotted as square) accurately fits in the processor thermal model (plotted as dashed line) using the identified thermal parameters. Fig. 3.5b plots the steady-state chip temperature at the different TEC cooling currents while the processor is idle. The chip temperature cools with increasing cooling currents, but

Joule heat dissipation grows quadratically, thereby limiting the net cooling capacity according to Eq. (3.3). The TEC thermal parameters can be identified from these measurements, and the chip temperature (plotted as square) can be approximated well using the TEC thermal model (plotted using dashed line).

The leakage parameters were also identified by placing the device under the various ambient temperatures while the processor was idle. The measured leakage power could be approximated well using a linear fitting from 45 °C to 85 °C of on-chip temperature. Table. 3.1 summarizes the identified power and thermal model parameters.

Table 3.1: Identified leakage and thermal parameters.

Leakage Power		Thermal Resistance		TEC parameters	
β_1	β_0	R_{TEC}	R_{amb}	S	r_{TEC}
0.016	0.035	3.4W/K	17.4 W/K	0.022 V/K	0.5 Ω

3.5.2 TEC Optimization

For a given processor activity and ambient temperature, we could use the proposed thermal model to find the optimal cooling current. Our goal was to minimize thermal throttling while meeting the chip temperature and power constraints, which can be formulated as follows:

Given runtime processor activity factor α and ambient temperature T_{amb},

Find the maximum allowed operating frequency f and the corresponding TEC cooling current I_{TEC} that meet the power and thermal constraints $P_{\max}, T_{\max}$.

$$\max_{I_{\text{TEC}}} \quad f \tag{3.6}$$

$$\textbf{s.t } T_{\text{chip}} \leq T_{\max} \tag{3.7}$$

$$P_{\text{total}} \leq P_{\max} \tag{3.8}$$

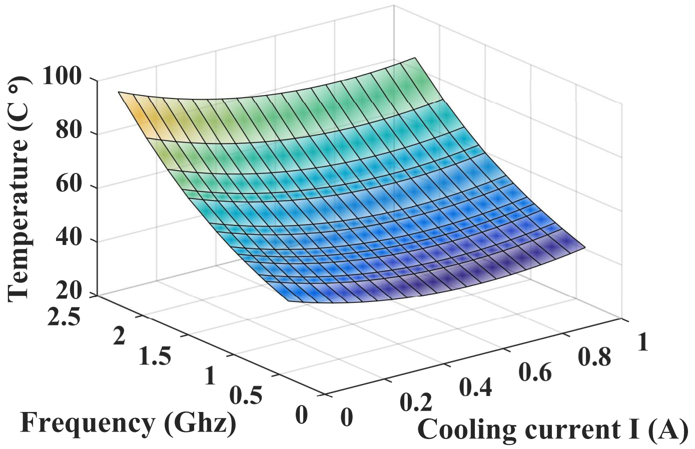


Figure 3.6: Convexity of chip temperatures for the various processor frequencies and TEC cooling currents.

Eq. (3.6) defines our objective to find the maximum allowed processor frequency subject to power and thermal constraints. Since interactive mobile applications are usually latency-sensitive, their performance is directly impacted by processor speed. We aimed to find the maximum allowed processor speed for the real-time responsiveness of mobile applications.

Eqs. (3.7) and (3.8) specify the thermal and power constraints of the platform. To guarantee maximum temperature for long-running workloads, the steady-state temperature must be lower than the temperature threshold. Furthermore, the total power consumption of the chip and cooling device must be lower than the power constraint.

Eq. (3.5) describes the steady-state chip temperature as a convex function of the processor frequency f and cooling current I. Fig. 3.6 illustrates the chip temperature for different processor frequencies and cooling currents using the identified model parameters. Because of the optimization problem's convexity, it can be efficiently solved using an interior point method with a complexity of $O(n^{3.5})$ for a typical case [72]. To avoid runtime overheads, we optimized the TEC in terms of design time for a range of chip and ambient temperatures, and online thermal control use it

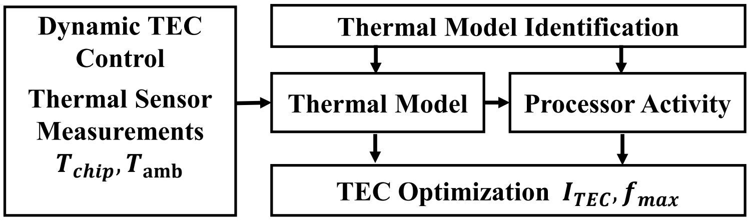

Dynamic TEC Control
Thermal Sensor Measurements
T_{chip}, T_{amb}
Thermal Model Identification
Thermal Model
Processor Activity
TEC Optimization I_{TEC}, f_{max}

Figure 3.7: Workflow diagram of dynamic TEC control.

through table lookup.

3.5.3 Dynamic TEC Control

In mobile systems, not only runtime workloads but also ambient temperatures dynamically change over time. Therefore, for efficient thermoelectric cooling, the cooling power must be adapted to the runtime CPU load and operating ambient temperature. Modern mobile devices are equipped with on and off-chip thermal sensors that can be used to learn the runtime chip and surrounding ambient temperatures.

Fig. 3.7 shows how we learned runtime CPU workload and operating ambient using thermal sensor measurements. We measured both the on and off-chip thermal sensors ($T_{\text{chip}}, T_{\text{amb}}$) and plugged into Eq. (3.5) to calculate the processor power (P_{chip}). From the processor power model in Eq. (3.4), we could learn the processor activity factor (α) caused by the application workloads running on the processor. Using the thus-obtained runtime thermal parameters (α, T_{amb}), we could optimize the TEC cooling current and processor frequency (I_{TEC}, f) to meet the thermal and power constraints.

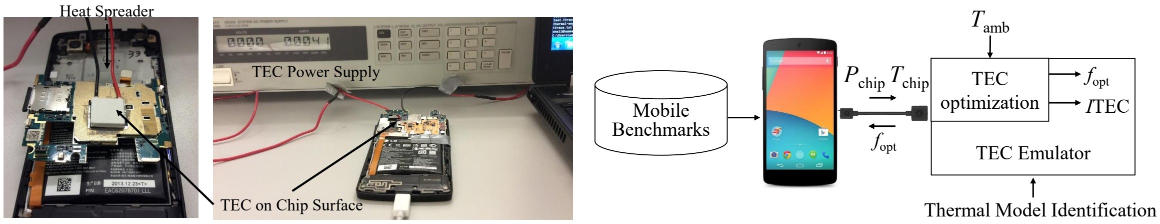
This dynamic thermal management was periodically invoked to adapt to various runtime thermal scenarios. The period must be short enough to quickly react to application workloads that may overheat the chip. The chip temperature does not increase instantly because of the thermal time constant, and it required several seconds to reach the threshold, as shown in Fig. 3.1. Thus, the invocation period could be orders of seconds such that the runtime overhead is low.

While our goal was to maximize performance, the CPU frequency could be lowered to save power when the CPU load was low. To allow the DVFS governor to save power, we only set the maximum allowed frequency corresponding to the TEC cooling. Thus, the underlying DVFS governor could adjust the frequency depending on the runtime workload within the allowed range through TEC cooling.

3.6 Evaluation

We have experimented with our TEC solution on a smartphone, as well as performed extensive simulations by emulating the TEC. Thermal simulations complemented the experiments allowing us to examine the extreme ambient and various workloads.

Experimental Setup. Our experiment was on a Nexus 5 powered by a quad-core Snapdragon 800 processor, which supports chip-wide DVFS with a maximum frequency of 2.26GHz. The device is equipped with on and off-chip thermal sensors with 1 °C precision. We used the on-chip thermal sensors to obtain the average chip die temperature, whereas we used the off-chip thermal sensors for the surrounding ambient temperature. For the TEC, we used CP60133 [7] with silver-based thermal paste (Arctic Silver 5) rated with a maximum cooling capacity of 12.2W at a maximum cooling current of 6.0A. The TEC was powered by HP 6632A programmable power supply. Fig. 3.8 demonstrates our experimental setup and overview. Because the processor chip faced the front panel, we placed the TEC on the chip surface and reinstalled it to the original configuration. Thus, the hot side of the TEC was faced the heat spreader on the front panel, and the back cover was also reinstalled in the experiment. We used the performance governor for DVFS and default thermal governor in Android kernel such that frequency was only reduced when a specified temperature threshold is reached. As representative



(a) Setup

(b) Overview

Figure 3.8: Experiment and simulation setup.

mobile workloads, we used Antutu benchmark suite v6.2 [5], which comprises 3D graphics and games as well as multi-thread CPU operations. We also tested Mibench benchmarks [57] with three different categories (computational, network, and communication) representing typical operations in mobile systems.

In addition, we performed simulations by emulating the TEC to examine the proposed solution for various workloads under different operating ambient temperatures. As shown in Fig. 3.8, we obtained power and temperature traces from the devices while running mobile applications, and emulated the TEC using thermal parameters identified from the experiments. The simulation was performed in MATLAB similar to the HotSpot thermal simulator [45].

Throughout the evaluation, we compared the proposed solution with the baseline system without the TEC. We evaluated the temperature control, CPU performance, and power consumption of the TEC.

Experimental Results. In the experiments, we focused on dynamic thermal control and performance gain using the TEC. We ran Antutu benchmark suite [5], which ran 3D graphics, 3D gaming and CPU-intensive workloads for 3 minutes. Fig. 3.9 shows the real-time traces of the chip temperature, processor frequency and TEC power consumption. While processing 3D graphics, the baseline without the TEC quickly reached the temperature threshold and throttled the processor frequency from 30s to 75s. The processor frequency was largely reduced to the minimum level, 0.3GHz, where graphics were significantly lagging. The overall processor frequency when

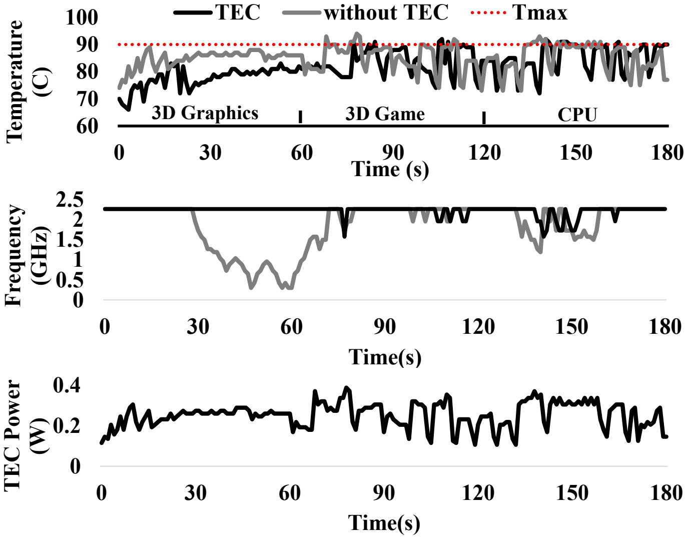
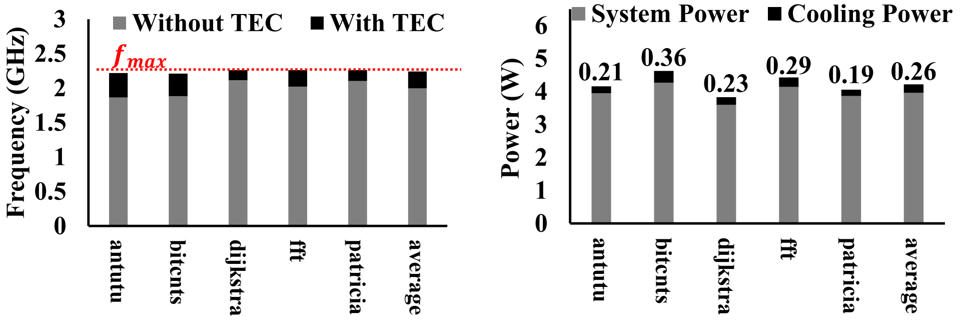


Figure 3.9: Temperature, frequency, and cooling current traces from the experiment running the Antutu mobile benchmark.

running the benchmark suite was 1.87GHz, which translated to a 19.2% performance loss with respect to the maximum processor speed.

The proposed TEC solution could maintain the processor temperature lower than the temperature threshold while maintaining the processor frequency close to its maximum level. In particular, the large performance loss from 30s to 75s could be mitigated using the TEC. The minimum processor frequency was 1.6GHz using the TEC compared with 0.3GHz without the TEC; thus, the TEC resulted in a more reliable latency for user applications. When the processor heat dissipation still exceeded the cooling capacity, the processor frequency was briefly lowered at around 150s. The average processor frequency was 2.22GHz, which translated to only a 1.8% performance loss, an improvement of 18.9% over baseline without the TEC. Our TEC solution also reduced thermal violation to 1.1% from 3.3% without the TEC. It did so with dynamic TEC control consuming an average cooling power of 0.21W, which is comparable to a Wifi module's power consumption [44]. Without dynamic cooling control, the TEC must maintain the worst-case cooling power of 0.39W for peak workloads, consuming 86% more cooling power. The results demonstrated the



(a) Processor Frequency

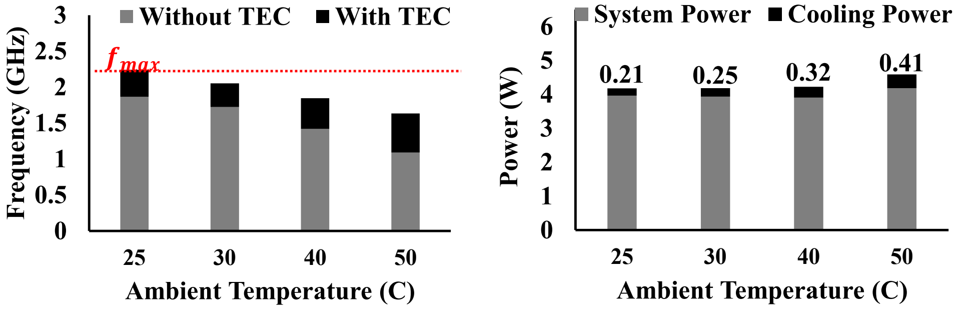
(b) Power Consumption

Figure 3.10: (a) Processor frequency and (b) power consumption with and without the TEC for benchmark applications.

feasibility of the TEC for mobile devices and efficient dynamic cooling control.

Simulation Results. In the simulations, we focused on a trade-off between performance versus cooling power consumption across different benchmarks and ambient temperatures. Fig. 3.10 shows the processor frequency and total power consumption under room temperature with and without the TEC; the TEC cooling power consumption is presented on the top of the bar. Using the TEC, all the applications could run close to the maximum processor frequency of 2.24GHz on average, which translated to a performance loss of 0.7%. The average cooling power consumption was 0.26W, which corresponded to a system power consumption of 6%. The performance improvement was especially significant for compute-intensive workloads because they suffer a larger performance degradation from thermal throttling; for example, 23.9% for *bitcnts* compared with 7.5% for *patricia*. As a consequence, compute-intensive *bitcnts* demanded more cooling power at 0.36W compared with 0.19W for *patricia*. Thus, the TEC system enables peak performance operations in mobile platforms, which is limited by cooling capacity; however, this peak performance comes at the cost of TEC power consumption.

In addition, we simulated different ambient temperatures for running Antutu benchmarks in Fig. 3.11. At the high ambient temperature, the baseline without the TEC suffered a large performance loss from thermal throttling. Without the TEC,



(a) Processor Frequency

(b) Power Consumption

Figure 3.11: (a) Processor frequency and (b) power consumption with and without the TEC for different ambient temperature.

the average frequency in case of 40°C (50 °C) ambient temperature was reduced to 1.42GHz (1.09GHz), which translated to a 37.1% (51.7%) performance loss. Using the TEC, the average frequency could be maintained to 1.84GHz (1.63GHz), which translated to an 18.5% (27.8%) performance loss for 40°C (50 °C). However, the cooling power consumption also increased with a higher ambient temperature of 0.41W for 50 °C. The results showed that the performance degradation under the high ambient temperature could be largely be mitigated, which is especially significant for mobile devices that experience large variations in operating ambient temperature.

3.7 Conclusion

In this chapter, we presented a thermoelectric cooling solution for mobile devices. In particular, dynamic TEC control was proposed for the efficient thermal management of processors in mobile devices. Because mobile systems face large variations in runtime workloads and ambient temperatures, our solution adaptively controls TECs using online information. Our evaluation on a smartphone demonstrated the solution's effectiveness at maintaining peak performance, which is especially critical for interactive mobile apps. Our experimentation with realistic mobile workloads showed a performance loss of only 1.8% with a TEC compared with a 19.2% loss without a TEC at a cost of 0.2W cooling power consumption.

CHAPTER IV

RT-TRM: Real-time Thermal-Aware Resource Management

4.1 Introduction

Thermal-aware resource management has become critical for modern embedded real-time systems, such as automotive controls and smartphones, as they are increasingly realized on powerful computing platforms with exponentially increasing power densities. High on-chip temperatures shorten a platform's lifetime and severely degrade its performance and reliability, risking safety (e.g., vehicle breaks or smartphone explosions). Therefore, the processor temperature must be maintained below the peak temperature constraints while all application timing constraints are satisfied.

There are two key thermal issues to consider for embedded real-time systems: (1) dynamically *varying ambient temperature* and (2) task-level *power dissipation.* Our experimental evaluations have shown that the ambient temperature of an automotive electronic module varies highly and dynamically even during a single driving event, and furthermore, its seasonal temperature varies widely up to a difference of 28°C. Moreover, according to our evaluation of various automotive benchmark applications, the average power consumed by each application differs by up to 140% (to be detailed in §4.3).

Motivation. These dynamic thermal features pose significant challenges to meeting application timing constraints. In particular, the maximum available computation power varies with the ambient temperature, because the temperature of a processor depends on its ambient. According to our experimental evaluation (§4.3), an increase of 14.9°C in ambient temperature results in a maximum reduction of 28.8% in the processor's computation power. In such a case, the processor's thermal constraint may be violated if it executes a task whose average power dissipation exceeds a certain limit. One may reduce the processor temperature by idling or slowing it, but such an action may also lead to task/job deadline miss(es), thus violating the app timing constraint. This calls for adaptive resource management that considers dynamically varying ambient temperatures and task-level power dissipation to meet both the processor's thermal and app's timing constraints.

A significant amount of work has been conducted on real-time thermal-aware scheduling (see [76] for a survey). Existing approaches have usually employed DVFS scheduling [31, 51], idle-time scheduling [77], or task scheduling [26] to minimize the peak temperature while guaranteeing the timing constraint. Worst-case temperature analyses [78, 109] have also been proposed for offline guarantees to meet thermal constraints. The concept of thermal utilization [12] was introduced to capture the thermal impact of periodic real-time tasks on processors. Most of these existing solutions, however, assume either *fixed* ambient temperature or *constant, task-independent* power dissipation. Although there exist real-time feedback thermal controllers that minimize the error between the current processor temperature and the desired temperature by regulating task utilization [54] or frequency [55], they are limited to guaranteeing thermal constraints due to temperature overshooting.

In this chapter, we propose a new real-time thermal-aware resource management framework, called RT-TRM, which captures not only varying computation power bounds caused by variations in ambient temperature but also different power demands by different tasks. RT-TRM adaptively makes a parameter assignment (voltage/frequency levels and a task period assignment) and builds a schedule (processor idling or task-execution as well as the priority ordering of tasks) to meet both thermal and timing constraints.

We first proposed a *task-level* dynamic power model that uses a simple task-level parameter called the *activity factor*, to capture different power dissipations by different tasks. Building on this dynamic power model, we studied the effect of task-execution or processor-idling on the processor temperature. Our model was also validated experimentally with several automotive benchmarks running in various realistic environments. Second, we proposed the notions of *dynamic power demand* of a task set and *dynamic power bound* at a given ambient temperature and derive the feasibility conditions for a parameter assignment with respect to both thermal and timing constraints. We then developed a runtime adaptive strategy that can preserve feasibility under ambient temperature variations by adjusting the parameter assignment. Third, building on a feasible parameter assignment, we developed an online scheduling policy that determines not only the processor state (active or idle) but also the order of executing tasks in the active state. Our scheduling algorithm could reclaim slack (spare capacity) at runtime and allocate it to tasks *in proportion to* their power demands by considering the fact that a task with higher power dissipation should be assigned more idle time. This way, we could meet both thermal and timing constraints with much fewer preemptions. Finally, we implemented RT-TRM on an automotive microcontroller to demonstrate its effectiveness for dealing with ambient temperature variations. RT-TRM was shown to improve resource utilization by 18.2% over the existing runtime feedback thermal controllers while guaranteeing both thermal and timing constraints.

Contribution. This chapter makes the following main contributions:

- Demonstration of the importance of accounting for dynamic ambient temperature and task-level power dissipation for thermal-aware resource management (§4.3);

- Development of a dynamic power model that captures different power dissipations with a simple task-level parameter called the *activity factor* and its experimental validation (§4.4);
- Development of an adaptive parameter-assignment framework under varying ambient temperature while preserving feasibility (§4.5); and
- Development of an online idle-time scheduling algorithm that enables dynamic idle-time allocation with much fewer preemptions while guaranteeing both thermal and timing constraints (§4.6);
- Implementation and evaluation of the effectiveness of RT-TRM on an automotive microcontroller (§4.7).

4.2 Related Work

Significant work has been conducted on thermal management at both hardware (e.g., architecture design and floorplans) and OS level (e.g., thermal-aware DVFS and scheduling) [76]. The focus of this chapter is on OS-level thermal-aware resource management for hard real-time systems, such as cars.

Thermal-aware real-time scheduling has been an active subject of research that attempts to meet timing and thermal constraints in a *constant* environment. DVFS scheduling determines the voltage and frequency of a processor to minimize the power consumption [15, 129] and peak temperature subject to timing constraints on single-core [30, 31, 119] or multi-core platforms [51]. Multi-core task scheduling [26] determines task-to-core assignment and scheduling to minimize the peak temperature. Thermal shaping inserts idle periods during task execution to reduce the temperature without missing deadlines [18, 77].

Researchers have focused on different task-level power dissipations to reduce the peak temperature [13, 70] or maximize throughput [68, 127] by interleaving the execution of hot and cold tasks. By analyzing such task-level power variations, the peak temperature was derived to meet the thermal constraint [78]. The concept of thermal utilization was introduced in [11, 12] to capture the different thermal impacts of real-time tasks. Several researchers have considered adaptive thermal-aware resource management for real-time tasks to cope with dynamic environments; feedback controllers regulate the processor temperature by adjusting processor utilization [54, 87] or operating frequency [55] subject to the timing constraint.

Although researchers have developed task-level scheduling and processor-level thermal control techniques to deal with both thermal and timing requirements, they have not yet addressed both large environmental variations and peak temperatures caused by task workloads together. To meet this need, we first verified the significance of these factors in automotive systems, and then developed and validated a task-level thermal model that could capture individual tasks' different power dissipations. Building upon the task-level thermal model, we proposed a new thermal-aware resource management scheme that (i) jointly adapts task periods and processor frequency in response to the varying ambient temperature and (ii) schedules tasks to meet both thermal and timing constraints.

4.3 Target System, Challenges, and Solution Overview

This section describes our target system (§4.3.1) and introduces the challenges faced therein (§4.3.2) followed by an overview of our approach (§4.3.3).

4.3.1 Target System

We consider an embedded real-time system running a set of real-time tasks on a computing platform.

Processor and Task Model. Our target system is a uniprocessor platform that provides DVFS with a separate set of discrete frequency/voltage levels. If an operating frequency f is adjustable within the specified range of $[f_{min},\ f_{max}]$, its corresponding voltage V is determined according to a typical implementation principle [52]. Furthermore, we consider a task set τ composed of implicit-deadline periodic tasks. Each task $\tau_i \in \tau$ is characterized by period p_i and its worst-case execution time (WCET) $e_i(f)$ as a function of operating frequency f. We assume that p_i is adjustable within $[p_i^{min},\ p_i^{max}]$ based on typical application elasticity [39, 111]. Note that, for a task whose period is fixed, we set its period range as $p_i^{min} = p_i = p_i^{max}$. Such τ_i is assumed to generate a sequence of jobs, once every p_i time-units, with each job needing to complete $e_i(f)$ within a relative deadline of p_i time-units.

Power and Thermal Model. We consider dynamic power management where the processor is in either *idle* or *active* state. The processor is said to be active if it is currently executing a job, or idle otherwise. Its power dissipation (P_{proc}) is then expressed as $P_{proc} = P_{leak} + P_{dyn}$, where P_{leak} is the leakage power for the processor to stay ready (in active or idle state) for the execution of jobs, and P_{dyn} is the additional dynamic power to execute a job (in active state). The term P_{leak} is modeled as [83]: $P_{leak} = V \cdot (\beta_1 \cdot T + \beta_0)$, where β_1 and β_0 are processor-dependent constants, and T is the processor's temperature. Note that P_{dyn} depends on the task currently running on the processor, and its detailed model is described in §4.4.1.

To translate the processor's power dissipation to its temperature, we use a well-known thermal circuit model [20]. If the average processor power and ambient temperature is $P_{proc}(t)$ and $T_{amb}(t)$, respectively, over a time period t, then the processor temperature $T(t)$ at the end of this period is

$$T(t) = T(0) \cdot e^{-\frac{t}{R \cdot C}} + (T_{amb}(t) + P_{proc}(t) \cdot R) \cdot (1 - e^{-\frac{t}{R \cdot C}}), \tag{4.1}$$

where R and C are the thermal resistance and capacitance, respectively, and $T(0)$ is the initial temperature of the processor. Eq. (4.1) shows that the temperature will increase/decrease towards and eventually reach $T_{amb}(t) + P_{proc}(t) \cdot R$. We define the steady temperature $T(\infty)$ of the processor as

$$T(\infty) = T_{amb}(t) + P_{proc}(t) \cdot R. \tag{4.2}$$

4.3.2 Problem Statement and Motivation

Problem Definition. We want to address the following real-time thermal-aware resource management problem.

Definition 4.1. Given a task set τ running on a uniprocessor, determine (i) the voltage/frequency (V/f) level, (ii) the period $\{p_i\}$ of task $\tau_i \in \tau$ parameters, and (iii) the schedule of jobs such that (a) temperature $T(t)$ never exceeds the peak temperature T_{max} (thermal constraint), and (b) all jobs of $\tau_i \in \tau$ meet their deadlines for all possible legitimate job arrival sequences (timing constraint).

To generate a job schedule, we need to determine not only the processor state (active or idle) but also the order of executing jobs in active state. From real embedded systems (e.g., cars and smartphones), we found two key thermal characteristics: (1) dynamic changes in the ambient temperature and (2) different power dissipations by different tasks. These are the primary motivation behind RT-TRM.

Dynamic Changes on Ambient Temperature. Unlike desktops or data-centers, embedded real-time systems experience a wide range of environmental variations (especially in ambient temperature) during their operation/life. To confirm this, we measured the ambient temperature of a vehicle infotainment module embedded in the dashboard over days and months, and the results are plotted in Fig. 4.1. When the car was driven for several days, the change in the ambient temperature was highly

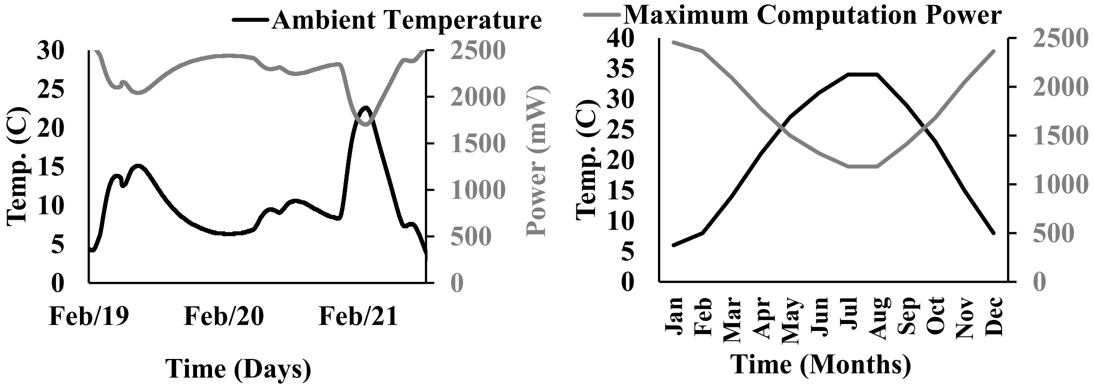


Figure 4.1: Ambient temperature variations over time and the corresponding available computation power.

dynamic and fluctuated between 0°C and 23°C. During a single driving event on Feb. 21, 2018 the ambient temperature increased by up to 180%. The seasonal variation in the ambient temperature is also very wide. A similar phenomenon was also reported in [71] for car engine and transmission control units.

Under such a varying ambient temperature, real-time thermal-aware resource management becomes much more challenging as the processor's temperature is affected by the ambient temperature. We can use Eq. (4.2) to calculate the maximum processor's power dissipation without exceeding T_{max} for a given ambient temperature T_{amb} as $\frac{T_{max}-T_{amb}}{R}$. The change in the maximum processor computation power under a varying ambient temperature is then plotted in Fig. 4.1 (the gray line).[1] For example, on Feb. 21, 2018 as the ambient temperature increased by 14.9°C from 8.3°C, the available processor computation power decreased by 28.8%.

Task-level Power Dissipations. We also measured the processor's average power consumption to run various automotive benchmarks [57]. The results are plotted in Fig. 4.2, where each app is shown to consume a different amount of power. For example, a table lookup task consumes 1726mW, whereas a bit manipulation task consumes 2348mW at the maximum processor frequency.[2]

[1] We set T_{max} = 60°C and R = 22°C/W based on the specification of an automotive microcontroller [53].

[2] A table lookup operation is used by an engine control module to find an output value

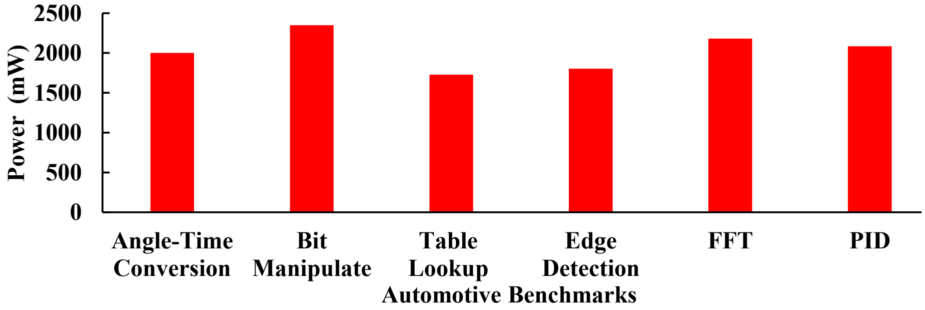


Figure 4.2: Average power consumptions for various automotive applications.

In summary, the available processor's computation power varies with the ambient temperature. In addition, the execution of each task imposes a different power demand on the processor. Therefore, to meet both thermal and timing requirements, we must consider different task-level power dissipations and make adaptive parameter assignments and job schedules according to the varying ambient temperature.

4.3.3 Overview of the Proposed Approach

To solve the real-time thermal-aware resource management problem while considering the varying ambient temperature and diverse task-level power dissipations, we address the following questions:

Q1. How to model power dissipations of different tasks and analyze the impact of their execution on the thermal behavior?

Q2. How to make adaptive parameter assignments under dynamically changing ambient temperatures while meeting both thermal and timing constraints?

Q3. How to derive a job schedule meeting both thermal and timing constraints based on parameter assignment?

corresponding to an input value (e.g., the ignition angle). A bit-manipulation operation is used by a display module where the pixels are moved into a display buffer until the entire buffer is displayed.

To answer Q1, we develop a *task-level* dynamic power model using a simple task-level parameter called the *activity factor*. Based on this task-level dynamic power model, we analyze the effect of task execution on the processor's temperature; that is, whether it increases or decreases the processor's temperature. Moreover, we empirically determine the activity factors for several automotive apps and verify our model in various environments, specifically, under different processor-frequency/task-utilization settings, varying ambient temperatures, and executing multiple tasks (§4.4).

To address Q2, we define a *dynamic power demand* of a task set τ that represents the total dynamic power demand by τ at the processor's steady temperature. We also define a *dynamic power bound* function of T_{amb} that represents the processor's maximum dynamic power P_{dyn} at T_{amb} without violating the thermal constraint. Based on these concepts, we derive the feasibility conditions of a task set and formulate an optimization problem that finds a feasible parameter assignment for a given T_{amb}. We also develop a runtime adaptive strategy that can preserve feasibility by adapting the parameter assignment to ambient temperature changes. We determine a tolerable ambient temperature range for parameter adaptation by considering the trade-off between adaptation overhead and resource efficiency. With our adaptive parameter assignment, the steady temperature is guaranteed not to exceed the peak temperature limit without missing any deadlines in the presence of ambient temperature variations (§4.5).

To answer Q3, building on the feasible parameter assignment derived by answering Q2, we develop an online scheduling policy. A task schedule may affect the *transient temperature*, potentially violating the thermal constraint before reaching the steady temperature. To avoid this, we calculate the *minimum idle-time* required for the execution of each job with respect to the thermal constraint. We then develop an idle-time scheduling algorithm that can reclaim unused resources at runtime and utilize them to allocate idle-time efficiently while meeting all deadlines with the minimum idle-time for each task. As a result, our algorithm can guarantee both thermal and timing constraints with much fewer preemptions (§4.6).

4.4 Task-Level Power Model

We present a task-level power-consumption model that captures different dynamic power dissipations by individual tasks. In particular, we use a simple task-level activity factor to capture each task's dynamic power dissipation (§4.4.1) and empirically validate the model using an automotive platform and workloads (§4.4.2).

4.4.1 Task-Level Dynamic Power Model

For automotive workloads, power dissipation is found to vary significantly with the executing task (Fig. 4.2). Since individual tasks programmed with distinct sets of instructions generate different switching activities and dynamic power dissipations, we used a task-level activity factor α_i to capture such different dynamic power P_i consumed by each task τ_i as $P_i = V^2 \cdot f \cdot \alpha_i$. Using this task-level dynamic power model, we can analyze how the processor's temperature changes with the execution of each job/task. Let $T(t)$ ($T_i(t + e_i(f))$) be the temperature at the beginning (end) of the execution of a job of τ_i. Using Eqs. (4.1) and (4.2), $T_i(t+e_i(f))$ can be written as:

$$T_i(t + e_i(f)) = T(t) \cdot e^{-\frac{e_i(f)}{R \cdot C}} + T_i^{\infty}(T_{amb}) \cdot (1 - e^{-\frac{e_i(f)}{R \cdot C}}), \tag{4.3}$$

where $T_i^{\infty}(T_{amb})$ is the steady temperature associated with the execution of τ_i that would be reached if the processor executes τ_i continuously. $T_i^{\infty}(T_{amb})$ can be expressed as:

$$T_i^{\infty}(T_{amb}) = T_{amb} + (P_i + P_{leak}) \cdot R. \tag{4.4}$$

We observe from Eqs. (4.3) and (4.4) that (i) if $T(t) < T_i^{\infty}(T_{amb})$ then the temperature increases towards $T_i^{\infty}(T_{amb})$, and (ii) if $T(t) \geq T_i^{\infty}(T_{amb})$ then the temperature decreases towards $T_i^{\infty}(T_{amb})$. A task τ_i is said to be *hot* if $T_i^{\infty}(T_{amb}) > T_{max}$, or *cold* otherwise. Depending on T_{amb}, τ_i can become hot or cold.

To consider the effect of idling the processor on its temperature, we let $T_0(t+l)$ denote the temperature at the end of an idle period of length l. Similar to Eq. (4.3), $T_0(t+l)$ can be written as:

$$T_0(t+l) = T(t) \cdot e^{-\frac{l}{R \cdot C}} + T_0^{\infty}(T_{amb}) \cdot (1 - e^{-\frac{l}{R \cdot C}}), \tag{4.5}$$

where $T_0^{\infty}(T_{amb}) = T_{amb} + P_{leak} \cdot R$ is the processor's steady temperature in an idle state.

Thus far, we have discussed the thermal effect of continuously executing (idling) a single job (a processor). Now, let us consider the impact of a schedule of periodic tasks and idle-times. Let $T(t, W(t))$ denote the temperature at the end of a schedule $W(t) = \{w_i(t)\}$, where $w_i(t)$ is the total workload scheduled in $(0, t]$. Then, $T(t, W(t))$ can be written as:

$$\begin{aligned} T(t, W(t)) &= T(0) \cdot e^{-\frac{t}{R \cdot C}} \\ &\quad + (T_{amb} + (\sum_{\tau_i} P_i \cdot \frac{w_i(t)}{t} + P_{leak}) \cdot R) \cdot (1 - e^{-\frac{t}{R \cdot C}}), \end{aligned} \tag{4.6}$$

where $\sum_{\tau_i} P_i \cdot \frac{w_i(t)}{t}$ is the average dynamic power consumed by $W(t)$. Note that every task τ_i generates a sequence of jobs executing $e_i(f)$ every p_i time-units, consuming an average dynamic power of $P_i \cdot \frac{e_i(f)}{p_i}$. We can then define the steady temperature $T(\infty, \tau)$ of a task set τ as:

$$T(\infty, \tau) = T_{amb} + (\sum_{\tau_i} P_i \cdot \frac{e_i(f)}{p_i} + P_{leak}) \cdot R. \tag{4.7}$$

Table 4.1: Identifying activity factors and maximum errors.

Task	Angle	Bit	Table	Edge	FFT	PID
T_i^∞ (°C)	66.1	73.6	59.9	61.6	69.5	67.8
α_i	0.355	0.446	0.284	0.304	0.435	0.377

Note that the steady temperature of a task set is independent of its schedule, which serves as a basis for the feasibility condition presented in §4.5.

Identifying Task-level Activity Factors. To identify the activity factor of each task, we ran automotive benchmarks, one at a time, with 100% resource utilization at the maximum frequency under room temperature. We then measured the steady temperature and determined each task's activity factor using Eq. (4.4), which are presented in Table 4.1.[3] The activity factor varies greatly with tasks by up to 65%. For example, a table-lookup task with a large number of conditional switches and I/O accesses shows a low activity factor, whereas a bit-manipulation task with a high instruction-per-cycle rate shows a high activity factor.[4]

4.4.2 Empirical Model Validation

To confirm that the task-level dynamic power model and its thermal effect represent real hardware behaviors, we measured the steady temperature of the processor and compared it with our model's estimation under various settings. In particular, we validated our model under (i) different processor-frequency/task-utilization settings, (ii) running multiple tasks together, and (iii) different ambient temperatures.

First, as shown in Fig. 4.3, we varied the task period to achieve the processor utilization ranging from 10% to 90% with 10% increments as well as the frequency level from 0.4GHz to 1GHz for each task before measuring the steady-state temperature.[5] Fig. 4.3 plots the measured steady temperature as dotted points while

[3] The detailed experimental setup will be given in §4.7.

[4] The activity factor α is normalized by the maximum power, i.e., $\alpha = 1$ means the maximum power dissipation.

[5] See §4.7 for more details of the experimental setup.

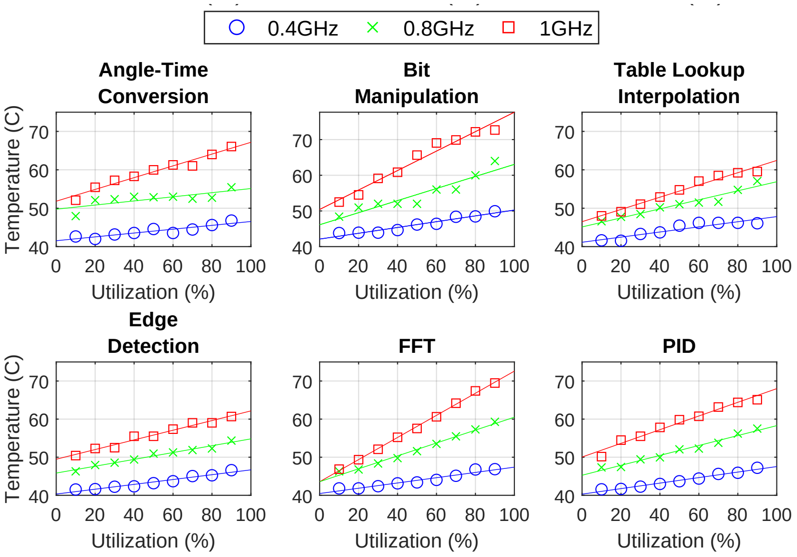


Figure 4.3: Model validation with varying utilizations.

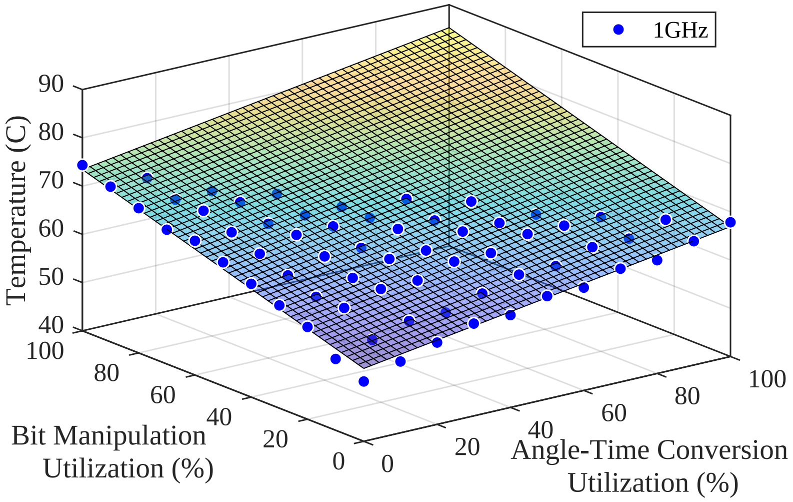


Figure 4.4: Model validation with two periodic tasks (*Bit manipulation*, *Angle-time Conversion*).

the estimations with our model are plotted as lines with an error up to 0.65°C.

Second, as shown in Fig. 4.4, we ran two tasks — bit manipulation and angle-time conversion — together on a single core at 1GHz by varying their utilizations. The result shows that the steady temperature of the processor linearly increases with each task's utilization (plotted as a linear surface) as formulated in Eq. (4.7). We also confirmed that the same tendency is observed for different numbers of tasks (i.e., 4 and 8 tasks) with an error up to 1.2°C.

Finally, we validated our model for different ambient temperatures from 20°C to 35°C. For each configuration, we repeated this 10 times with sufficient intervals, revealing an error of up to 0.98°C.

4.5 Adaptive Parameter Assignment

We now present how to adjust a voltage/frequency level and task period assignment according to the varying ambient temperature, called the *Adaptive Parameter Assignment Framework* (APAF). Specifically, we derive feasibility conditions for a parameter assignment, formulate a parameter optimization problem, and introduce a runtime strategy for adapting to varying ambient temperatures.

4.5.1 Parameter Assignment

We first consider the *feasible parameter assignment* problem for a given ambient temperature.

Definition 4.2 (Feasible parameter assignment)**.** Given a task set τ and the ambient temperature T_{amb}, determine V, f and p_i for every $\tau_i \in \tau$ such that if τ is feasible (i.e., it meets the thermal and timing constraints), it remains feasible even with the new parameter assignment.

To solve this problem, we introduce two conditions for a parameter assignment to be feasible with respect to thermal and timing constraints for a given ambient temperature, and then formulate an optimization problem to find a feasible parameter assignment.

Feasibility Condition. Recall that for a given task set τ, the processor temperature will eventually reach the steady temperature $T(\infty, \tau)$ of τ (defined in Eq. (4.7)) regardless of its schedule. Therefore, to meet the thermal constraint, the steady temperature $T(\infty, \tau)$ should be lower than or equal to the peak temperature limit T_{max},

$$\text{C1: } T(\infty, \tau) \leq T_{max}. \tag{4.8}$$

We define a *dynamic power demand* $PD(\tau)$ of τ as the total dynamic power demand by τ at the steady temperature, which is as follows:

$$PD(\tau) = \sum_{\tau_i} P_i \cdot \frac{e_i(f)}{p_i} = V^2 \cdot f \cdot \sum_{\tau_i} \alpha_i \cdot \frac{e_i(f)}{p_i}. \tag{4.9}$$

We also define a *dynamic power bound* $PB(T_{amb})$ of T_{amb} as the processor's maximum dynamic power at T_{amb} without exceeding T_{max}. We can derive $PB(T_{amb})$ by solving $T(\infty, \tau) = T_{max}$:

$$PB(T_{amb}) = \frac{T_{max} - T_{amb}}{R} - V \cdot (\beta_1 \cdot T_{max} + \beta_0). \tag{4.10}$$

Using these, the feasibility condition $C1$ with respect to the thermal constraint can be re-written as

$$\overline{\text{C1}}\text{: } PD(\tau) \leq PB(T_{amb}). \tag{4.11}$$

To meet the timing constraint, we use the well-known exact feasibility analysis by Liu and Layland [82]:

$$\text{C2: } \sum_{\tau_i} \frac{e_i(f)}{p_i} \leq 1. \tag{4.12}$$

If a parameter assignment satisfies both $\overline{\text{C1}}$ and C2, the steady temperature of τ is guaranteed not to exceed T_{max} without missing any task deadline when a task set is scheduled by an optimal scheduling algorithm. However, as can be seen in Eq. (4.6), a job schedule may affect a transient temperature $T(t, W(t))$, potentially violating the thermal constraint before reaching the steady temperature. To avoid this situation, we define the minimum idle-time $I_i^{min}(T_{amb})$ required for the execution of each job without violating the thermal constraint and include the term in C2 (to be detailed in §4.6). Then, the feasibility condition $C2$ can be extended to:

$$\overline{\text{C2}}\text{:}\ \sum_{\tau_i} \frac{e_i(f) + I_i^{min}(T_{amb})}{p_i} \leq 1. \tag{4.13}$$

Thus, if a parameter assignment exists that satisfies both $\overline{\text{C1}}$ and $\overline{\text{C2}}$, we can guarantee that a task set τ is feasible with respect to both thermal and timing constraints.

Parameter Optimization. We formulate the parameter assignment problem as an optimization problem subject to the feasibility conditions ($\overline{\text{C1}}$ and $\overline{\text{C2}}$):

$$\begin{aligned}
&\underset{f,p_i}{\textbf{maximize}} && \sum_{\tau_i} w_i \cdot \frac{1}{p_i} && (4.14)\\
&\textbf{s.t.} && \overline{\text{C1}}\text{:}\ PD(\tau) = V^2 \cdot f \cdot \sum_{\tau_i} \alpha_i \cdot \frac{e_i(f)}{p_i} \leq PB(T_{amb}) && (4.15)\\
& && \overline{\text{C2}}\text{:}\ \sum_{\tau_i} \frac{e_i(f) + I_i^{min}(T_{amb})}{p_i} \leq 1 && (4.16)\\
& && f \in [f_{min}, ..., f_{max}]. && (4.17)\\
& && \forall \tau_i \quad p_i^{\min} \leq p_i \leq p_i^{\max} && (4.18)
\end{aligned}$$

As an optimization goal, a QoS function associated with resource usage can be used as in [39, 111]. Our objective in Eq. (4.14) is to maximize the weighted sum of each task-rate $\frac{1}{p_i}$.[6] Eq. (4.17) specifies the discrete frequency scaling levels available on the processor. Eq. (4.18) specifies the minimum and maximum bounds of an allowable task period within $[p_i^{\min},\ p_i^{\max}]$. We use linear programming to determine a task period assignment for a given voltage/frequency level starting from the maximum level. If there is no solution, we lower the voltage/frequency level until a feasible solution is found. The computational complexity is $O(m \cdot n^{3.5})$ for n tasks and m frequency scaling levels [88].

[6] The value of w_i can be determined by the importance of each task.

4.5.2 Runtime Parameter Adaptation

We propose a runtime parameter adaptation strategy that samples ambient temperature variations and dynamically adjusts the voltage/frequency level and period assignment. For this, we need to determine when and how to adjust the parameter assignment. We set fixed points of the ambient temperature threshold $\{TS_{amb}(k)\}$, which are determined by

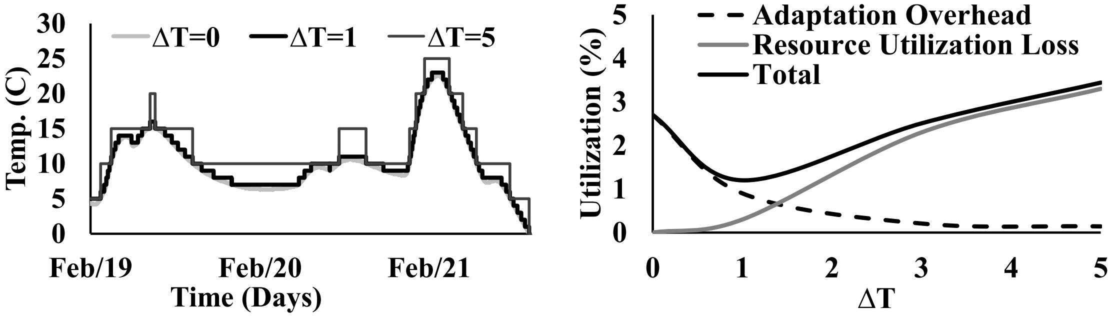
$$TS_{amb}(k+1) = TS_{amb}(k) + \Delta T, \tag{4.19}$$

where ΔT is a tolerable ambient temperature range.

Our runtime adaptation strategy periodically estimates the ambient temperature and adjusts the parameter assignment whenever the sampled ambient temperature is out of range $(TS_{amb}(k), TS_{amb}(k+1)]$ for any k. The parameter assignment in each range is determined by solving the optimization problem in Eq. (4.14) with the ambient temperature of $TS_{amb}(k+1)$. The challenge is how to choose ΔT and estimate the ambient temperature.

Determining a Tolerable Ambient Temperature Range. A trade-off exists between resource utilization and adaptation overhead when choosing ΔT. A smaller ΔT can achieve efficient resource utilization with a prompt response upon small ambient temperature changes at the expense of a high adaptation overhead. If the adaptation interval ΔT is too large, coarse-grained parameter adaptation incurs resource utilization loss.

To determine the optimal value of ΔT, we analyze the ambient temperature trace in Fig. 4.1a and compare the runtime overhead and resource efficiency depending on ΔT. Fig. 4.5a illustrates how our parameter adaptation responds to the varying ambient temperature for different values of ΔT. From the trace, we obtain the adaptation overhead and resource utilization loss for each value of ΔT.



(a) Different Adaptation Interval ΔT

(b) Impact of ΔT on Overhead

Figure 4.5: Runtime adaptation with (a) different different adaptation intervals and (b) the trade-off between adaptation overhead and resource efficiency

Fig. 4.5b illustrates the trade-off between resource efficiency and adaptation overhead, where the adaptation overhead (dotted line) decreases but the resource utilization loss (grey line) increases as ΔT increases. The adaptation overhead in our experimental setup (§4.7) was approximately 27ms. When adapting every sampling period ($\Delta T = 0$), the incurred processor utilization overhead was 2.7%. (Fig. 4.5b). We set the optimal value of ΔT to the point where the sum of the adaptation overhead and resource utilization loss (solid line) was minimized, which was $\Delta T = 1$°C.

4.6 Online Idle-time Scheduling

Thus far, we have discussed how to adaptively adjust the processor's voltage/frequency and task periods under the varying ambient temperature. We now consider how to schedule task/job executions and idle-times to meet both thermal and timing requirements. Specifically, we want to address the following problem, which we call the *schedule-generation* problem.

Definition 4.3 (Schedule generation)**.** Given the assignment of V, f, and $\{p_i\}$ (with APAF), determine a schedule of job executions and idle-times such that the processor temperature $T(t)$ does not exceed T_{max} at any time t while all jobs of all tasks $\tau_i \in \tau$ meet their deadlines.

To solve this problem, we must consider two key issues: 1) transient temperature $T(t)$ varies with the task running at any given time, and 2) ambient temperature T_{amb} also affects $T(t)$. Suppose that the processor has reached T_{max} (i.e., $T(t) = T_{max}$) and two tasks — a cold task τ_1 and a hot task τ_2 — are ready to run at time t. If a cold task τ_1 is scheduled, the temperature will decrease because $T_1^{\infty}(T_{amb}) \leq T_{max}$. By contrast, if a hot task τ_2 starts to run immediately, the temperature will increase (because $T_2^{\infty}(T_{amb}) > T_{max}$), and the thermal constraint will be violated. To avoid the processor temperature exceeding T_{max}, we must idle the processor to drop its temperature to a *safe* temperature before executing τ_2. With this safe temperature, the continued execution of τ_2 will not violate the thermal constraint. The main challenge is then how to derive a safe temperature and schedule idle-times to reach the temperature before executing each hot task. Note that each task has a different power dissipation, and thus the safe temperature may vary with tasks. Moreover, the amount of idle time required to reach a safe temperature varies with the ambient temperature. Without a proper idle-time scheduling decision, the result may end up with some undesirable situations, such as those where (a) the temperature exceeds T_{max} and/or (b) a task/job deadline is missed.

To resolve such problems, we develop a thermal-aware online idle-time scheduling policy that determines idle-times between the execution of tasks to meet both thermal and timing constraints. We assume that tasks are priority-ordered according to the earliest deadline first (EDF) policy. We calculate the *minimum idle-time* required for the execution of each task to avoid the aforementioned situation (a) and take the minimum idle-time into account in our adaptive parameter assignment to avoid situation (b). Our proposed online scheduling algorithm then makes a trade-off between the total amount of required idle-time and preemption overhead. In particular, it updates the available slack at runtime and effectively utilizes it to allocate more idle-time with much fewer preemptions while guaranteeing both thermal and timing constraints.

Calculating the Minimum Idle-Time. Now, we describe the relationship between the amount of necessary idle-time and the number of preemptions. We first consider the case of executing a hot task τ_i for $e_i(f)$ units without any preemption. We define the safe temperature of τ_i to execute for $e_i(f)$ units at T_{amb} (denoted by $T_i^{safe}(e_i(f), T_{amb})$) as the initial temperature at which the temperature reaches T_{max} after the execution of $e_i(f)$ units. The safe temperature can then be derived by solving the term $T(t)$ in Eq. (4.3) when $T_i(t + e_i(f)) = T_{max}$:

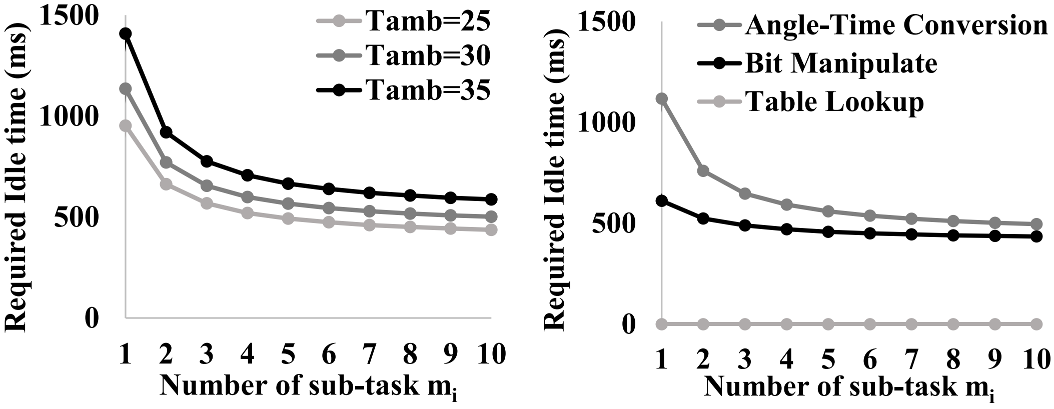
$$T_i^{safe}(e_i(f), T_{amb}) = T_i^{\infty}(T_{amb}) - \frac{T_i^{\infty}(T_{amb}) - T_{max}}{e^{-\frac{e_i(f)}{R \cdot C}}}. \tag{4.20}$$

Similarly, we can calculate the idle-time necessary to reach $T_i^{safe}(e_i(f), T_{amb})$ (denoted by $t_{idle}(e_i(f), T_{amb})$ by solving the term l in Eq. (4.5) when $T_0(t + l) = T_i^{safe}(e_i(f), T_{amb})$ and $T(t) = T_{max}$:

$$t_{idle}(e_i(f), T_{amb}) = R \cdot C \cdot ln(\frac{T_{max} - T_0^{\infty}(T_{amb})}{T_i^{safe}(e_i(f), T_{amb}) - T_0^{\infty}(T_{amb})}). \tag{4.21}$$

Now, let us consider a case where preemption is allowed specifically, each task τ_i is split into multiple — m_i ($m_i > 1$) — sub-tasks and idle-time is inserted in between. Likewise, by using Eqs. (4.20) and (4.21), we can calculate the safe temperature and idle-time required for executing each sub-task for $\frac{e_i(f)}{m_i}$ units. Then, the cumulative idle-time to execute m_i sub-tasks at T_{amb} can be calculated as $m_i \cdot t_{idle}(\frac{e_i(f)}{m_i}, T_{amb})$.

Fig. 4.6 depicts the cumulative idle-time as m_i increases. It is important to observe that the more sub-tasks there are, the less cumulative idle-time is required, as was also observed in [68]. In Fig. 4.6(a), it can be seen that the amount of required idle-time also depends on the ambient temperature. As shown in Fig. 4.6(b), each task requires a different amount of idle-time. Note that cold tasks — such as a table-lookup task — do not require idle-time. The results shown in Fig. 4.6 imply that the cumulative



(a) Different Ambient Temperature

(b) Different Tasks

Figure 4.6: Cumulative idle-time for (a) different ambient temperature and (b) different tasks decreases with the number of subtasks m_i.

idle-time can be reduced by splitting each task into more sub-tasks with frequent idling of the processor. However, the benefit of frequent idling becomes saturated as m_i increases, and the preemption overhead can no longer be ignored. Considering this, we derive the minimum idle-time for a task τ_i (denoted by $I_i^{min}(T_{amb})$) as follows. We calculate a decreasing amount of cumulative idle-time by taking derivative $\frac{\partial}{\partial m_i}(m_i \cdot t_{idle}(\frac{e_i(f)}{m_i}, T_{amb}))$, and find the value of m_i (denoted by m_i^{max}) where the value of the derivative becomes closest to the preemption cost for switching between active and idle states. Then, the minimum idle-time of τ_i can be calculated as follows:

$$I_i^{min}(T_{amb}) = m_i^{max} \cdot t_{idle}(\frac{e_i(f)}{m_i^{max}}, T_{amb}). \tag{4.22}$$

Note that it is sufficient to update the minimum idle-time of each task only when there exists any parameter change caused by our adaptive parameter assignment.

Guarantee of Thermal and Timing Constraints. For every invocation of a task τ_i, if the minimum idle-time is correctly scheduled before the execution of τ_i is finished, we can guarantee that the thermal constraint is never violated. The question then becomes how to guarantee the timing constraint when all tasks are scheduled together with their minimum idle-time. To address this, we derive a new feasibility condition by incorporating the minimum idle-time for each task. For a task set τ to be feasible under both thermal and timing constraints, every job of each task τ_i should have its minimum idle-time (for at least $I_i^{min}(T_{amb})$) and finish its execution (for at most its WCET $e_i(f)$) before its deadline. Then, a new feasibility condition can be derived by extending the utilization-based exact feasibility analysis by Liu and Layland [82]:

$$\sum_{\tau_i} \frac{e_i(f) + I_i^{min}(T_{amb})}{p_i} \leq 1. \tag{4.23}$$

We include the feasibility condition (Eq. (4.23)) in the optimization formulation for the parameter assignment presented in §4.5.1. This way, RT-TRM can guarantee both thermal and timing constraints.

Online Idle-time Scheduling. Building upon the parameter assignment obtained by APAF, if we divide each task τ_i into $m_i(I_i^{min}(T_{amb}))$ sub-tasks and evenly distribute the idle-time $I_i^{min}(T_{amb})$ between the execution of each sub-task, we can schedule all tasks without violating thermal and timing constraints. However, such *static* idle time allocation under pessimistic assumptions cannot efficiently utilize all available slack resources at runtime, which may in turn incur unnecessary preemption overheads. Therefore, we develop an online idle-time scheduling algorithm that reclaims unused resources and utilizes them to allocate *dynamic* idle-time for each task in an efficient manner. As a result, our algorithm can meet both thermal and timing requirements with much fewer preemptions.

Described below is our online idle-time scheduling algorithm. The scheduler is invoked upon the (i) release of a new job (`JOB_RELEASE`), (ii) completion of a job (`JOB_COMPLETION`), or (iii) update of frequency by APAF (`FREQ_UPDATE`). The scheduler keeps track of the worst-case remaining execution time, $e\_left_i(f)$ for the active job of τ_i. This is set to $e_i(f)$ on `JOB_RELEASE`, decremented as the job executes, updated according to the frequency change on `FREQ_UPDATE`, and set to 0 upon `JOB_COMPLETION`. Upon each invocation (either `JOB_RELEASE`, `JOB_COMPLETION`,

Algorithm IV.1 Slack calculation

```
$U = \sum_{\tau_i} \frac{e_i(f)+I_i^{min}(T_{amb})}{p_i}$
$p = 0$
for $i = n$ to 1, $\tau_i \in \{\tau_1, ..., \tau_n | d_1(t_{cur}) \leq \cdots \leq d_n(t_{cur})\}$ do
    {In reverse EDF order of tasks}
    $U = U - \frac{e_i(f)+I_i^{min}(T_{amb})}{p_i}$
    $q_i = \max\left(0, e\_left_i(f) + I_i^{min}(T_{amb}) - (1-U)\cdot(d_i(t_{cur}) - d_1(t_{cur}))\right)$
    $U = \min\left(1.0, U + \frac{e\_left_i(f)+I_i^{min}(T_{amb})-q_i}{d_i(t_{cur})-d_1(t_{cur})}\right)$
    $p = p + q_i$
end for
$S(t_{cur}, d_1(t_{cur})) = d_1(t_{cur}) - t_{cur} - p$
```

or `FREQ_UPDATE`), the scheduler updates the available slack $S(t_{cur}, d_1(t_{cur}))$ for the interval of $[t_{cur}, d_1(t_{cur}))$, where t_{cur} is the current time instant and $d_1(t_{cur})$ is the earliest absolute deadline among all released jobs whose deadline is after t_{cur}. Subsequently, the scheduler assigns slack $S(t_{cur}, d_1(t_{cur}))$ to tasks *in proportion to* their average power dissipation (i.e., $P_i \cdot \frac{e_i(f)}{p_i}$). The rationale for such a proportional slack distribution is that a task with higher power dissipation requires more idle-time. In this way, each task is assigned an amount of idle-time equal to

$$I_i(t_{cur}) = I_i^{min}(T_{amb}) + S(t_{cur}, d_1(t_{cur})) \cdot \frac{P_i \cdot \frac{e_i(f)}{p_i}}{\sum_{\tau_i} P_i \cdot \frac{e_i(f)}{p_i}}. \tag{4.24}$$

Based on the assigned idle time $I_i(t_{cur})$ and the remaining execution time $e\_left_i(f)$, the scheduler splits τ_i into $m_i(I_i(t_{cur}))$ sub-tasks and alternates the processor to be idle for $\frac{I_i(t_{cur})}{m_i(I_i(t_{cur}))}$ units and task execution for $\frac{e\_left_i(f)}{m_i(I_i(t_{cur}))}$ units.

Let us consider how to calculate slack $S(t_{cur}, d_1(t_{cur}))$. Our goal is to find the maximum amount of slack time, which may be available during the interval $[t_{cur}, d_1(t_{cur}))$, while guaranteeing 1) at least the minimum idle-time for all future jobs and 2) all future deadlines ($\geq t_{cur}$) are met. Algorithm IV.1 presents our slack calculation method. At time t_{cur}, we examine at the interval until the earliest absolute deadline $d_1(t_{cur})$ among all tasks as well as examine all tasks in reverse

EDF order; that is, the latest deadline first (Line 4). Note that tasks are indexed in EDF order (i.e., for τ_i and τ_k where $i < k$, $d_i(t_{cur}) \leq d_k(t_{cur})$). We assume that future task invocations require the worst-case execution and minimum idle- times, and thus their utilization is $\sum_{\tau_i} \frac{e_i(f)+I_i^{min}(T_{amb})}{p_i}$ (Line 1). We attempt to defer as much execution/idling as possible beyond $d_1(t_{cur})$ and compute the minimum amount of execution/idling p that must execute before $d_1(t_{cur})$ to meet all future deadlines (Lines 5–8). This step is repeated for all tasks. To calculate p, we use a similar approach used in [37, 97]. Then, the slack is set to the remaining time slots except for p over the interval $[t_{cur}, d_1(t_{cur}))$ (Line 10). The underlying principle behind our slack calculation is that EDF will determine a feasible schedule if the utilization in Eq. (4.23) is $\leq$1.0 at any time [23].

Runtime Complexity. At each invocation (either `JOB_RELEASE`, `JOB_COMPLETION`, or `FREQ_UPDATE`), our scheduling algorithm updates the slack by Algorithm IV.1 with the complexity of $O(n)$, where n is the number of tasks. Then, our algorithm allocates the slack to a job with the earliest deadline according to Eq. (4.24) with the complexity of $O(1)$. Thus, the total complexity is $O(n)$.

4.7 Evaluation

We implemented and evaluated RT-TRM on a commercial embedded processor for automotive and infotainment applications. Our evaluation focused on how it guarantees thermal and real-time constraints under various conditions.

Experimental Setup. Our evaluation platform was an i.MX6 [53] with ARM A9 supporting three discrete frequency levels (1, 0.8, and 0.4GHz) and corresponding voltage levels (1.25, 1.15, and 0.95V). The chip was equipped with an on-chip thermal sensor with a precision of 0.4°C. Table 4.2 specifies the power and thermal parameters of our target platform. We set the peak temperature constraint T_{max} to 60°C.[7]

[7] According to the mean-time-to-failure (MTTF) model [27], the thermal constraint of 60°C can

Table 4.2: Thermal parameters of the iMX6 processor.

R (°C/W)	C (J/°C)	β_1 (mA/°C)	β_0 (mA)	P_{max}(mW)
22	0.0454	0.435	611	3860

Table 4.3: WCET and min/maximum periods.

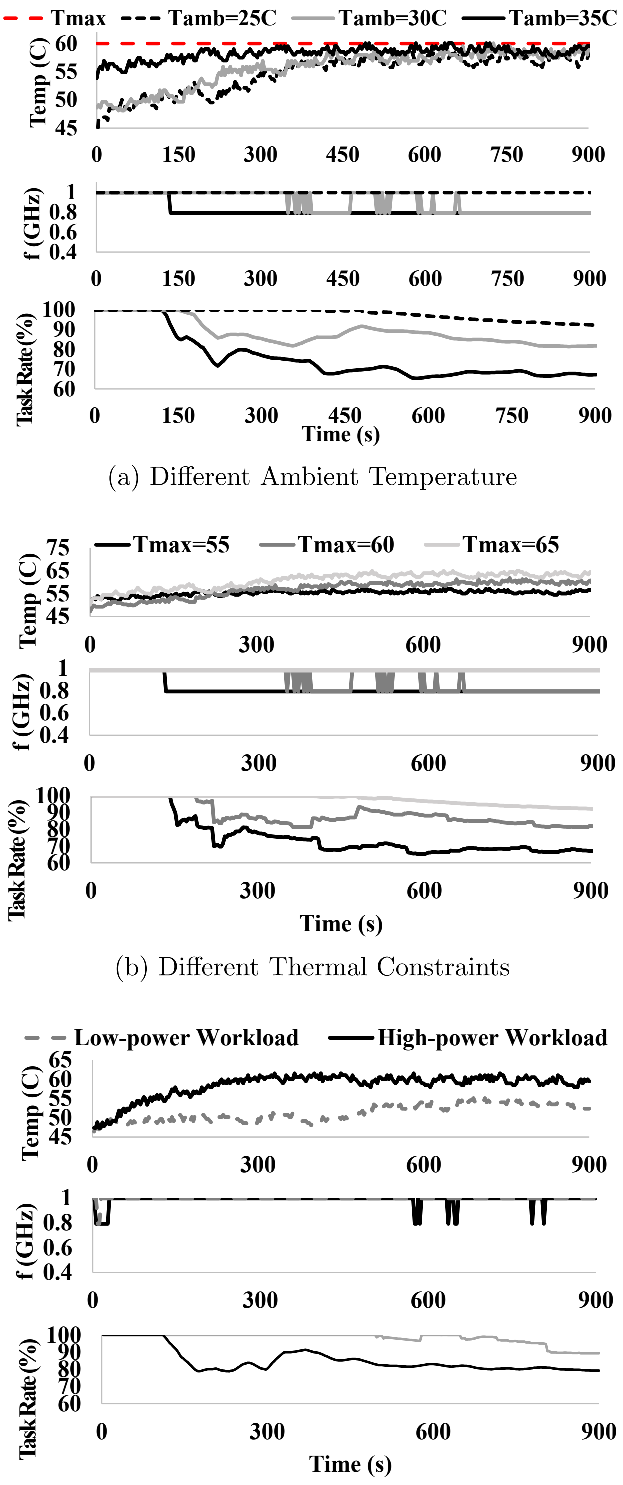
(s)	Angle	Bit	Table	Edge	FFT	PID
e_i	2.51	1.03	0.919	0.872	0.456	0.151
p_i^{min}, p_i^{max}	15, 30	6, 12	6, 12	5, 10	2.5, 5	1, 2

For demonstration purposes, we used realistic automotive workloads obtained from MiBench [57], including *Angle-time Conversion, Bit Manipulation, Table Lookup, Edge Detection, FFT, PID.* Table 4.3 provides the configuration of each workload.[8] We used real-time kernel [94] to periodically execute above benchmark applications. For idle-time scheduling, we generate a kernel idle thread to preempt task execution.

Handling Ambient Temperature Variation. To illustrate how RT-TRM adapts to various environmental conditions to meet the thermal constraint, we conducted a set of experiments at different ambient temperatures (25, 30, and 35°C). Fig. 4.7a plots the real-time traces of the processor temperature, frequency and task-rate. The task-rate was defined in Eq. (4.14) and normalized by the maximum rate. The results showed that RT-TRM effectively regulated the processor temperature below T_{max}. At an ambient temperature of 25°C (dotted line), RT-TRM was shown to be able to maintain the 1GHz processor frequency and 91.5% of the maximum task-rate. At 30°C (grey line), the processor frequency was switched between 1 and 0.8GHz, and the task-rate was dynamically adjusted, achieving 82.6% of the maximum task-rate. At 35°C (solid line), the processor frequency had to be reduced at a time around 150s to meet the thermal constraint, resulting in 65.6% of the maximum task-rate. We also looked closer at the results in Fig. 4.7a in a shorter time interval $(0, 40]$ and presented the execution behavior of the hottest task (bit manipulation) and

cover a typical vehicle warranty period of 10 years.

[8] Note that MiBench does not specify task period and execution time. We thus measure the worst-case execution time of each task in our experimental setup and synthetically assign a period range of each task proportional to the WCET.



(a) Different Ambient Temperature

(b) Different Thermal Constraints

(c) Different Power Dissipation

Figure 4.7: Experimental results of RT-TRM showing the processor temperature, frequency, and task-rate traces under (a) different ambient temperatures, (b) thermal constraints and (c) power dissipations.

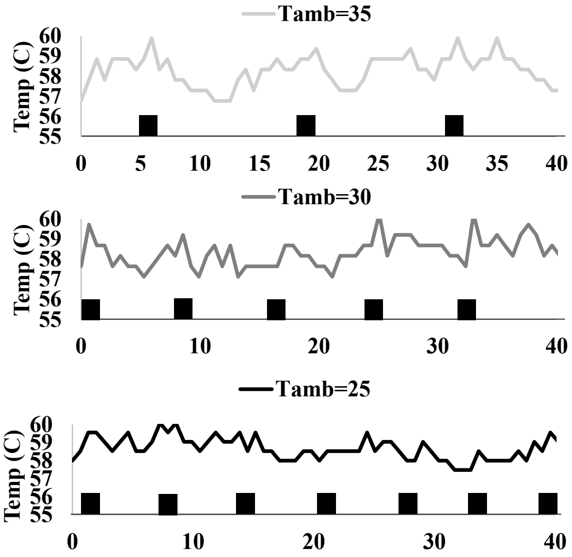


Figure 4.8: Job schedule of a task (*bit manipulation*) and the corresponding temperature variation by RT-TRM.

its corresponding temperature variation under RT-TRM as shown in Fig. 4.8. In the figure, the processor temperature increased whenever the bit manipulation task executed. When the ambient temperature was 35°C, a job of the bit manipulation task was invoked every 12 seconds, whereas it was invoked every 8 seconds (6 seconds) when the ambient temperature was 30°C (25°C). RT-TRM could adaptively adjust the processor frequency and task periods under different ambient temperatures while meeting both thermal and timing requirements.

Handling Different Thermal Constraints. Fig. 4.7b presents the results of RT-TRM under different thermal constraints (55, 60, and 65°C). Under the thermal constraint of 65°C, RT-TRM achieves a higher task-rate of 91% without reducing the core frequency. Under the thermal constraint of 55°C, it achieved a lower task-rate of 64.6% by reducing the core frequency to 0.8GHz to meet the thermal constraint. RT-TRM was shown to effectively control the processor temperature close to the thermal constraint and maximize resource utilization.

Handling Different Power Dissipations. Fig. 4.7c presents the results of RT-TRM for different power dissipation workloads under the thermal constraint of

Table 4.4: The number of preemptions and idle-time per job.

	Preemption	Used idle-time (s)
Static minimum idle-time	8.77	0.221
Online idle-time scheduling	1.18	0.275

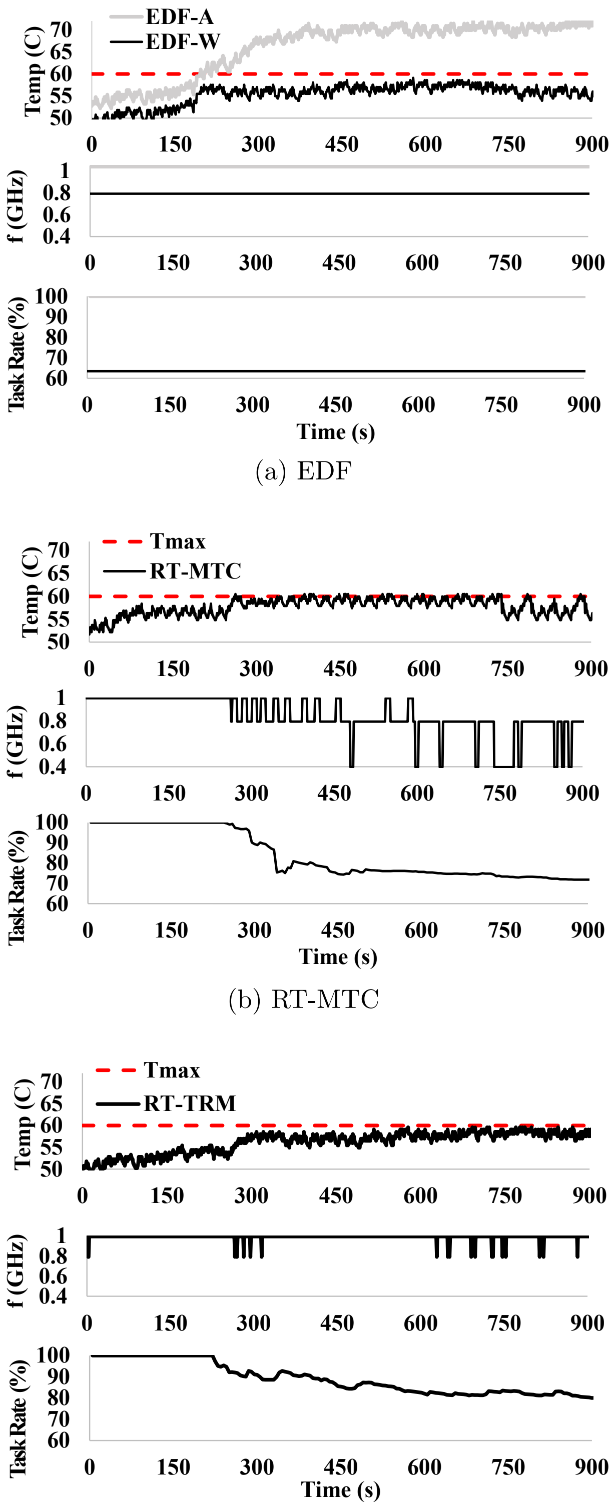
60°C. For the low power dissipation workload, RT-TRM achieved a higher task-rate of 88.9% with the maximum core frequency. Whereas, for the high power dissipation workload, it dynamically adjusted the task-rate to meet the thermal constraint, achieving a task rate of 79.3%.

Effect of Online Slack Usage. We also analyzed the effect of slack usage on online idle-time scheduling. During the above-mentioned experiment, we measured the total idle-time and number of preemptions per job, as shown in Table 4.4. Our online idle-time scheduling algorithm can assign more idle-time by 0.054s by efficiently utilizing runtime slack, and thus reducing the number of preemptions by 7.4x, compared with the static minimum idle-time allocation method. We observe that a small amount of additional idle-time could dramatically reduce the number of preemptions. By reclaiming the available slack at runtime, RT-TRM used 24.4% more idle-time to reduce 86.5% of preemptions without violating both thermal and timing constraints.

Performance Evaluation. Thus far, we have demonstrated how RT-TRM handles the dynamically changing ambient temperature and uses runtime slack to reduce the number of preemptions while satisfying thermal and timing constraints. We now focus on resource-efficiency and compare RT-TRM with two baseline approaches:

- EDF: static processor frequency and task period assignment under EDF [9]
- RT-MTC : dynamic processor frequency scaling using feedback control under EDF [55]
- RT-TRM: adaptive parameter assignment (§4.5) and online idle-time scheduling

[9] Parameters are assigned by solving the optimization problem Eq. (4.14)



(a) EDF

(b) RT-MTC

(c) RT-TRM

Figure 4.9: Experimental results of different schemes showing the processor temperature, frequency, and task-rate traces.

(§4.6)

Under EDF, we considered two static parameter assignments: one assumes an average ambient temperature of 25°C (EDF-A) and the other assumes the worst-case ambient temperature of 35°C (EDF-W). Under RT-MTC, if the processor utilization exceeds the schedulable utilization by lowering the processor frequency, task periods are scaled to meet deadlines. We used the following two metrics: (1) the percentage of time during the thermal constraint being violated and (2) the task-rate. A higher task-rate indicated higher resource-efficiency.

Fig. 4.9 compares the processor temperature, frequency and task-rate for three different thermal management schemes. EDF-A assigned the processor frequency of 1GHz and the task-rate of a 100%, whereas EDF-W assigned the frequency of 0.8GHz and a task-rate of 63.5% (Fig. 4.9a). Under EDF-A, the maximum temperature was 71.5°C, and thus the thermal constraint was violated 76.7% of the time. By contrast, under EDF-W, the thermal constraint was satisfied all the time with the maximum temperature of 59.1°C, but resources were severely under-utilized.

Under RT-MTC as shown in Fig. 4.9b, when the processor temperature hit the threshold at time 250s, the processor frequency was lowered to 0.8GHz. The temperature still exceeded the limit, and thus the frequency was lowered again to 0.4GHz at 750s. Due to the frequency reduced to the lowest level, the task-rate for RT-MTC was reduced to 67.2%. Although the feedback controller regulated the temperature close to the set point, it also violated the thermal constraint 3% of the time with a maximum temperature of 60.5 °C.

Fig. 4.9c shows that RT-TRM maintained the maximum processor frequency most of the time by adaptively adjusting the task periods, achieving a task-rate of 79.4% — an 18.2% improvement over RT-MTC. Note that parameters were adjusted every 3 seconds on average. Through efficiently scheduling idle-time, RT-TRM was always able to meet thermal constraints with a maximum temperature of 59.6 °C. The runtime overheads of parameter adjustment and idle-time scheduling were 27ms and 1ms, respectively.

4.8 Discussion

Thus far, we have presented a task-level power/thermal model and developed RT-TRM, which guarantees both thermal and timing constraints in the presence of dynamic ambient temperature variations. To demonstrate the importance of accounting for dynamic ambient temperature and different power dissipations, we considered a simple model and a platform as an example — a task-level linear power model and a uniprocessor platform, respectively.

We now discuss the applicability of RT-TRM to general models and multi-core platforms.

Task-level Power Variations. We assumed a task-level dynamic power model where power dissipation is constant across the jobs of a task and during the execution of a job. To guarantee the feasibility of RT-TRM without this assumption, we used the maximum power dissipation among all jobs as task-level dynamic power.[10]

Linear Power/Thermal Model. According to [20,83], we assumed that the leakage power P_{leak} (thermal resistance R) has a linear (no) relation with the processor temperature. We validated that such relations hold in a small temperature range (i.e., 20°C–35°C), but this may not hold in a wider temperature range. For example, the leakage power is known to increase exponentially as the temperature increases from 20°C to 120°C [84]. To apply RT-TRM in a wider temperature range, the leakage power and thermal resistance can be approximated using a piecewise linear model; specifically, the operating temperature range can be divided into multiple sub-ranges, each of which can be approximated using a linear model as shown in [84].

Multi-core Platform. To apply RT-TRM to multi-core platforms, we can consider

[10] Characterization of precise job-level power dissipation is part of our future work.

partitioned or global scheduling. In the case of partitioned scheduling, RT-TRM can be directly applied once a task-to-core assignment is made. Because the task-to-core assignment is known to be NP-hard [117], well-known heuristics can be used for the assignment. In the case of global scheduling, we need to extend our slack calculation in Alg. IV.1 to consider the concurrent execution of multiple tasks on a multi-core platform (which is part of our future work). Note that the calculation of the minimum idle-time for each task has nothing to do with a task schedule, and hence it can be directly applied for multi-core platforms. Moreover, on multi-core platforms, tasks scheduled on a core could affect the temperature of its neighboring cores. For this situation, a new thermal model is required that can capture the thermal effect between neighboring cores. Furthermore, the new idle-time must be calculated.

4.9 Conclusion

Emerging embedded real-time systems, such as connected cars and smartphones, pose new challenges to meeting timing constraints under the processors' thermal constraints. Such a system should consider a new *dynamic computation power bound* in addition to the conventional schedulable utilization bound. In this chapter, we developed a new thermal model that captures individual tasks' heat generation as their *activity factors*. We then developed two new mechanisms, *adaptive parameter assignment* and *online idle-time scheduling*. By tightly coupling the solutions of these two mechanisms, we can guarantee both thermal and timing constraints in the presence of dynamic ambient temperature variations. Our evaluation of RT-TRM on a realistic microcontroller using automotive benchmarks demonstrated the validity of the proposed thermal model and the effectiveness of RT-TRM at meeting both real-time and thermal constraints.

CHAPTER V

RT-TAS: Real-time Thermal-Aware CPUs–GPU Scheduling

5.1 Introduction

While modern embedded systems such as cars are increasingly using integrated CPUs–GPU system-on-chips (SoCs) with growing power dissipations, thermal challenges therein have become critical. Hardware cooling solutions such as fans have been used to lower chip temperature, but the cooling cost has been increasing rapidly, estimated to be US$3 per watt of heat dissipation [114]. Chip overheating not only incurs higher cooling costs but also degrades chip reliability [118], which may in turn risk physical safety. Thus, reducing on-chip temperature while simultaneously meeting application timing constraints have become key system design objectives.

Two key thermal characteristics must be considered for integrated CPUs–GPU platforms: (1) the platform's temperature imbalance and (2) different CPU and GPU power dissipations for different tasks. Our experimentation on a representative CPUs–GPU SoC showed the GPU's power dissipation to raise CPUs' temperatures (i.e., *CPUs–GPU thermal coupling*) at different rates, creating a large temperature imbalance among CPU cores (up to a $10°C$ difference); some (*hot*) CPU cores exhibit higher temperatures than others due to heat conducted from the GPU. In addition to this platform's temperature imbalance, our experimentation with automotive vision tasks has demonstrated a difference of up to 1.35× in CPU power dissipations and a difference of up to 2.68× in GPU power dissipations for different tasks; some (*hot*) tasks dissipate more power than others when executed on a CPU/GPU.

Motivation. These distinct thermal features for integrated CPUs–GPU systems pose significant challenges in partitioned fixed-priority scheduling of real-time tasks. Our CPU–GPU stress test demonstrates that the *concurrent* execution of hot tasks on both CPU and GPU generates a 24°C higher CPU temperature than CPU execution alone. Moreover, assigning a *hot* task to a *hot* core raises the temperature 5.6°C higher than assigning it to a cold core does, significantly increasing cooling costs and/or severely degrading app performance through drastic hardware throttling (to be detailed in §5.3, §5.6). This calls for thermal-aware task assignment and scheduling tailored to integrated CPUs–GPU platforms; a task-to-core assignment must distribute workloads to cores in a *thermally-balanced* manner by taking into account both the platform's temperature imbalance and different power dissipations of tasks; furthermore, a scheduling decision on CPUs and the GPU must be made *cooperatively* to avoid any burst of power dissipations on a CPUs–GPU platform while guaranteeing all app timing constraints.

Numerous thermal-aware scheduling schemes have been proposed for real-time uni-processor systems [3, 77] and multiprocessor systems [10, 26, 80]. They usually employ idle-time scheduling [77], DVFS scheduling [80], or a thermal-isolation server [10] to regulate the chip temperature at runtime. Although these prior studies have made many contributions to thermal-aware real-time scheduling, they are not directly applicable to integrated CPUs–GPU platforms because they have not considered the thermal effect of GPU workloads on CPUs, i.e., CPUs–GPU thermal coupling. GPU thermal management has also been studied for non-real-time systems [42, 99, 107, 112]. Studies have recognized thermally-efficient cores [107] and demonstrated the platform's thermal imbalance through infrared imaging [42]. However, they have not been suitable for safety/time-critical systems such as in-vehicle vision systems.

To the best of our knowledge, no studies exist on thermal-aware task assignment and scheduling of real-time tasks on CPUs–GPU platforms while accounting for the platform's temperature imbalance.

In this chapter, we propose a new *Real-Time Thermal-Aware Scheduling* (RT-TAS) framework that accounts for not only the platform's temperature imbalance but also diverse power dissipations of app tasks running on integrated CPUs–GPU platforms. RT-TAS generates a thermally-balanced task-to-core assignment and co-schedules CPUs and GPU to reduce the maximum chip temperature while meeting task/job deadlines.

We first capture the different CPUs' temperatures caused by the GPU's power dissipation with core-specific GPU thermal coupling coefficients. We analyze the effect of executing individual tasks on CPUs and GPU temperatures by taking the thermal coupling into account and validating such a thermal coupling effect on a representative CPUs–GPU platform with automotive vision workloads. Second, we introduce the notion of *thermally-balanced* task-to-core assignment to gauge the heat distribution across cores on a CPUs–GPU platform and derive a sufficient condition for an assignment to be thermally-balanced, while simultaneously considering CPUs–GPU thermal coupling. We then develop a thermally-balanced task-to-core assignment called T-WFD, which equilibrates the platform's thermal imbalance by considering different power dissipations of tasks while preserving schedule feasibility.

Third, building on a thermally-balanced assignment, we develop an online scheduling policy called *CPU–GPU co-scheduling*, for CPUs and GPU. It determines which tasks to schedule on CPUs by considering the task running on its counterpart (GPU), and vice versa, to avoid simultaneous executions of hot tasks on both CPUs and the GPU, thus mitigating excessive temperature increase without missing any task deadline.

Finally, we implemented RT-TAS on a representative CPUs–GPU platform [9] and evaluated it with automotive vision workloads [100], demonstrating its effectiveness at reducing the maximum temperature by $6-12.2°C$ compared with existing approaches without violating timing constraints; this provided a reliable response time under a given chip temperature limit. This $6°C$ reduction translates to a 1.52× improvement in chip lifetime reliability [118] or savings on cooling costs of US$ 15.6 per chip [81, 114].

Contribution. This chapter makes the following contributions:

- Demonstration of the importance of co-scheduling CPUs and GPU while accounting for their thermal coupling (§5.3);
- Empirically capturing CPUs–GPU thermal coupling effect and temperature differences among CPU cores (§5.4);
- Development of thermally-balanced task-to-core assignment and CPUs–GPU co-scheduling (§5.5);
- Implementation and evaluation of RT-TAS on a popular CPUs–GPU platform with automotive vision workloads (§5.6).

5.2 Related Work

Prior research in the field of real-time systems has focused on thermal-aware task and DVFS scheduling while meeting timing constraints for uni-processor [3, 77], and multiprocessor platforms [10, 26, 80]. Kumar *et al.* [77] proposed a thermal shaper to regulate the runtime chip temperature by inserting idle periods. Lampka *et al.* [80] proposed a history-aware dynamic voltage/frequency scaling (DVFS) scheme that raises the core frequency only in case of potential timing violations. A thermal-isolation server was proposed in [10] to avoid thermal interference between tasks in temporal and spatial domains with thermal composability. However, these solutions did not consider the thermal effect of GPU workloads on CPUs, i.e., CPUs–GPU thermal coupling, and thus they are not directly applicable to integrated CPUs–GPU platforms.

Studies have been conducted on GPU thermal management for non-real-time systems [42, 99, 107, 112]. Singla et al. provide a thermal modeling methodology via system identification on a CPU and GPU mobile platform and present a proactive DTM policy to prevent thermal violations [112]. Prakash et al. proposed CPU–GPU cooperative frequency scaling for a mobile gaming app [99]. The notion of a thermally-efficient core was proposed in [107], where the CPU core less impacted by GPU heat dissipation was identified offline, and tasks were assigned in the order of thermally-efficient cores. Infrared imaging characterized the CPUs–GPU thermal coupling, introducing scheduling challenges [42]. Although all of the aforementioned studies have made valuable contributions, they have not dealt with the timing constraint when applying DVFS or scheduling tasks, rendering them infeasible for time-critical embedded systems.

To the best of our knowledge, no prior work has addressed the challenges of thermal-aware assignments and the scheduling of real-time tasks on CPUs–GPU platforms while accounting for the platform's temperature imbalance. Unlike the state of the art, RT-TAS captures both CPUs–GPU thermal coupling and power-dissipation variations of tasks to lower the maximum temperature of thermally-unbalanced CPUs–GPU platforms while meeting the app timing constraint. We implemented and evaluated RT-TAS, demonstrating its effectiveness on a representative CPUs–GPU platform with automotive vision workloads.

5.3 Motivation

We first present a case study to demonstrate the distinct thermal characteristics of integrated CPUs–GPU systems and describe the challenges faced therein.

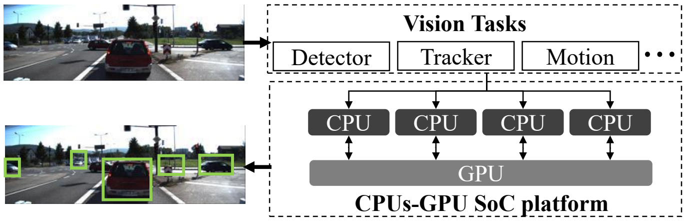


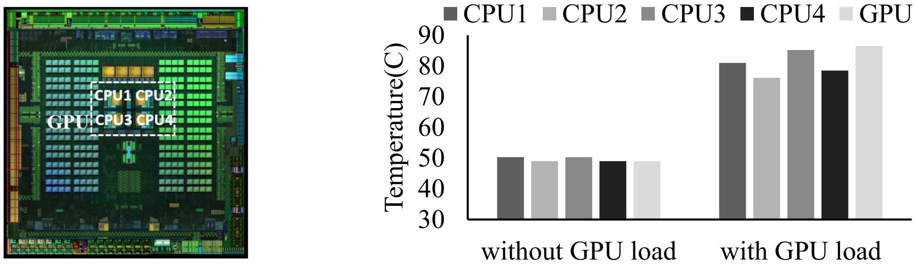
Figure 5.1: Example of an embedded vision systems.

5.3.1 Target System

We consider an automotive vision system — a prototypical real-time CPUs–GPU system — composed of multiple CPU cores and one GPU core running various real-time vision tasks (Fig. 5.1) . Typical vision apps include feature detectors, object trackers, and motion estimators [100]. A feature detector captures features to detect various objects, such as cars, road signs, and pedestrians; an object tracker maps and tracks moving/standing objects in consecutive input frames; and a motion estimator determines the motion/movement between consecutive frames to predict objects' motions. Real-time processing of these tasks relies on a GPU that supplements the computing capabilities of the primary CPUs. Each vision task consists of CPU and GPU sections of computation. To use the GPU, the CPU transfers data to the GPU memory and calls GPU functions. The GPU then performs the required computation and returns the results back to the CPU. See [48] for GPU operation details for real-time apps.

5.3.2 Thermal Characteristics of CPUs–GPU Platforms

To understand the thermal characteristics of CPUs–GPU platforms, we conducted experiments on an Nvidia Tegra X1 [9] equipped with four CPUs and a GPU with representative vision workloads [100]. Here, we highlight two key findings from this experimentation. First, GPU power dissipation raise CPUs' temperatures significantly at different rates, creating a large temperature imbalance on the platform



(a) CPUs–GPU SoC [9]

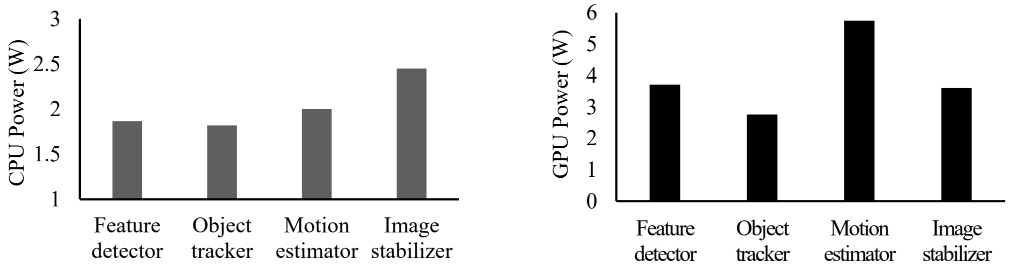
(b) Platform's temperature imbalance

Figure 5.2: CPUs surrounded by the GPU cluster on a SoC (Tegra X1), where the GPU's power dissipation affects the CPUs' temperatures, creating temperature imbalance across the CPUs.

cores. Second, different tasks dissipate different amounts of power on the CPU and GPU cores, i.e., some tasks are GPU-intensive and others are CPU-intensive.

Temperature Imbalance. In typical embedded vision platforms, unlike in desktops/servers, CPU and GPU cores are integrated on a single SoC (Fig. 5.2a) for cost, power, and communication efficiency [118]. CPU cores are usually surrounded by a GPU cluster [42]; hence, the GPU's power dissipation greatly affects CPU core temperatures because of heat transfer. To understand the GPU's thermal impact on CPU cores, we measured the temperatures of CPU and GPU cores with and without GPU workload.[1] Fig. 5.2b corroborates CPUs–GPU thermal coupling where the GPU workload raises the CPU cores' temperatures from $50°C$ to $77°C$ on average without any CPU workload. More crucially, we observed a significant temperature difference across CPU cores up to 10 °C (CPU2 vs. CPU3) in the presence of GPU workload. This imbalance was caused by CPU3's close proximity to the GPU cluster, and thus the significant impact of GPU power dissipation (Fig. 5.2a). We refer to CPU cores with higher (lower) temperature than the average in the presence of GPU workload as *hot* (*cold*) cores. For example, in this example, CPU1/CPU3 are hot and CPU2/CPU4 are cold cores.

[1] Note that in this motivational experiment, we used the GPU thermal benchmark [41], i.e., CPU remains idle, to minimize the impact of each CPU's own power dissipation.



(a) CPU Power

(b) GPU Power

Figure 5.3: Average power dissipations of (a) CPU and (b) GPU vary greatly by application tasks.

We conducted the same experiments on other SoCs — a Snapdragon 810 and an Exynos 5420 — with different chip layouts and observed similar trends of temperature imbalance [4]. Existing studies have also reported large temperature imbalances in various integrated CPUs–GPU platforms, such as MD A10-5700 [42] and Trinity APU [96].

Power Dissipations of App Tasks. In addition to the underlying platform's temperature imbalance, different tasks incur significantly different amounts of power dissipation on CPU and GPU. Fig. 5.3 plots the average power dissipations on (a) CPU and (b) GPU of sample vision workloads. We refer to tasks with power dissipations higher (lower) than the average as *hot* (*cold*) tasks (a *hot/cold* task is defined formally in Section 5.5.2). On the CPU, the image stabilizer is the hottest, dissipating 1.35× more power than the coldest, the object tracker. On the GPU, the motion estimator is the hottest, dissipating 2.68× more power than the coldest, the object tracker.

5.3.3 Why Thermal-Aware Task Scheduling?

We now demonstrate how the above-mentioned features can adversely affect system performance and reliability if they are not figured into task scheduling on integrated CPUs–GPU platforms. For a motivational purpose, we ran high-power

CPU and GPU workloads[2] on our testbed for 10 minutes and measured the maximum CPU temperature for the following three cases: i) the simultaneous execution of both CPU and GPU workloads; ii) the execution of CPU workload alone; and iii) no execution (idling). For cases i), ii), and iii), the maximum temperature were recored as 79.6, 55.6, and 50.3°C, respectively. Simultaneous CPU and GPU executions made the CPU temperature 24°C higher than the case of CPU execution alone. Moreover, assigning the CPU workload to a *hot* core resulted in 85.2°C, a CPU temperature increase of 5.6°C compared with the CPU workload being assigned to a *cold* core. Such an excessive temperature increase/imbalance may result in i) high cooling costs, ii) poor reliability, and/or iii) performance degradation caused by thermal throttling,[3] making it likely to extend the response time of tasks beyond their deadlines.

To avoid this, we need a new thermal-aware task assignment and scheduling framework that captures not only the platform's thermal imbalance but also task power-dissipation variations to mitigate excessive temperature rises. Thus, assigning *hot* tasks to *hot* cores without capturing the underlying temperature gap can raise the maximum temperature significantly. Traditional load-balancing schemes that evenly distribute workloads to CPU cores can be *thermally-unbalanced* because they do not consider the distinct thermal characteristics of individual cores. We must, therefore, accurately capture the platform's temperature imbalance and distribute tasks in a *thermally-balanced* manner to lower the maximum temperature.

In addition to the platform's temperature imbalance, tasks' different CPU/GPU power-dissipation variations make the scheduling of app tasks on CPUs and the GPU critical. Scheduling CPUs and GPU *independently* may adversely affect the peak temperature if both CPUs and the GPU simultaneously run *hot* tasks, both dissipating a large amount of heat. Therefore, we need to *cooperatively* schedule CPU

[2] We chose the CPU and GPU workloads from the thermal benchmark [41, 105] designed for CPU/GPU stress tests with high power dissipations of 4.67W and 9.89W, respectively.

[3] Thermal throttling is a hardware technique that can lower the processor's frequency on-the-fly to reduce the amount of heat generated by the chip.

and GPU computations to avoid a burst of power dissipations on the CPUs–GPU platform while simultaneously meeting all timing constraints.

5.4 CPUs–GPU System Model

This section presents the task execution and power models, analyzes how the power dissipations of tasks are converted into chip temperatures by taking the thermal coupling into account, and presents a validation of the models on a CPUs–GPU platform running various vision workloads.

5.4.1 Task Execution Model

Each independent[4] task $\tau_i \in \tau$ can be represented as $(p_i, d_i, \eta_i, e^C_{i,j}, e^G_{i,j})$, where p_i is the task period, d_i is its relative deadline equal to p_i, η_i is the number of GPU sections of computation that are enclosed by $\eta_i + 1$ CPU sections of computation, and $e^C_{i,j}$ and $e^G_{i,k}$ are the worst-case execution times (WCETs) of the j-th CPU section and the k-th GPU section, respectively. Let $e^C_i = \sum_{j=1}^{\eta_i+1} e^C_{i,j}$ be the total WCET of all the CPU sections, and $e^G_i = \sum_{k=1}^{\eta_i} e^G_{i,k}$ be the total WCET of all the GPU sections. For tasks without a GPU section, $\eta_i = e^G_i = 0$. We also define the CPU and GPU utilizations of τ_i as $u^C_i = \frac{e^C_i}{p_i}$ and $u^G_i = \frac{e^G_i}{p_i}$, respectively, and define the total utilization of τ_i as $u_i = u^C_i + u^G_i$.

Under partitioned fixed-priority preemptive scheduling, each task τ_i is statically assigned to a CPU with a unique priority and let $p(\pi_c)$ be the set of tasks assigned to $\pi_c \in \{\pi_1, \ldots, \pi_m\}$. Let $hp(\tau_i)$ ($lp(\tau_i)$) be the set of all tasks with a priority higher (lower) than τ_i. Likewise, Let $hpp(\tau_i)$ ($lpp(\tau_i)$) be the set of higher (lower)-priority tasks assigned to the same CPU as τ_i. GPU is a shared resource among tasks, and it is modeled as a critical section protected by a suspension-based mutually-exclusive

[4] Assuming "independent" tasks does not lower the general applicability of our approach, since one can use shared buffers to eliminate inter-task dependencies as shown in [75].

lock (mutex) because most contemporary GPUs perform their assigned computation non-preemptively. The GPU access is then made with the MPCP protocol, a well-known locking-based GPU access scheme [56, 95]. Under this protocol, a task requesting access to a lock held by a different task is suspended and inserted into a priority queue. During that time, other ready tasks may use the CPU. When the lock is released, a task in the priority queue is woken and granted access to the GPU. At a time instant, a task is either i) executing its CPU section, ii) executing its GPU section, or iii) idle.

Response Time Analysis. Under partitioned fixed-priority scheduling with the MPCP protocol, the worst-case response time (WCRT), w_i, of τ_i can be calculated iteratively using the following expression:

$$w_i^{a+1} = e_i^C + e_i^G + I_i^a + B_i^a, \tag{5.1}$$

where I_i^a is τ_i's preemption delay caused by higher-priority tasks and B_i^a is the blocking time for τ_i to acquire the GPU lock [95]. Note that the initial value w_i^0 is set to $e_i^C + e_i^G$, and the iteration halts when $w_i^{a+1} > d_i$ (unschedulable) or $w_i^{a+1} = w_i^a$ (the response time no larger than w_i^a). To derive I_i^a and B_i^a, we use the job-driven response time and blocking time analyses in [95]. A task τ_i can be preempted by higher-priority tasks τ_h running on the same CPU, i.e., $hpp(\tau_i)$. The number of jobs of τ_h released during the execution of a single job of τ_i is at most $\left\lceil \frac{w_i + w_h - e_h^C}{p_h} \right\rceil$. Then, I_i^a is derived as

$$I_i^a = \sum_{\tau_h \in hpp(\tau_i)} \left\lceil \frac{w_i + w_h - e_h^C}{p_h} \right\rceil \cdot e_h^C. \tag{5.2}$$

The blocking time for τ_i to acquire the GPU lock under the MPCP protocol can be divided into i) direct blocking (B_i^{dr}), which occurs when there is a task using τ_i's requested resource, and ii) prioritized blocking (B_i^{pr}), which occurs when lower-priority tasks executing with priority ceilings preempt the execution of τ_i. Using the blocking time analysis in [95], B_i^a is derived as

$$B_i^a = B_i^{dr} + B_i^{pr}, \tag{5.3}$$

where

$$B_i^{dr} = \eta_i \cdot \max_{\tau_l \in lp(\tau_i)} e_{l,j}^G + \sum_{\tau_h \in hp(\tau_i)} \left\lceil \frac{w_i + w_h - e_h^C}{T_h} \right\rceil \cdot e_{h,j}^G,$$
$$B_i^{pr} = \sum_{\tau_h \in lpp(\tau_i)} \left\lceil \frac{w_i + d_l - e_l^C}{T_l} \right\rceil \cdot e_l^G.$$

Detailed proof of this can be found in [95]. Then, we can check the schedulability of a task set as presented in the following lemma:

Lemma 5.1. *[95] A task set τ is schedulable if*

$$\forall \tau_i \in \tau, \; w_i \leq d_i. \tag{5.4}$$

5.4.2 CPU and GPU Power-dissipation Model

As shown in Fig. 5.3, CPU and GPU power dissipations are found to vary significantly with the executing task. This is becausse individual tasks realize distinct vision algorithms with different CPU and GPU sections that incur different CPU and GPU power dissipations. Thus, we model different power dissipations during the CPU execution (P_i^C) and GPU execution (P_i^G) of τ_i. Because every τ_i generates a sequence of CPU jobs, each with execution time e_i^C, at the interval of p_i time units, the average CPU power dissipation by τ_i is $P_i^C \cdot u_i^C$. Likewise, the average GPU power dissipation of τ_i is $P_i^G \cdot u_i^G$. Given a task set τ and a task-to-core assignment Λ, the average CPU and GPU power dissipations can be calculated as follows:

$$P_{\pi_c}(\Lambda) = \sum_{\tau_i \in p(\pi_c)} P_i^C \cdot u_i^C, \quad P_{\pi_g}(\Lambda) = \sum_{\tau_i \in \tau} P_i^G \cdot u_i^G, \tag{5.5}$$

where π_c denotes a CPU core among the set of CPUs (i.e., $\pi_c \in \{\pi_1, \ldots, \pi_m\}$).

5.4.3 Platform's Thermal Model

To translate the power dissipations of tasks into chip temperature together with the consideration of CPUs–GPU thermal coupling, we adopt a core-level thermal circuit model[5]:

$$T_\pi(t) = T_A + R \cdot P_\pi(t) + R \cdot C \cdot \frac{dT_\pi(t)}{dt} \tag{5.6}$$

where $T_\pi(t) = [T_{\pi_1}(t), \ldots, T_{\pi_m}(t), T_{\pi_g}(t)]$ is an $m+1$ element vector of CPU and GPU temperatures at time t; T_A is also an $(m+1)$-element vector of ambient temperature; $P_\pi(t) = [P_{\pi_1}(t), \ldots, P_{\pi_m}(t), P_{\pi_g}(t)]$ is an $(m+1)$-element vector of power dissipated by each CPU or GPU at time t; R represents an $(m+1) \times (m+1)$ matrix of thermal resistances between each component describing the heating impact of each component on each other component; and C is a diagonal matrix with the thermal capacitance of each component. With the thermal circuit model shown in Eq. (5.6), if the average power of a processor is $P_\pi(t)$ over a time period t, then the *transient* temperature $T_\pi(t)$ at the end of this period is

$$T_\pi(t) = e^{(R \cdot C)^{-1} \cdot t} \cdot T_\pi(0) + (1 - e^{(R \cdot C)^{-1} \cdot t}) \cdot (T_A + R \cdot P_\pi(t)). \tag{5.7}$$

where $T_\pi(0)$ is the initial temperature of the processor. One can observe from Eq. (5.7) that the temperature will increase/decrease toward and eventually reach $T_A + R \cdot P_\pi(t)$ in the steady state. We define the *steady-state* temperature T_π of a processor as

$$T_\pi = T_A + R \cdot P_\pi. \tag{5.8}$$

[5] Note that this thermal model has been shown to be reasonably accurate [3, 112].

Table 5.1: Thermal coupling coefficients for the Tegra X1 (°C/W)

$$R = \begin{bmatrix} R_1 & R_{1,2} & R_{1,3} & R_{1,4} & R_{1,g} \\ R_{2,1} & R_2 & R_{2,3} & R_{2,4} & R_{2,g} \\ R_{3,1} & R_{3,2} & R_3 & R_{3,4} & R_{3,g} \\ R_{4,1} & R_{4,2} & R_{4,3} & R_4 & R_{4,g} \\ R_{g,1} & R_{g,2} & R_{g,3} & R_{g,4} & R_g \end{bmatrix} = \begin{bmatrix} 2.54 & 1.66 & 1.68 & 1.68 & 2.20 \\ 1.66 & 2.37 & 1.71 & 1.73 & 1.43 \\ 1.68 & 1.71 & 2.93 & 1.72 & 2.27 \\ 1.68 & 1.73 & 1.72 & 2.62 & 1.76 \\ 1.50 & 1.71 & 1.60 & 1.71 & 1.87 \end{bmatrix}$$

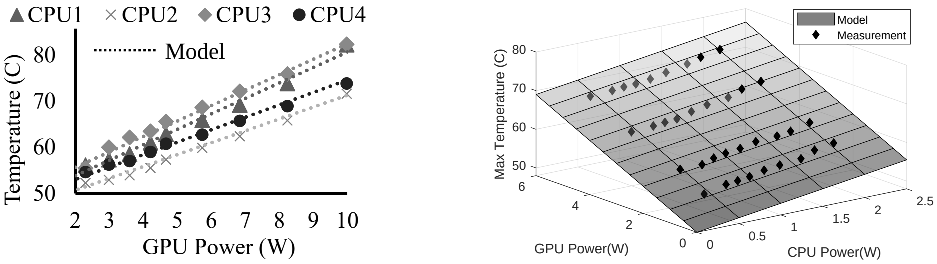
The steady-state temperature of π_x for a given task-to-core assignment Λ (denoted by $T_{\pi_x}(\Lambda)$) can be computed as follows:

$$T_{\pi_x}(\Lambda) = T_A + R_x \cdot P_{\pi_x}(\Lambda) + \sum_{\pi_y \in \pi \setminus \pi_x} R_{x,y} \cdot P_{\pi_y}(\Lambda) \tag{5.9}$$

where R_x represents the heating impact of π_x by itself, and $R_{x,y}$ represents the heating impact of other CPU and GPU cores π_y on π_x caused by the thermal couplings. Note that thermal coupling coefficients $R_{x,g}$, $1 \leq x \leq m$, capture the different impact of GPU heat dissipation on other cores depending on the thermal properties and chip layout; for example, how cores are geometrically positioned w.r.t. GPU.

5.4.4 Parameter Identification and Validation

Parameter Identification. The thermal resistance R and capacitance C are SoC-specific parameters in the platform's thermal model in Eq. (5.6), . While considering the CPUs–GPU thermal coupling, we identify these SoC-specific parameters using a typical thermal parameter identification process [33] as follows. We ran a GPU benchmark [41] on the GPU with CPU cores maintained as idle and measured the power dissipation of the GPU and steady-state temperatures of the CPU and GPU cores. We then determined each core's thermal coefficient w.r.t. GPU heating impact using Eq. (5.8). Similarly, we ran a CPU benchmark [105] on each CPU core, one at a time, and determined each core's thermal coefficient w.r.t. CPU heating impact. From these results, we identified the thermal-coupling coefficients using $R_{x,y} = \Delta T_{\pi_x}/P_{\pi_y}$ as in Table 5.1. Note that the thermal coefficient values



(a) Varied GPU power

(b) Varied CPU/GPU power

Figure 5.4: (a) CPU temperatures resulting from varied GPU power dissipations and (b) maximum CPU temperature resulting from varied CPU and GPU power dissipations.

shown in Table 5.1 are for an Nvidia Tegra X1. Through applying the above parameter identification process to other SoCs, we can directly apply our thermal model and the proposed thermal-aware scheduling framework to other CPUs–GPU SoCs.

Model Validation. To confirm that the CPUs–GPU thermal coupling model correctly represents the platform's thermal behavior, we measured the maximum temperature of the CPUs and GPU and compared it with the model's estimation under various settings. We validated our model by varying (i) GPU power settings and (ii) both CPU and GPU power dissipations with vision workloads as shown in Fig. 5.4.

Fig. 5.4a plots the measured CPU temperatures resulting from the GPU's varying power dissipation. CPU temperatures linearly increased with the GPU's power dissipation at different rates, which were captured by core-level thermal coupling coefficients. As the GPU's power dissipation increased from 2.2W to 8.2W, the temperature of CPU3 increased at most by 14.3 °C while that of CPU2 increased by 9.4 °C. Fig. 5.4b plots the maximum chip temperature with varying CPU and GPU power dissipations. The results shows that the maximum chip temperature linearly increases with both CPU and GPU power dissipations, as in Eq. (5.9).

5.5 Thermal-Aware Scheduling

To achieve both G1 and G2 in integrated CPUs–GPU platforms, we propose RT-TAS which takes both the platform's temperature imbalance and different power dissipations of tasks into account for the assignment and scheduling of real-time tasks. To this end, we first introduce a sufficient condition for a task-to-core assignment to be thermally-balanced in the presence of the underlying platform's thermal imbalance among CPU cores, and then present a thermally-balanced assignment called T-WFD, which equilibrates the platform's thermal imbalance by considering different power dissipations of tasks while preserving feasibility (§5.5.1). Building upon the thermally-balanced assignment, we then present an online CPU–GPU co-scheduling policy that cooperatively schedules jobs to avoid simultaneous executions of hot tasks on both CPUs and the GPU, and thus effectively reduces the peak chip temperature while meeting task deadlines (§5.5.2). Whereas our task-to-core assignment algorithm minimizes the maximum steady-state temperature of CPU cores, our co-scheduling policy regulates CPUs' and the GPU's power dissipations to mitigate the increase of transient temperature.

5.5.1 Thermally-Balanced Assignment

We now formally state the *task-to-core assignment* problem.

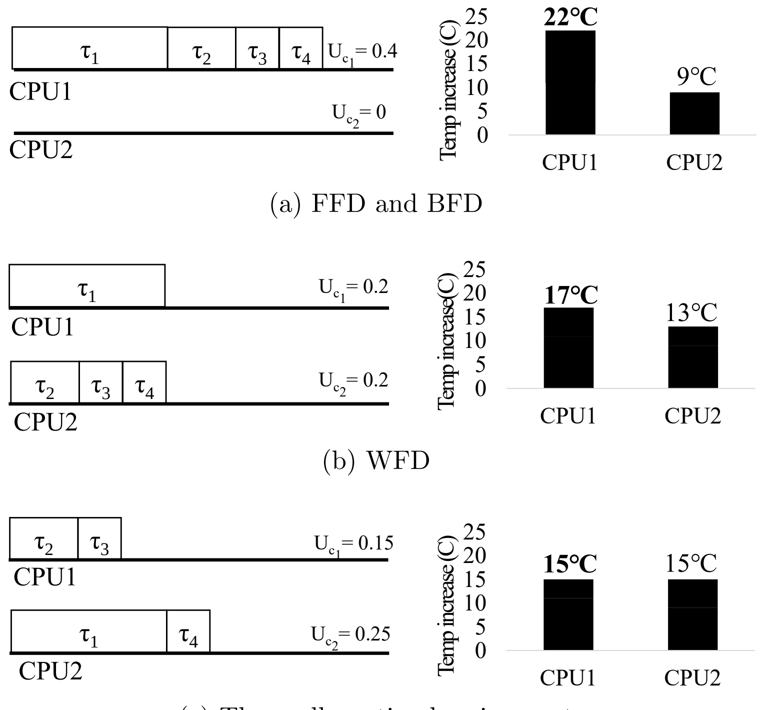
Definition 5.2 (Task-to-core assignment)**.** Given a task set τ and a CPUs–GPU platform π, find a mapping from the tasks of τ to the CPU cores in π (i.e., task-to-core assignment Λ) such that the maximum steady-state temperature of the cores is minimized while all tasks mapped onto each core meet their deadlines under fixed-priority scheduling with MPCP.

The task-to-core assignment problem is NP-hard, because finding a feasible mapping is equivalent to the bin-packing problem which is known to be NP-hard in the strong sense [14]. Thus, we must look for heuristics. Focusing on feasibility, a typical task-to-core assignment is to apply variants of well-known bin-packing algorithms, including First-Fit Decreasing (FFD), Best-Fit Decreasing (BFD), and Worst-Fit Decreasing (WFD) [38]. These algorithms process tasks one-by-one in the order of non-increasing utilization, assigning each task to a core according to the heuristic function that determines how to break ties if multiple cores exist that can accommodate the new task. Whether a core can accommodate each task or not is determined by the schedulability test in Lemma 5.1.

Example 5.3. Let us consider a set of four vision tasks shown in Fig. 5.3 and a platform consisting of two CPU cores and a single GPU. The CPU utilizations of individual tasks, i.e., $u_i^C = e_i^C / p_i$, are $u_1^C = 0.2$, $u_2^C = 0.1$, $u_3^C = 0.05$, and $u_4^C = 0.05$. The CPU's power dissipations by individual tasks are $P_1^C = 1.8W$, $P_2^C = 1.8W$, $P_3^C = 2.0W$, and $P_4^C = 2.5W$. In this example, we consider CPU1 and CPU2 in Fig. 5.2b where CPU1 heats up faster than CPU2 because of the CPUs–GPU thermal coupling. We consider four possible task-to-core assignment algorithms as shown in Fig. 5.5: (a) FFD and BFD, (b) WFD, and (c) a thermally-optimal assignment. In FFD, each task is assigned to the first CPU on which it fits. In BFD and WFD, each task is assigned to the minimum remaining capacity exceeding its own CPU utilization and the maximum remaining capacity, respectively. After assignment, under FFD and BFD, the temperatures of CPU1 and CPU2 are increased by 22 °C and 9 °C,[6] respectively, while under WFD, the temperature increases are 17 °C and 13 °C, respectively. Although WFD results a lower maximum steady-state temperature than FFD/BFD, it is not thermally-optimal. In fact, there exists a thermally-optimal assignment, as shown in Fig. 5.5c.

Note that FFD and BFD attempt to pack as many tasks as possible onto one

[6] Note that the temperature rise of CPU2 in Fig. 5.5a is due to the indirect effect of the execution of workloads on GPU, i.e., CPUs–GPU thermal coupling.



(a) FFD and BFD

(b) WFD

(c) Thermally-optimal assignment

Figure 5.5: Task-to-core assignment algorithms and their corresponding temperature increases.

core while keeping the other cores empty to accommodate other unassigned tasks. By contrast, WFD tends to distribute the workloads evenly across all cores. In general, FFD and BFD have shown better feasibility than WFD [16]. However, they may result in higher temperatures than WFD because the workloads are allocated (heavily) to one core in many cases. Although WFD may decrease the maximum steady-state temperature by evenly distributing the workloads across all cores, it does not consider different power dissipations of tasks and CPUs–GPU thermal coupling, resulting in *thermally-unbalanced* assignments as shown in Fig. 5.5b. As shown in Fig. 5.5c, a thermally-optimal assignment in this example turns out to be the one that assigns slightly more CPU workloads to CPU2 than to CPU1. This is because CPU2 provides a more thermally-efficient operation than CPU1 does because of the CPUs–GPU thermal coupling.

Considering the thermal coupling between GPU and CPU cores, we present the concept of *thermally-balanced* assignment to gauge the heat distribution across cores in a multi-core platform.

Definition 5.4 (Thermally-balanced assignment)**.** A task-to-core assignment Λ is said to be *thermally-unbalanced* if the maximum steady-state temperature among cores can be lowered by moving one task from one core to another without losing feasibility. Otherwise, it is said to be *thermally-balanced.*

Clearly, the optimal task-to-core assignment that achieves both G1 and G2 must be thermally-balanced, since by definition, its maximum steady-state temperature among cores cannot be lowered. We now derive a sufficient condition for a task-to-core assignment to be thermally-balanced. Note that, based on the thermal coefficient values in Table 5.1, in task-to-core assignment, we assume that the difference in thermal conduction rate from one CPU to others is negligible ($R_{c1,c2} \simeq R_{c2,c1} \simeq R_{c1,c3}$ where $\forall 1 \leq c1, c2, c3 \leq m$).[7]

Lemma 5.5. *A task-to-core assignment Λ is thermally-balanced if for every pair (π_p, π_q) s.t. $\pi_p, \pi_q \in \{\pi_1, ..., \pi_m\}$ and every task $\tau_i \in p(\pi_p)$ satisfy*

$$T_{\pi_p}(\Lambda) - T_{\pi_q}(\Lambda) \leq R_p \cdot P_i^C \cdot u_i^C. \tag{5.10}$$

Proof. Suppose that a task-to-core assignment Λ satisfies Eq. (5.10). Without a loss of generality, we consider a pair (π_p,π_q) that satisfies (a) $T_{\pi_p}(\Lambda) - T_{\pi_q}(\Lambda) \leq R_p \cdot P_i^C \cdot u_i^C$ (by assumption). Consider a new assignment Λ' obtained from Λ by transferring a task τ_i from π_p to π_q. We will prove that the maximum steady-state temperature among cores cannot be lowered by moving τ_i from π_p to π_q. Two possibilities exist: i) $T_{\pi_p}(\Lambda) > T_{\pi_q}(\Lambda)$, and ii) $T_{\pi_p}(\Lambda) \leq T_{\pi_q}(\Lambda)$.

Case i): according to Eqs. (5.5) and (5.9), $T_{\pi_p}(\Lambda') = T_{\pi_p}(\Lambda) - R_p \cdot P_i^C \cdot u_i^C$ and

[7] Note that we still consider a different thermal coefficient value for other elements in Table 5.1, such as R_c, R_g, $R_{g,c}$, and $R_{c,g}$.

Algorithm V.1 T-WFD (τ, π)

```
 1: for $\pi_c \in \{\pi_1, \ldots, \pi_m\}$ do
 2:     $\Lambda_{\pi_c} \leftarrow \emptyset$
 3: end for
 4: $\tau' \leftarrow$ Sort($\tau$ by non-increasing $P_i^C \cdot u_i^C$)
 5: for $\tau_i \in \tau'$ do
 6:     $\pi' \leftarrow \{\pi_c :$ feasible-assignment$(\Lambda_{\pi_c} \cup \tau_i)\}$
 7:     if $\pi' = \emptyset$ then
 8:         return Failed to assign
 9:     end if
10:     $\pi_k \leftarrow \arg\min_{\pi_c \in \pi'} T_{\pi_c}(\Lambda_{\pi_c} \cup \tau_i)$
11:     $\Lambda_{\pi_k} \leftarrow \Lambda_{\pi_k} \cup \tau_i$
12: end for
13: return $\Lambda$
```

$T_{\pi_q}(\Lambda') = T_{\pi_q}(\Lambda) + R_q \cdot P_i^C \cdot u_i^C$. Then, we have

$$\begin{aligned} T_{\pi_q}(\Lambda') - T_{\pi_p}(\Lambda') =& T_{\pi_q}(\Lambda) - T_{\pi_p}(\Lambda) + R_p \cdot P_i^C \cdot u_i^C + R_q \cdot P_i^C \cdot u_i^C \\ \geq& - R_p \cdot P_i^C \cdot u_i^C + R_p \cdot P_i^C \cdot u_i^C + R_q \cdot P_i^C \cdot u_i^C \text{ (by (a))} \\ =& R_q \cdot P_i^C \cdot u_i^C. \end{aligned}$$

That is, (b) $T_{\pi_q}(\Lambda') - T_{\pi_p}(\Lambda') \geq R_q \cdot P_i^C \cdot u_i^C$. By (a) and (b), we have $T_{\pi_q}(\Lambda') - T_{\pi_p}(\Lambda') \geq T_{\pi_p}(\Lambda) - T_{\pi_q}(\Lambda)$ for any pair (π_p,π_q). Hence, the new assignment Λ' is unbalanced because the temperature difference between π_p and π_q only increases compared with the original assignment Λ. Therefore, returning to the original assignment Λ (by moving back τ_i to π_p) always lower the maximum steady-state temperature.

Case ii): that is, moving a task τ_i from a low temperature core to a high temperature one. The resulting assignment Λ' can easily be seen to be unbalanced and just like Case i). Therefore, we should be able to further lower the maximum steady-state temperature by returning to the original assignment Λ. □

To achieve a thermally-balanced assignment, we propose a new thermal-aware task-to-core assignment called T-WFD, as presented in Algorithm V.1. Unlike the previous algorithms presented in Example 5.3, tasks are sorted into a non-increasing order of their *average CPU power dissipations* (Line 4). T-WFD then assigns each task to the core with the *lowest temperature* on which it fits (Lines 5–12).

Note that T-WFD considers feasibility and thermal issues together in task-to-core assignment. In particular, tasks are sorted according to their average power dissipation by considering different power dissipations of tasks and effects on CPU temperature. Cores are arranged in increasing order of temperature, taking the CPUs–GPU coupling into account. Then, T-WFD allocates each task to the core with the lowest temperature on which the allocation can preserve feasibility with the schedulability test in Lemma 5.1. This way, it is possible to find a thermally-balanced assignment.

Next, we prove that T-WFD never produces a thermally-unbalanced assignment in the following theorem.

Theorem 5.6. *The T-WFD scheme always generates a thermally-balanced task-to-core assignment.*

Proof. Consider a set τ of n periodic tasks (indexed according to non-increasing average CPU power dissipations) that are to be assigned on m CPU cores. We will prove this statement by induction. Clearly, after assigning the first task τ_1 to the core with the lowest temperature upon which it fits, the assignment is balanced. Suppose that the statement holds after assigning $\tau_1, \ldots, \tau_k$ $(1 \leq k < n)$ to the cores according to T-WFD. Let us define $\Lambda(k)$ to be the assignment after allocating the k-th task. Let us also define π_c to be the core with the lowest temperature on which τ_k fits in $\Lambda(k)$.

T-WFD chooses π_c to allocate τ_{k+1}. Any pair (π_p, π_q) such that $\pi_p \neq \pi_c$ and $\pi_q \neq \pi_c$ cannot be the source of a thermally-unbalanced assignment, because their workload did not change and $\Lambda(k)$ is supposed to be balanced by the inductive hypothesis. Therefore, we need to focus only on pairs (π_c, π_p) where $1 \leq p \leq m$, and $p \neq c$. Two possible cases exist: after assignment of τ_{k+1} to π_c, i) π_c becomes the highest temperature core, and ii) otherwise.

Case i): consider a pair (π_c, π_p) such that $T_{\pi_c}(\Lambda(k+1)) > T_{\pi_p}(\Lambda(k+1))$, where $T_{\pi_p}(\Lambda(k+1))$ is the temperature of π_p in $\Lambda(k+1)$. Note that $T_{\pi_c}(\Lambda(k)) \leq T_{\pi_p}(\Lambda(k))$ in $\Lambda(k)$ by T-WFD. Thus,

$$\begin{aligned}&T_{\pi_c}(\Lambda(k)) = T_{\pi_c}(\Lambda(k+1)) - R_c \cdot P^C_{k+1} \cdot u^C_{k+1} \leq T_{\pi_p}(\Lambda(k)) \leq T_{\pi_p}(\Lambda(k+1))\\ \Leftrightarrow &T_{\pi_c}(\Lambda(k+1)) - T_{\pi_p}(\Lambda(k+1)) \leq R_c \cdot P^C_{k+1} \cdot u^C_{k+1}.\end{aligned}$$

Because of the pre-ordering of tasks according to average power dissipations, $P^C_{k+1} \cdot u^C_{k+1} \leq P^C_x \cdot u^C_x$ for any task τ_x allocated to π_c ($x \leq k+1$). Therefore, $T_{\pi_c}(\Lambda(k+1)) - T_{\pi_p}(\Lambda(k+1)) \leq R_c \cdot P^C_x \cdot u^C_x$ for any task τ_x allocated to π_c ($x \leq k+1$). Then, according to Lemma 5.5, the pair (π_c, π_p) cannot be unbalanced.

Case ii): after the assignment of τ_{k+1} to π_c, let π_q be the highest temperature core and consider a pair (π_c, π_q). The new assignment $\Lambda(k+1)$ cannot make the pair (π_c, π_q) thermally-unbalanced, because if it were, then the same pair would be thermally-unbalanced in $\Lambda(k)$ as well ($\because$ $\Lambda(k+1)$ only reduced the temperature difference between π_c and π_q). This contradicts the inductive hypothesis. □

Runtime Complexity. Alg. V.1 first sorts the tasks with $O(n \cdot logn)$ complexity, where n is the number of tasks. Then, the algorithm allocates each task to a feasible core starting from the core with the lowest temperature with $O(n \cdot m)$ complexity where m is the number of cores. Thus, the total complexity is $O(\max(n \cdot logn, n \cdot m))$.

5.5.2 CPU–GPU Co-Scheduling

Thus far, we have discussed the task assignment to handle the platform's temperature imbalance. Building on the thermally-balanced assignment, we now demonstrate how to schedule task/job executions on CPU and GPU cores to mitigate the peak temperature. Specifically, we want to address the following *schedule-generation* problem.

Definition 5.7 (schedule-generation)**.** Given the task-to-core assignment, determine a schedule of job executions and idle-times on both CPUs and the GPU such that the maximum transient temperature across the CPU and GPU cores is minimized while all the jobs of all tasks $\tau_i \in \tau$ meet their deadlines.

Addressing the Peak Temperature. According to our proposed task-to-core assignment, tasks are allocated in a thermally-balanced manner in terms of the *steady-state* temperature while feasibility is preserved under fixed-priority scheduling with MPCP [95]. However, a job schedule on CPU and GPU cores may affect the *transient* temperature, potentially leading to the chip overheating before it reaches the steady-state temperature. This situation would be caused by the following two key issues: 1) different power dissipations of tasks on the CPU and GPU cores, and 2) CPUs–GPU thermal coupling. Specifically, because of different power dissipations caused by different tasks (as shown in Fig. 5.3), the temperatures of the CPUs and GPU vary greatly depending on the tasks currently running on their cores. We observe from Eqs. (5.7) and (5.9) that (i) if $P_i^C > P_{\pi_c}(\Lambda)$ (i.e., the power dissipation during CPU execution of τ_i is greater than the average power dissipation on π_c), the temperature of π_c increases above the steady-state temperature $T_{\pi_c}(\Lambda)$, and (ii) if $P_i^C \leq P_{\pi_c}(\Lambda)$ then the temperature of π_c decreases below $T_{\pi_c}(\Lambda)$. The same holds for the GPU case. A task τ_i is said to be *hot* if $P_i^C > P_{\pi_c}(\Lambda)$ ($P_i^G > P_{\pi_g}(\Lambda)$), or *cold* otherwise. Depending on P_i^C and P_i^G, τ_i can become hot or cold on CPUs and the GPU. In addition, because of heat conduction by CPUs–GPU thermal coupling, the tasks scheduled on the GPU could affect the temperature of its neighboring CPU cores, and vice versa. For example, scheduling a *hot* task on CPU (GPU) in the presence of *hot* GPU (CPU) workloads tends to cause a rapid rise in the temperature of CPUs and the GPU together. One may slow the temperature increase by suspending the execution of a hot task and scheduling a cold task or idle-time on CPU/GPU, but such an action may also lead to a hot task's deadline being missed. This calls for cooperative scheduling of CPU and GPU computations, i.e., scheduling

Algorithm V.2 CPU-GPU co-scheduling

```
1: $Q_{CPU}$ : CPU ready queue
2: **Upon job release/completion or no remaining inversion budget:**
3: **for** $\tau_i \in Q_{CPU}$ **do**
4:     **if** $\forall \tau_h \in hpp(\tau_i)$ satisfies $v_h - e^C_{i,cur} \geq 0$ **then**
5:         Put $\tau_i$ in the candidate set $\Gamma$.
6:     **end if**
7: **end for**
8: **if** $\Gamma$ is not empty **then**
9:     $\tau_{CPU} = \min_{\tau_i \in \Gamma_{\pi_c}} |\bar{P}_{tot} - (P^C_i + \sum_{\pi \setminus \pi_c} P^C_{cur} + P^G_{cur})|$
10:    Schedule $\tau_{CPU}$
11: **else**
12:    Schedule a task with the highest priority in $Q_{CPU}$.
13: **end if**
```

CPU jobs while considering GPU schedules, and vice versa, to effectively mitigate excessive temperature rises without missing any task deadlines.

We develop a thermal-aware CPU–GPU co-scheduling mechanism that determines which tasks to run on CPUs and the GPU in a cooperative manner. Basically, at each scheduling instant, our mechanism is based upon partitioned fixed-priority scheduling and restrictively allows *priority inversions* – executing cold/idle tasks with lower-priorities ahead of a hot task with the highest-priority on CPUs when its counterpart (the GPU) is running a hot task, and vice versa – subject to schedulability constraints. Such a mechanism avoids simultaneous executions of hot tasks on both CPUs and the GPU and thus reduces the peak temperature while ensuring that all tasks still meet their deadlines. Algorithm V.2[8] presents our thermal-aware CPU–GPU co-scheduling mechanism, which consists of two steps: (i) *candidate selection* and (ii) *job selection*. Whenever a scheduling decision is to be made on CPUs and the GPU, the algorithm first constructs a list of candidate jobs that are allowed to execute without missing any others' deadline (lines 3–7) and then selects one job from the list by taking the current job on its counterpart into consideration;

[8] Algorithm V.2 describes CPU scheduling, and GPU scheduling is also performed similarly. RT-TAS uses GPU lock and priority queue to schedule GPU, and the implementation is presented in Sec. 5.6.1.

thus, the difference between the steady-state temperature and transient temperature caused by the execution of the selected and current jobs on the CPUs and the GPU is minimized (lines 8–10).

Finding Candidate Jobs. To prevent any deadline misses caused by the priority inversions, we calculate the *worst-case maximum inversion budget* V_i for each task τ_i allowing lower-priority tasks to execute while τ_i waits. V_i is calculated using the WCRT analysis shown in Lemma 5.1 with a similar approach proposed in [125]. Note that in the presence of priority inversions, τ_i can experience more interference from higher-priority tasks than when no priority inversion is allowed, which is because of the additional interference by the deferred executions (also known as back-to-back hits) [125]. To consider such deferred executions when calculating V_i, we derive a pessimistic upper-bound on the worst-case response time w_i^* using the WCRT analysis under the assumption that the worst-case busy interval of τ_i is equal to d_i, instead of the iterative increment of the busy interval of τ_i until it no longer increases. Using the pessimistic upper-bound on w_i^*, we define the worst-case maximum inversion budget V_i as

$$V_i = d_i - w_i^*. \tag{5.11}$$

We then only allow bounded priority inversions using V_i to guarantee that deadlines are met. To enforce these budgets at run-time, our mechanism maintains a *remaining inversion budget* v_i where $0 \leq v_i \leq V_i$. This indicates the time budget left for lower-priority tasks than τ_i to execute in a priority inversion mode while τ_i has an unfinished job. The budget v_i is replenished to V_i when a new job of τ_i is released. It is decreased as the CPU/GPU execution of τ_i is blocked by a lower-priority job. When the budget becomes 0, no lower-priority task is allowed to run until τ_i finishes its current job.

The scheduler is invoked upon (i) the release of a new job, (ii) the completion of a job, or (iii) when no inversion budget of a job remains. Upon each invocation, our scheduling algorithm finds candidate jobs that are allowed to execute on CPU/GPU based on the following lemmas. For each task τ_i in the CPU run queue, we let $e^C_{i,cur}$ denote the remaining CPU section execution time at time t_{cur}.

Lemma 5.8. *For a task τ_i, if $\forall \tau_h \in hpp(\tau_i)$ satisfies $v_h - e^C_{i,cur} \geq 0$ or τ_i is the highest-priority task, τ_i can be a candidate for CPU execution without missing any deadlines of higher-priority tasks $hpp(\tau_i)$.*

Proof. A busy interval of $\tau_h \in hpp(\tau_i)$ is composed of its CPU and GPU executions, the preemption delay of CPU execution by $hpp(\tau_h)$, the blocking time to acquire a GPU access, and a further delay of CPU execution caused by priority inversions using our co-scheduling policy. Suppose that at time t_{cur}, our co-scheduling policy decides to execute τ_i, which is a lower priority than τ_h at time t. Then, the worst-case busy interval of τ_h is bounded by

$$e^C_h + e^G_h + I_h + B_h + e^C_{i,cur} \leq w^*_h + e^C_{i,cur} \leq w^*_h + v_h = d_h,$$

because $v_h - e^C_{i,cur} \geq 0$. The execution of the remaining CPU section of τ_i will not miss the deadline of $\tau_h \in hpp(\tau_i)$. Thus, τ_i can be a candidate for CPU execution at time t_{cur}. □

For each task τ_i in the GPU run queue, let $e^G_{i,cur}$ denote the remaining GPU section execution time at time t_{cur}.

Lemma 5.9. *If $\forall \tau_h \in hp(\tau_i)$ satisfies $v_h - e^G_{i,cur} \geq 0$ or τ_i is the highest-priority task, τ_i is a candidate for GPU execution.*

Proof. This lemma can be proved in a similar manner to Lemma 5.8. □

Note that we also include an idle CPU task, which is a special cold task that allows an idling CPU during hot task execution on the GPU.

Select a Job among Candidates. For CPU scheduling, we select one job to execute on π_c from the candidate set Γ_{π_c} by considering the current job on its counterpart (GPU). The total average power dissipation $\bar{P}_{tot}$ for the entire task set is calculated as follows:

$$\bar{P}_{tot} = \sum_i (P_i^C \cdot \frac{e_i^C}{p_i} + P_i^G \cdot \frac{e_i^G}{p_i}). \tag{5.12}$$

Let P_{cur}^C and P_{cur}^G denote the power dissipation by the current running job on the CPUs and GPU, respectively. Then, we pick a task τ_s in Γ_{π_c} such that the difference between the total average power dissipation $\bar{P}_{tot}$ and the expected power dissipation $(P_s^C + \sum_{\pi \setminus \pi_c P_{cur}^C} + P_{cur}^G)$ by the selected and currently running jobs on the CPUs and GPU is minimized; that is,

$$\min_{\tau_s \in \Gamma_{\pi_c}} |\bar{P}_{tot} - (P_s^C + \sum_{\pi \setminus \pi_c} P_{cur}^C + P_{cur}^G)|. \tag{5.13}$$

This way, we co-schedule CPU and GPU cores such that the total transient power dissipation on them maintains the total average power dissipation ($\bar{P}_{tot}$) for a task set as close as possible, thereby reducing the transient temperature. Such a job selection process can be performed similarly for GPU scheduling. We pick a task τ_s in Γ_{π_g} such that

$$\min_{\tau_s \in \Gamma_{\pi_g}} |\bar{P}_{tot} - (\sum_{\pi_c} P_{cur}^C + P_s^G)|. \tag{5.14}$$

With the proposed CPU–GPU co-scheduling algorithm, we can reduce the variation in steady-state and transient temperatures, thus effectively mitigating any excessive rise in transient temperature while guaranteeing all deadlines are met.

Runtime Complexity. Upon each invocation (either job release/completion or no remaining inversion budget), our scheduling algorithm checks/updates the remaining inversion budget and finds candidate jobs using Alg. V.2 with $O(n)$ complexity, where n is the number of tasks. Then, our algorithm selects a task based on

Eqs. (5.13)–(5.14) with $O(1)$ complexity. Thus, the total complexity is $O(n)$.

5.6 Evaluation

We implemented and evaluated RT-TAS on a representative CPUs–GPU platform with automotive vision workloads, and the key results as follows:

- Maximum temperature was reduced by $6°C$ and $12.2°C$ w.r.t. the state of the art [107] and a default OS scheduler, respectively.
- Our thermally-balanced assignment reduced the maximum temperature by $3.9°C$ and CPU–GPU co-scheduling reduced it further by $2.1°C$ w.r.t. the state-of-the-art.
- Maximum temperature is reduced by up to $8.3°C$ ($5.0°C$ on average) across various task sets w.r.t. WFD (Fig. 5.5b).

5.6.1 Methodology

Our experimental platform was an Nvidia Tegra X1 equipped with four CPU cores and a shared GPU [9] rated at the maximum power dissipation of $15W$. To avoid chip overheating, each CPU/GPU was equipped with a thermal sensor. The built-in hardware temperature management kicks in when one of its cores reaches the temperature threshold, and it lowers the CPU frequency to cool the temperature. According to the thermal specifications in [9], chip thermal resistance is $1.15°C/W$. To evaluate the benefit of RT-TAS under a realistic setup, we implemented a real-time vision system running representative vision workloads [100]: (i) a feature detector, (ii) an object tracker, (iii) a motion estimator, and (iv) an image stabilizer. An in-vehicle camera video was provided to these tasks for input. Specifically, we implemented RT-TAS on top of the Linux kernel as a user-level application that

Table 5.2: Vision tasks used in the experiments.

Task	P_i^C (W)	P_i^G (W)	e_i^C (ms)	e_i^G (ms)	p_i(ms)
Feature detector	1.8	3.7	14	25	400
Object tracker	1.8	2.8	34	17	400
Motion estimator	2.0	5.7	63	105	400
Video stabilizer	2.5	3.6	35	65	400

executes a set of tasks each running one of the above vision workloads periodically. The implementation details are summarized as follows:

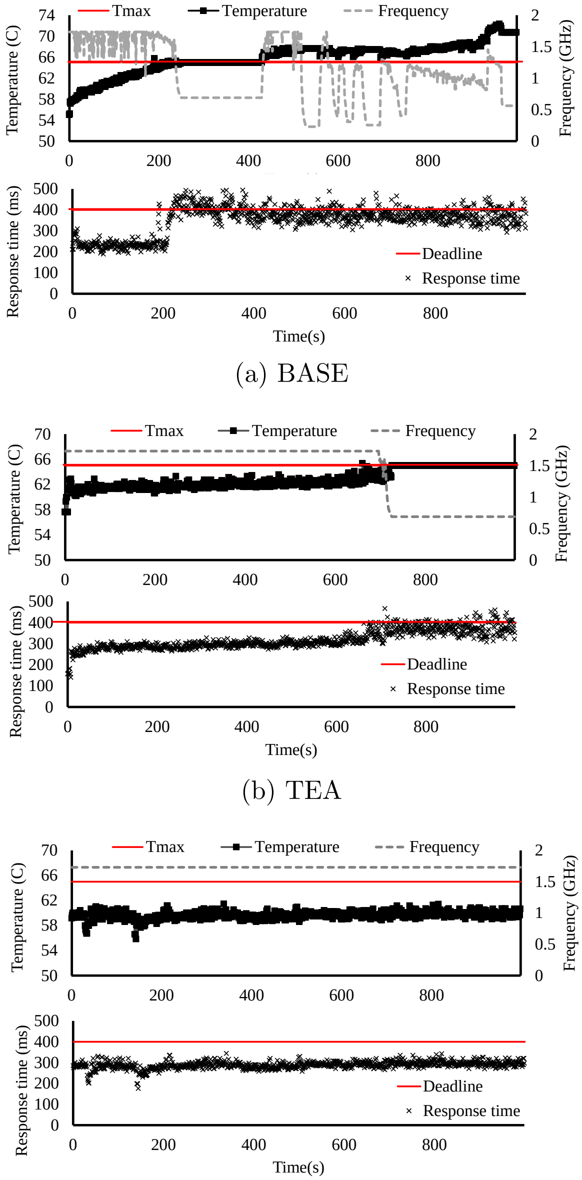
- Assigning tasks to CPU cores using `sched_setaffinity` and `CPU_SET`;
- Priority-based scheduling using `sched_setscheduler` under the `SCHED_FIFO`; and
- GPU lock is implemented using `pthread_mutex`, and the highest priority task waiting for the lock will grab the lock.

Throughout the evaluation, we compared the following approaches:

- BASE: default OS scheduler (completely fair scheduling);
- TEA: thermally-efficient allocation [107] assigning tasks from the most thermally-efficient core first[9];
- RT-TAS: the proposed thermally-balanced assignment (§5.5.1) and CPU–GPU co-scheduling (§5.5.2).

To avoid external influences, the external fan was turned off. Unless otherwise specified, the temperature threshold was set to 65°C, and the CPU and GPU cores were running at the maximum frequency. During our experiments, the WCET of each job was recorded to check whether the job deadlines were met. The CPU/GPU execution time, power dissipation, and period of tasks are provided in Table 5.2.

[9] TEA identifies thermally-efficient cores depending on the CPU power dissipation offline and then sequentially bind the task with the highest CPU usage to the next most thermally-efficient core.



(a) BASE

(b) TEA

(c) RT-TAS

Figure 5.6: RT-TAS avoids thermal throttling by reducing maximum temperature, thus achieving a reliable response time.

5.6.2 Effectiveness at Reducing Temperature

We first demonstrated RT-TAS's effective reduction of the maximum chip temperature, thus achieving reliable performance of a real-time vision system. Fig. 5.6 plots the maximum transient temperature among cores, CPU frequency, and task response time for different schemes. With BASE (Fig. 5.6a), the temperature

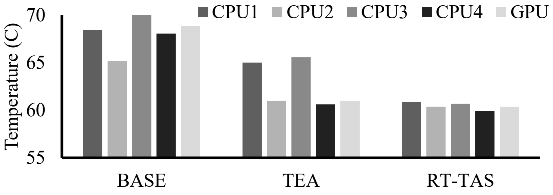


Figure 5.7: Core-level peak temperature under different scheduling policies.

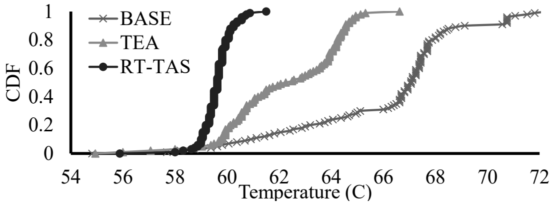


Figure 5.8: Maximum temperature CDF under different scheduling policies.

exceeded the threshold at approximately 200s and hardware thermal throttling was triggered to reduce the processor frequency from 1.7GHz to 0.5GHz. As a result, the maximum response time increased from 309ms to 493ms, violating the deadline (400ms). However, the chip temperature increased further, reaching up to 73°C. With TEA (Fig. 5.6b), the maximum temperature increased less rapidly than with BASE, but the temperature exceeded the threshold at approximately 800s, and experienced thermal throttling thereafter. With RT-TAS (Fig. 5.6c), the temperature remained below the threshold maintaining the maximum CPU frequency and reliable response time. RT-TAS achieves this by addressing the temperature imbalance on its underlying platform.

Fig. 5.7 compares the CPUs' and GPU's peak temperatures, demonstrating how RT-TAS mitigated the temperature imbalance. With BASE, tasks were assigned without considering the temperature imbalance, which led to the max-min temperature difference of 7.5°C. Consequently, the CPU temperatures increased unevenly, causing the highest maximum temperature of 72.9 °C. With TEA, tasks were assigned to the thermally-efficient core first, distributing workloads more effectively across CPU cores than with BASE. However, tasks were assigned to hot CPU cores (i.e., CPU1, CPU3), resulting in the maximum temperature of 66.7°C with a max-min temperature difference of 5.7°C. RT-TAS assigned tasks to the core in a thermally-balanced manner by capturing the different core-level GPU heating impact and power variations of tasks. Therefore, RT-TAS reduced the max-min difference to 1°C, and the maximum temperature to 60.7°C. Fig. 5.8 compares the maximum

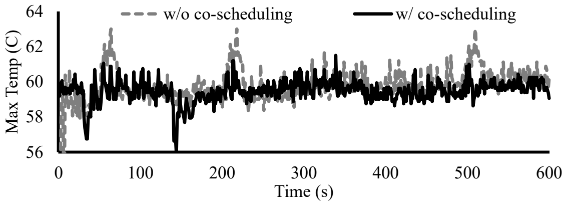
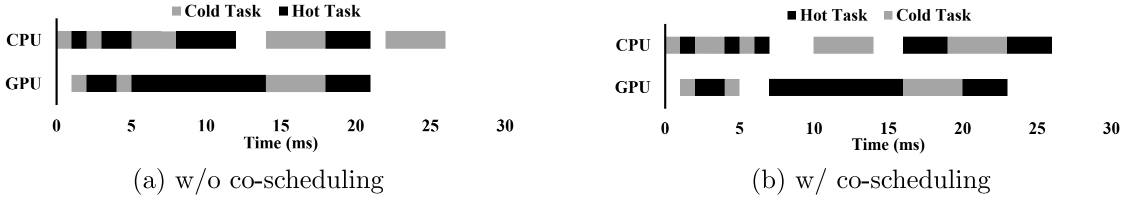


Figure 5.9: Transient temperatures w/o and w/ CPU–GPU co-scheduling.



(a) w/o co-scheduling

(b) w/ co-scheduling

Figure 5.10: Job schedule (a) w/o and (b) w/ co-scheduling.

temperature dynamics over time for different schemes. RT-TAS could reduce the peak temperature (at the 100% percentile) by up to 6°C and 12.2°C compared with TEA and BASE, respectively.

Next, we analyzed the impact of co-scheduling by running the same experiment without and with co-scheduling. Fig. 5.9 compares the transient temperature variations over time without and with co-scheduling. The peak temperature under CPU–GPU co-scheduling was 61.5°C at 217 seconds, whereas that without co-scheduling was 63.6°C. Fig. 5.10 shows the actual schedule of jobs in a specific time interval without and with co-scheduling. Without co-scheduling, tasks were scheduled in a fixed-priority order on the CPUs and GPU independently (Fig. 5.10a). In such a case, the CPUs and GPU may simultaneously perform peak computations (running hot tasks), and thus incur a peak total power dissipation. With co-scheduling the CPUs and GPU together, RT-TAS avoided the overlap of simultaneous peak computations on the CPUs and GPU to reduce the peak power dissipation (Fig. 5.10b). As a result, co-scheduling effectively mitigated any excessive rise in transient temperature by avoiding bursts of peak power dissipation. We also analyzed the effectiveness of task assignment and co-scheduling at reducing the peak

Table 5.3: Task-set generation parameters.

Number of CPUs (M)	4
Number of tasks (n)	8
Maximum number of GPU sections (η_i)	2
Maximum CPU execution time (e_i^C)	100ms
Maximum CPU power dissipation (P_i^C)	2.5W
Maximum GPU execution time (e_i^G)	100ms
Maximum GPU power dissipation (P_i^G)	6W
Utilization per CPU ($\sum_{\tau_i} u_i/M$)	0.3

temperature, respectively. Overall, thermally-balanced assignment and CPU–GPU co-scheduling reduced the maximum temperature by 3.9°C and 2.1°C, respectively, resulting in a total temperature reduction of 6°C compared with TEA. This 6°C reduction in maximum temperature translates to a 1.52× longer chip lifetime[10] [118] and cooling cost savings of US$ 15.6 per chip[11] [9, 114].

5.6.3 Evaluation with Different Task Sets

Next, we evaluated RT-TAS with different task sets using the parameters measured through the experiments. The base parameters in Table 5.3 were acquired from the above mentioned platform and sample vision tasks. We randomly generated 1,000 task sets, and task execution parameters $(\eta_i, e_i^C, e_i^G, P_i^G)$ were set to be uniformly distributed within their maximum bounds. Next, the task utilization was determined according to the UUniFast algorithm [21], and the task period was set to $p_i = (e_i^C + e_i^G)/u_i$. We also used the identified platform thermal parameters in Table 5.1 and the thermal model in Eq. (5.6) to estimate the chip temperature. We compared RT-TAS against two baseline algorithms, FFD and WFD, in Sec. 5.5.1.

[10] Chip lifetime is typically estimated by mean-time-to-failure $MTTF \propto mean(\frac{kT(t)}{exp(-E_a/k \cdot T(t))})$ where k, E_a are the Boltzmann and activation energy constant [118]. We evaluate MTTF using the above equation and temperature traces.

[11] Cooling power and chip temperature is modeled by $P_{cooling} = \frac{\Delta T_{chip}}{R_{chip}}$ where $P_{cooling}$ is the heat extracted, R_{chip} is the chip thermal resistance, ΔT_{chip} is the temperature reduction. To reduce 6°C for the chip with the thermal resistance of 1.15°C/W, the cooling solution needs to extract $\frac{6}{1.15} = 5.2W$ of thermal dissipation. The cooling cost is estimated by 3$/$W$ [114] and the saving is $5.2 \times 3 = 15.6$$.

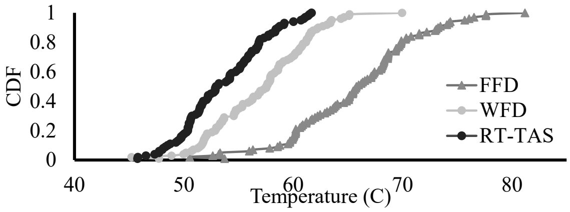
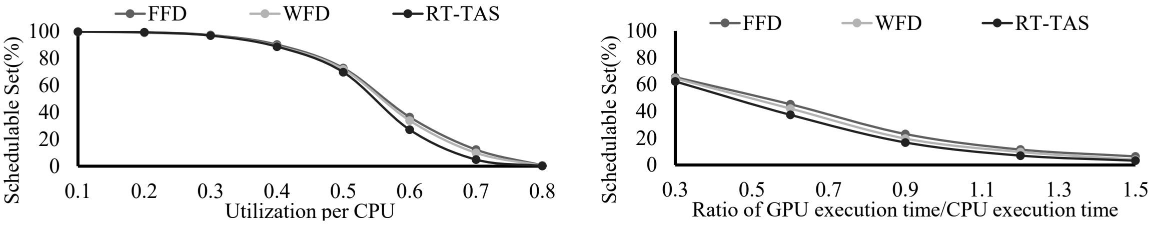


Figure 5.11: Maximum temperature CDF for different task sets.

Fig. 5.11 plots the maximum temperature dynamics for different task sets with base parameters. The maximum temperature reduction by RT-TAS was up to 8.3°C and 5.0°C on average. From the base task set parameters, we varied i) utilization ii) GPU execution time, and iii) task-level power variation for each experiment setting. We highlight the following three observations: the maximum temperature reduction by RT-TAS became more pronounced for i) lower overall utilization, ii) higher maximum GPU execution time, and iii) larger variation of task-level power dissipations. The temperature decrease by RT-TAS diminished as the utilization increased, because a task assignment can no longer avoid assigning tasks to hot CPU cores. As the utilization per CPU increased from 0.3 to 0.6, the maximum temperature reduction by RT-TAS decreased from 5.0°C to 2.1°C on average. As the GPU execution time increased, the temperature reduction by RT-TAS became more pronounced because of the increasing temperature imbalance across CPU cores and the overlap between GPU and CPU executions. When the ratio of GPU execution time to CPU execution time increased from 0.1 to 1, the maximum temperature reduction by RT-TAS increased from 1.1°C to 4.6°C on average.

When the variation in task-level power dissipation was large, the temperature decrease by RT-TAS became more pronounced. Furthermore, when the maximum difference between task-level power dissipations increased from $10W$ to $15W$, maximum temperature reduction by RT-TAS increases from 5.6°C to 11.5°C on average. Such an improvement can be interpreted as the benefit of taking different task-level power dissipations and the platform's temperature imbalance into account



(a) Varied utilization

(b) Varied GPU/GPU execution time

Figure 5.12: Schedulability for various utilizations and GPU execution times.

in task-to-core assignment and scheduling.

We finally discuss RT-TAS's impact on schedulability. Fig. 5.12 plots the schedulability as a percentage of the schedulable task sets out of 1,000 task sets with varied (a) utilization per CPU, and (b) ratio of GPU execution time to CPU execution time. While FFD generated the largest number of feasible assignments across different task set configurations, T-WFD could achieve schedulability comparable to FFD. Across various configurations, T-WFD yielded only 3.6% less schedulable sets than FFD on average.

5.7 Conclusion

Embedded real-time systems running on integrated CPUs–GPU platforms should consider CPUs–GPU thermal coupling and the different CPU and GPU power dissipations of tasks when making their scheduling decisions. To address this problem, we developed RT-TAS, a new thermal-aware scheduling framework, by proposing a *thermally-balanced* task assignment algorithm. We considered the platform-level temperature imbalance and a *CPU–GPU co-scheduling* policy to prevent CPUs and the GPU from generating large amounts of heat simultaneously while meeting all timing constraints. Our evaluation on a typical embedded platform with automotive vision workloads demonstrated the effectiveness of RT-TAS in reducing the maximum chip temperature, thus improving reliability and saving cooling costs.

In this chapter, we considered partitioned fixed-priority scheduling with the MPCP protocol as a baseline. In the future, we would like to extend our task-to-core assignment and CPU–GPU co-scheduling to other baseline scheduling algorithms and GPU access protocols, and identify which scheduling algorithm with which GPU access protocol is effective in thermal-aware task scheduling on integrated CPUs–GPU platforms. We also plan to extend the proposed approach to multi-GPU platforms.

CHAPTER VI

Conclusion and Future Directions

Emerging embedded systems, such as wearable/IoT devices and connected cars, pose new challenges in meeting both their thermal and QoS requirements. Thermal- and QoS-aware embedded systems must cope with changing ambient temperature, platform thermal characteristics, and dynamic application workloads. We have developed four novel embedded battery and thermal management systems that are adaptive, effective, and practical to meet these challenges. We now summarize the contributions, limitations, and lessons learned as well as discuss future research directions.

6.1 Contributions, Limitations, and Lessons Learned

Contributions. This thesis covers the intersection between cyber system dynamics and battery/thermal dynamics. By capturing such multi-dynamics in environment, platform, and application levels, the proposed systems adapt to changing battery/thermal conditions in real time. Building on the analysis of underlying hardware and applications, they are shown to be effective in satisfying applications' QoS and battery/thermal requirements. Finally, the thesis has shown the proposed systems to be practical by implementing and demonstrating them on real-world smartphones or industrial embedded controllers used in cars.

In Chapter II, we developed BPM to address unpredictable shutoffs for smartphones in cold environments based on battery temperature characteristics. In Chapter III, we developed eTEC to mitigate system and battery overheating for mobile devices by jointly optimizing cooling power and chip heat dissipation. In Chapter IV and V, we developed a thermal-aware scheduler specifically designed for embedded real-time systems. RT-TRM defines the dynamic thermal budget as a function of changing ambient temperature and adaptively allocates thermal budget to individual tasks. Furthermore, RT-TAS identifies the thermal imbalance in CPUs–GPU SoCs and efficiently schedules CPU–GPU power dissipations to minimize the maximum chip temperature while meeting application timing constraints. All of these four proposed systems were implemented and evaluated on mobile/automotive platforms to demonstrate their adaptivity, effectiveness, and practicality.

Limitations. Although our thermal management systems provide adaptivity, effectiveness, and practicality, they have three limitations as highlighted next. First, hardware cooling solutions remain as the primary approach to removing heat from electronic devices. Our thermal management systems cannot substitute the adequate hardware cooling design; they rather supplement such hardware cooling design with various runtime methods to provide adaptivity and effectiveness. Second, advanced thermal management algorithms often incur high power dissipation, defeating the purpose of thermal management. This limits the complexity and runtime overhead of our thermal management algorithms. Third, our thermal management relies on offline characterization to compensate for hardware heterogeneity among different platforms, which limits the usability of the proposed frameworks for various devices/platforms.

Lessons Learned. As physical limits become the norm for modern computer systems, computer scientists must understand the underlying physical dynamics as well as the cyber systems. We have learned that thermal and battery engineers often oversimplify cyber systems behavior; thus, computer scientists must provide unique insights on dynamic application behavior, workload characterization, and user-level QoS. Finally, we must stress the importance of the design of experiment (DoE) because physical behavior is often hard to replicate and takes longer to experiment with than cyber systems.

6.2 Future Directions

Battery and thermal management will continue to present unique challenges to future cyber-physical systems and wearable/IoT devices, mainly because of advancing embedded artificial intelligence applications and increasing hardware heterogeneity. A large and diverse set of underlying architectures and autonomous systems therein will dominate the cyber-physical systems of the future, creating new power and thermal challenges on constrained embedded platforms. We must rethink power and thermal management strategies to deploy state-of-the-art machine learning systems on embedded platforms. Here, we discuss future research directions that can build on this thesis at both the hardware- (i.e., energy storage and computation platform) and application-level.

Software-Defined Batteries for Cyber-Physical Systems/IoTs. A major challenge related to the mass proliferation of IoT devices is that of various types of energy storage or batteries. Users are increasingly frustrated with unexpected device shutoffs, battery capacity fading, and even explosions. Traditional system power/thermal management mechanisms assume batteries to be the ideal power source, thus failing to understand and control dynamic battery power/thermal behaviors. We should explore new system-level interfaces between batteries, systems, and users that abstract complex battery dynamics for systems and users perspectives [61]. Software-defined batteries will improve the state of the art by offering users a more intuitive interface to monitor and/or control their battery health, potential thermal emergency, and power supply capability [58]. Software-defined batteries can be part of a more general direction toward heterogeneous power sources. We should extend hybrid energy storage systems (HESS) [73] as a new way of designing power delivery systems for sensors and IoTs. Additionally, we can explore the system-level implication and interfaces of HESS that will empower future cyber-physical systems.

Machine Learning on Constrained Embedded Platforms. Because of the constrained nature of many embedded platforms in practice, state-of-the-art machine learning algorithms often cannot run on embedded platforms. We need to optimize machine learning algorithms to be run on constrained platforms. In particular, we must address the various challenges of power/thermal constraints and real-time support for machine learning systems running on heterogeneous platforms. Such systems can be built on highly heterogeneous platforms consisting of GPUs, FPGAs, ASICs, and traditional CPUs. We will need to establish a methodology for distributing machine learning tasks on heterogeneous computing units in both the OS and compiler-level while meeting both thermal and real-time requirements.

Application Context-Aware Power Management. Various application-level contexts in many cyber-physical systems and IoT applications present unique opportunities for efficient resource management. We often observe system workloads and timing/power behaviors significantly changing in different application-level contexts. For example, depending on the distance to the car ahead, the processing times of adaptive cruise control (ACC) and active vehicle steering (AVS) vary significantly. In vision-based object detection systems, different driving contexts (e.g., a highway or urban area) generate different amounts of workload and power consumption behaviors for vision tasks. We need to investigate context-aware resource/power management mechanisms that adapt to different application-level contexts and QoS requirements.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Youngmoon Lee, Liang He, Eugene Kim, and Kang G Shin. Causes and fixes of unexpected mobile device shutoffs. In *arXiv preprint*, 2019.

[2] Youngmoon Lee, Eugene Kim, and Kang G. Shin. Efficient thermoelectric cooling for mobile devices. In *ISLPED*, 2017.

[3] Youngmoon Lee, Hoon Sung Chwa, Kang G Shin, and Shige Wang. Thermal-aware resource management for embedded real-time systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2857–2868, 2018.

[4] Youngmoon Lee, Hoon Sung Chwa, and Kang G Shin. Thermal-aware scheduling for integrated cpus-gpu platforms. *ACM Transactions on Embedded Computing Systems*, 18(6):90, 2019.

[5] Antutu Mobile Benchmark. `http://web.archive.org/web/20190915034122/https://play.google.com/store/apps/details?id=com.antutu.ABenchMark`.

[6] iPhone Performance and Battery Age. `https://web.archive.org/web/20190925175455/https://www.geekbench.com/blog/2017/12/iphone-performance-and-battery-age/`.

[7] Peltier Module CP60. `https://web.archive.org/web/20190615193958/https://www.cui.com/product/resource/cp60.pdf`.

[8] UI/Application Exerciser Monkey. `https://web.archive.org/web/20191010145634/https://developer.android.com/studio/test/monkey`.

[9] Tegra X1 Thermal Design Guide. Technical Report TDG-08214-001, Nvidia, 2018.

[10] Rehan Ahmed, Pengcheng Huang, Max Millen, and Lothar Thiele. On the design and application of thermal isolation servers. *ACM Transactions on Embedded Computing Systems*, 16, 2017.

[11] Rehan Ahmed, Parameswaran Ramanathan, and Kewal K Saluja. On thermal utilization of periodic task sets in uni-processor systems. In *RTCSA*, 2013.

[12] Rehan Ahmed, Parameswaran Ramanathan, and Kewal K Saluja. Necessary and sufficient conditions for thermal schedulability of periodic real-time tasks. In *ECRTS*, 2014.

[13] Rehan Ahmed, Parameswaran Ramanathan, and Kewal K Saluja. Temperature minimization using power redistribution in embedded systems. In *VLSI Design*, 2014.

[14] Tarek A AlEnawy and Hakan Aydin. Energy-aware task allocation for rate monotonic scheduling. In *RTAS*, 2005.

[15] Hakan Aydin, Rami Melhem, Daniel Mossé, and Pedro Mejía-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 53(5):584–600, 2004.

[16] Hakan Aydin and Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Parallel and Distributed Processing Symposium*, 2003.

[17] Anirudh Badam, Ranveer Chandra, Jon Dutra, Anthony Ferrese, Steve Hodges, Pan Hu, Julia Meinershagen, Thomas Moscibroda, Bodhi Priyantha, and Evangelia Skiani. Software defined batteries. In *SOSP*, 2015.

[18] Peter Bailis, Vijay Janapa Reddi, Sanjay Gandhi, David Brooks, and Margo Seltzer. Dimetrodon: processor-level preventive thermal management via idle cycle injection. In *DAC*, 2011.

[19] Luca Benini, Giuliano Castelli, Alberto Macii, Enrico Macii, Massimo Poncino, and Riccardo Scarsi. Discrete-time battery models for system-level low-power design. *IEEE Transactions on Very Large Scale Integration Systems*, 9(5):630–640, 2001.

[20] Theodore L Bergman. *Introduction to heat transfer*. John Wiley & Sons, 2011.

[21] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[22] Christoph Birkl, Euan McTurk, Matthew Roberts, Peter Bruce, and David Howey. A parametric open circuit voltage model for lithium ion batteries. *Journal of The Electrochemical Society*, 162(12):A2271–A2280, 2015.

[23] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *RTSS*, 2003.

[24] Duc Hoang Bui, Yunxin Liu, Hyosu Kim, Insik Shin, and Feng Zhao. Rethinking energy-performance trade-off in mobile web page loading. In *MobiCom*, 2015.

[25] Thidapat Chantem, X. Sharon Hu, and Robert P. Dick. Online Work Maximization Under a Peak Temperature Constraint. In *ISPLED*, 2009.

[26] Thidapat Chantem, X Sharon Hu, and Robert P Dick. Temperature-aware scheduling and assignment for hard real-time applications on MPSoCs. *IEEE Transactions on Very Large Scale Integration Systems*, 2011.

[27] Thidapat Chantem, Y Xiang, Xs Hu, and Robert Dick. Enhancing multicore reliability through wear compensation in online assignment and scheduling. In *DATE*, 2013.

[28] Thidapat Chantem, Yun Xiang, X Sharon Hu, and Robert P Dick. Enhancing multicore reliability through wear compensation in online assignment and scheduling. In *DATE*, 2013.

[29] Pedro Chaparro, José González, Qiong Cai, and Greg Chrysler. Dynamic Thermal Management using Thin-Film Thermoelectric Cooling. In *ISPLED*, 2009.

[30] Jian-Jia Chen, Chia-Mei Hung, and Tei-Wei Kuo. On the minimization fo the instantaneous temperature for periodic real-time tasks. In *RTAS*, 2007.

[31] Jian-Jia Chen, Shengquan Wang, and Lothar Thiele. Proactive speed scheduling for real-time tasks under thermal constraints. In *RTAS*, 2009.

[32] Carla-Fabiana Chiasserini and Ramesh R Rao. Pulsed battery discharge in communication devices. In *MobiCom*, 1999.

[33] Minki Cho, William Song, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Thermal system identification (TSI): A methodology for post-silicon characterization and prediction of the transient thermal field in multicore chips. In *SEMI-THERM*, 2012.

[34] Seunghyuk Choi, Thalmayr, Florian, Dominik Wee, and Florian Weig. Advanced driver-assistance systems: Challenges and opportunities ahead. `https://www.mckinsey.com/industries/semiconductors/our-insights/advanced-driver-assistance-systems-challenges-and-opportunities-ahead`, Feb 2016.

[35] Yohan Chon, GwangMin Lee, Rhan Ha, and Hojung Cha. Crowdsensing-based smartphone use guide for battery life extension. In *UbiComp*, 2016.

[36] Ihtesham Chowdhury, Ravi Prasher, Kelly Lofgreen, Gregory Chrysler, Sridhar Narasimhan, Ravi Mahajan, David Koester, Randall Alley, and Rama Venkatasubramanian. On-chip cooling by superlattice-based thin-film thermoelectrics. *Nature nanotechnology*, 4(4):235–238, 2009.

[37] Hoon Sung Chwa, Kang G. Shin, Hyeongboo Baek, and Jinkyu Lee. Physical-state-aware dynamic slack management for mixed-criticality systems. In *RTAS*, 2018.

[38] Edward G Coffman, Gabor Galambos, Silvano Martello, and Daniele Vigo. Bin packing approximation algorithms: Combinatorial analysis. In *Handbook of combinatorial optimization*, 151–207, 1999.

[39] Tommaso Cucinotta, Luigi Palopoli, Luca Abeni, Dario Faggioli, and Giuseppe Lipari. On the integration of application level and resource level QoS control for real-time applications. *IEEE Transactions on Industrial Informatics*, 6(4):479–491, 2010.

[40] Anup Das, Matthew J Walker, Andreas Hansson, Bashir M Al-Hashimi, and Geoff V Merrett. Hardware-software interaction for run-time power optimization: A case study of embedded linux on multicore smartphones. In *ISLPED*, 2015.

[41] David Defour and Eric Petit. Gpuburn: A system to test and mitigate gpu hardware failures. In *SAMOS*, 2013.

[42] Kapil Dev and Sherief Reda. Scheduling challenges and opportunities in integrated cpu+ gpu processors. In *ESTIMedia*, 2016.

[43] Kapil Dev, Xin Zhan, and Sherief Reda. Scheduling on CPU+GPU Processors Under Dynamic Conditions. *Journal of Low Power Electronics*, 13(4):551–568, 2017.

[44] Mohammad Javad Dousti, Majid Ghasemi-gol, Mahdi Nazemi, and Massoud Pedram. ThermTap : An Online Power Analyzer and Thermal Simulator for Android Devices. In *ISPLED*, 2015.

[45] Mohammad Javad Dousti and Massoud Pedram. Platform-dependent, leakage-aware control of the driving current of embedded thermoelectric coolers. In *ISPLED*, 2013.

[46] Mohammad Javad Dousti and Massoud Pedram. Power-Aware Deployment and Control of Forced-Convection and Thermoelectric Coolers. In *DAC*, 2014.

[47] Christof Ebert and John Favaro. Automotive software. *IEEE Software*, 34:33–39, May 2017.

[48] Glenn A Elliott, Bryan C Ward, and James H Anderson. GPUSync: A framework for real-time GPU management. In *RTSS*, 2013.

[49] Ericsson. Ericsson Mobility Report. `http://web.archive.org/web/20190904133641/https://www.ericsson.com/assets/local/mobility-report/documents/2015/ericsson-mobility-report-june-2015.pdf`, June 2016.

[50] Matteo Ferroni, Andrea Cazzola, Domenico Matteo, Alessandro Antonio Nacci, Donatella Sciuto, and Marco Domenico Santambrogio. Mpower: gain back your android battery life! In *UbiComp*, 2013.

[51] Nathan Fisher, Jian-Jia Jia Chen, Shengquan Wang, and Lothar Thiele. Thermal-Aware Global Real-Time Scheduling on Multicore Systems. In *RTAS*, 2009.

[52] Freescale. iMX 6Dual/6Quad Power Consumption Measurement: Table 1. VDDARM, VDDSOC, VDDPU Voltage Levels. `http://web.archive.org/web/20190826131445/https://cache.freescale.com/files/32bit/doc/app_note/AN4509.pdf`.

[53] Freescale. Technical Data Applications Processors: Table 5. FCPBGA Package Thermal Resistance. `http://web.archive.org/web/20190827023750/https://www.nxp.com/docs/en/data-sheet/IMX6DQIEC.pdf`.

[54] Yong Fu, Nicholas Kottenstette, Yingming Chen, Chenyang Lu, Xenofon D. Koutsoukos, and Hongan Wang. Feedback Thermal Control for Real-time Systems. In *RTAS*, 2010.

[55] Yong Fu, Nicholas Kottenstette, Chenyang Lu, and Xenofon D Koutsoukos. Feedback thermal control of real-time systems on multicore processors. In *EMSOFT*, 2012.

[56] Paolo Gai, Marco Di Natale, Giuseppe Lipari, Alberto Ferrari, Claudio Gabellini, and Paolo Marceca. A comparison of mpcp and msrp when sharing resources in the janus multiple-processor on a chip platform. In *RTAS*, 2003.

[57] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. MiBench : A free , commercially representative embedded benchmark suite. In *IISWC*, 2001.

[58] Liang He, Dongyao Chen, Youngmoon Lee, Yuanchao Shu, and Kang G Shin. Authenticating Drivers Using Automotive Batteries. In *arXiv preprint*, 2019.

[59] Liang He, Eugene Kim, and Kang G Shin. *-aware charging of lithium-ion battery cells. In *ICCPS*, 2016.

[60] Liang He, Eugene Kim, Kang G Shin, Guozhu Meng, and Tian He. Battery state-of-health estimation for mobile devices. In *ICCPS*, 2017.

[61] Liang He, Youngmoon Lee, Eugene Kim, and Kang G Shin. Mobile device batteries as thermometers. In *arXiv preprint*, 2019.

[62] Liang He, Youngmoon Lee, Eugene Kim, and Kang G Shin. Environment-aware estimation of battery state-of-charge for mobile devices. In *ICCPS*, 2019.

[63] Liang He, Guozhu Meng, Yu Gu, Cong Liu, Jun Sun, Ting Zhu, Yang Liu, and Kang G Shin. Battery-aware mobile data service. *IEEE Transactions on Mobile Computing*, 16(6):1544–1558, 2017.

[64] Liang He, Yu-Chih Tung, and Kang G Shin. icharge: User-interactive charging of mobile devices. In *MobiSys*, 2017.

[65] Mohammad A Hoque and Sasu Tarkoma. Characterizing smartphone power management in the wild. In *UbiComp*, 2016.

[66] Mohammad A Hoque and Sasu Tarkoma. Sudden drop in the battery level?: understanding smartphone state of charge anomaly. *ACM SIGOPS Operating Systems Review*, 49(2):70–74, 2016.

[67] Mohammad Ashraful Hoque, Matti Siekkinen, Jonghoe Koo, and Sasu Tarkoma. Full charge capacity and charging diagnosis of smartphone batteries. *IEEE Transactions on Mobile Computing*, 16(11):3042–3055, 2017.

[68] Huang Huang, Gang Quan, Jeffery Fan, and Meikang Qiu. Throughput Maximization for Periodic Real-Time Systems Under the Maximal Temperature Constraint. In *DAC*, 2011.

[69] Sriram Jayakumar. Making Sense of Thermoelectrics for Processor Thermal Management and Energy Harvesting. In *ISLPED*, 2015.

[70] Ramkumar Jayaseelan and Tulika Mitra. Temperature aware task sequencing and voltage scaling. In *ICCAD*, 2008.

[71] R Wayne Johnson, John L Evans, Peter Jacobsen, James R Thompson, and Mark Christopher. The changing automotive environment: high-temperature electronics. *IEEE Transactions on Electronics Packaging Manufacturing*, 27(3):164–176, 2004.

[72] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[73] Eugene Kim, Jinkyu Lee, Liang He, Youngmoon Lee, and Kang G Shin. Offline guarantee and online management of power demand and supply in cyber-physical systems. In *RTSS*, 2016.

[74] Minyong Kim, Young Geun Kim, Sung Woo Chung, and Cheol Hong Kim. Measuring variance between smartphone energy consumption and battery life. *Computer*, 47(7):59–65, 2014.

[75] Sharath Kodase, Shige Wang, Zonghua Gu, and Kang G Shin. Improving scalability of task allocation and scheduling in large distributed real-time systems using shared buffers. In *RTAS*, 2003.

[76] Joonho Kong, Sung Woo Chung, and Kevin Skadron. Recent thermal management techniques for microprocessors. *ACM Computing Surveys*, 44(3):13, 2012.

[77] Pratyush Kumar and Lothar Thiele. Cool shapers: Shaping real-time tasks for improved thermal guarantees. In *DAC*, 2011.

[78] Pratyush Kumar and Lothar Thiele. System-level power and timing variability characterization to compute thermal guarantees. In *CODES+ISSS*, 2011.

[79] Eren Kursun and Chen-Yong Cher. Temperature variation characterization and thermal management of multicore architectures. *IEEE Micro*, 29(1):116–126, 2009.

[80] Kai Lampka and Bjorn Forsberg. Keep It Slow and in Time : Online DVFS with Hard Real-Time Workloads. In *DATE*, 2016.

[81] Sheng-Chih Lin and Kaustav Banerjee. Cool chips: Opportunities and implications for power and thermal management. *IEEE Transactions on Electron Devices*, 55(1):245–255, 2008.

[82] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

[83] Y. Liu, R. P. Dick, L. Shang, and H. Yang. Accurate temperature dependent integrated circuit leakage power estimation is easy. In *DATE*, 2007.

[84] Yongpan Liu and Huazhong Yang. Temperature-aware leakage estimation using piecewise linear power models. *IEICE Transactions on Electronics*, 93(12):1679–1691, 2010.

[85] Jieyi Long and Seda Ogrenci Memik. A framework for optimizing thermoelectric active cooling systems. In *DAC*, 2010.

[86] Yue Ma, Thidapat Chantem, Robert P Dick, Shige Wang, and X Sharon Hu. An On-Line Framework for Improving Reliability of Real-Time Systems on Big-Little Type MPSoCs. In *DATE*, 2017.

[87] Yue Ma, Thidapat Chantem, X Sharon Hu, and Robert P Dick. Improving lifetime of multicore soft real-time systems through global utilization control. In *GVLSI*, 2015.

[88] Nimrod Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the ACM*, 31(1):114–127, 1984.

[89] Chulhong Min, Chungkuk Yoo, Inseok Hwang, Seungwoo Kang, Youngki Lee, Seungchul Lee, Pillsoon Park, Changhun Lee, Seungpyo Choi, and Junehwa Song. Sandra helps you learn: the more you walk, the more battery your phone drains. In *UbiComp*, 2015.

[90] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *MobiCom*, 2012.

[91] Navigant Research. Transportation Outlook: 2025 to 2050. http://web.archive.org/web/20181020005230/https://www.navigantresearch.com/reports/transportation-outlook-2025-to-2050, 2016.

[92] New York Times. Accused of Slowing Old iPhones, Apple Offers Battery Discounts. https://web.archive.org/web/20190805185043/https://www.nytimes.com/2017/12/28/business/apple-iphone-batteries.html, Dec 2017.

[93] New York Times. Galaxy Note 7 Fires Caused by Battery and Design Flaws, Samsung Says. https://web.archive.org/web/20190501141352/https://www.nytimes.com/2017/01/22/business/samsung-galaxy-note-7-battery-fires-report.html, Jan 2017.

[94] Shuichi Oikawa and Raj Rajkumar. Linux RK: A Portable Resource Kernel in Linux. In *RTSS*, 1998.

[95] Pratyush Patel, Iljoo Baek, Hyoseung Kim, and Ragunathan Rajkumar. Analytical Enhancements and Practical Insights for MPCP with Self-Suspensions. In *RTAS*, 2018.

[96] Indrani Paul, Srilatha Manne, Manish Arora, W Lloyd Bircher, and Sudhakar Yalamanchili. Cooperative boosting: needy versus greedy power management. In *ISCA*, 2013.

[97] Padmanabhan Pillai and Kang Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP*, 2001.

[98] Mary C Potter, Brad Wyble, Carl Erick Hagmann, and Emily S McCourt. Detecting meaning in rsvp at 13 ms per picture. *Attention, Perception, & Psychophysics*, 76(2):270–279, 2014.

[99] Alok Prakash, Hussam Amrouch, Muhammad Shafique, Tulika Mitra, and Jörg Henkel. Improving mobile gaming performance through cooperative cpu-gpu thermal management. In *DAC*, 2016.

[100] Danil Prokhorov. *Computational intelligence in automotive applications*, 132, Springer, 2008.

[101] Moo-Ryong Ra, Jeongyeup Paek, Abhishek B Sharma, Ramesh Govindan, Martin H Krieger, and Michael J Neely. Energy-delay tradeoffs in smartphone applications. In *MobiSys*, 2010.

[102] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P Pipe, Thomas F Wenisch, and Milo MK Martin. Computational sprinting. In *HPCA*, 2012.

[103] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P Pipe, Thomas F Wenisch, and Milo MK Martin. Computational sprinting. In *HPCA*, 2012.

[104] Daler Rakhmatov, Sarma Vrudhula, and Chaitali Chakrabarti. Battery-conscious task sequencing for portable devices including voltage/clock scaling. In *DAC*, 2002.

[105] Robert Redelmeier. cpuburn. `https://github.com/patrickmn/cpuburn`.

[106] David Michael Rowe. *Thermoelectrics handbook: macro to nano*. CRC press, 2005.

[107] Onur Sahin, Lothar Thiele, and Ayse K Coskun. Maestro: Autonomous qos management for mobile applications under thermal constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(8):1557–1570, 2018.

[108] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[109] Lars Schor, Iuliana Bacivarov, Hoeseok Yang, and Lothar Thiele. Worst-case temperature guarantees for real-time applications on multi-core systems. In *RTAS*, 2012.

[110] Krishna Sekar. Power and thermal challenges in mobile devices. In *MobiCom*, 2013.

[111] Danbing Seto, John P Lehoczky, Lui Sha, and Kang G Shin. Trade-off analysis of real-time control performance and schedulability. *Real-Time Systems*, 21(3):199–217, 2001.

[112] Gaurav Singla, Gurinderjit Kaur, Ali K. Unver, and Umit Y. Ogras. Predictive Dynamic Thermal and Power Management for Heterogeneous Mobile Platforms. In *DATE*, 2015.

[113] Gaurav Singla, Gurinderjit Kaur, Ali K Unver, and Umit Y Ogras. Predictive dynamic thermal and power management for heterogeneous mobile platforms. In *DATE*, 2015.

[114] Kevin Skadron, Mircea Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-aware microarchitecture. In *ISCA*, 2003.

[115] Yongquan Sun, Lingxi Kong, Hassan Abbas Khan, and Michael Pecht. Li-ion battery reliability–a case study of the apple iphone. *IEEE Access*, 7:71131–71141, 2019.

[116] Xiaopeng Tang, Yujie Wang, Changfu Zou, Ke Yao, Yongxiao Xia, and Furong Gao. A novel framework for lithium-ion battery modeling considering uncertainties of temperature and aging. *Energy conversion and management*, 180:162–170, 2019.

[117] Ken W Tindell, Alan Burns, and Andy J. Wellings. Allocating hard real-time tasks: an NP-hard problem made easy. *Real-Time Systems*, 4(2):145–165, 1992.

[118] Liang Wang, Xiaohang Wang, and Terrence Mak. Adaptive routing algorithms for lifetime reliability optimization in network-on-chip. *IEEE Transactions on Computers*, 65(9):2896–2902, 2016.

[119] Shengquan Wang and Riccardo Bettati. Delay analysis in temperature-constrained hard real-time systems with general task arrivals. In *RTSS*, 2006.

[120] Xie, Qing and Kim, Jaemin and Wang, Yanzhi and Shin, Donghwa and Chang, Naehyuck and Pedram, Massoud. Dynamic thermal management in mobile devices considering the thermal coupling between battery and application processor. In *ICCAD*, 2013.

[121] Fengyuan Xu, Yunxin Liu, Qun Li, and Yongguang Zhang. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *NSDI*, 2013.

[122] Fengyuan Xu, Yunxin Liu, Thomas Moscibroda, Ranveer Chandra, Long Jin, Yongguang Zhang, and Qun Li. Optimizing background email sync on smartphones. In *MobiSys*, 2013.

[123] Shichun Yang, Cheng Deng, Yulong Zhang, and Yongling He. State of charge estimation for lithium-ion battery with a temperature-compensated model. *Energies*, 10(10):1560, 2017.

[124] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *USENIX ATC*, 2012.

[125] Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *RTAS*, 2016.

[126] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert Dick, Zhuoqing Mao, and Lei Yang. Accurate power estimation and automatic battery behavior based power model generation for smartphones. In *CODES+ISSS*, 2010.

[127] Sushu Zhang and KS Chatha. Thermal aware task sequencing on embedded processors. In *DAC*, 2010.

[128] Xinyu Zhang and Kang G Shin. E-mili: energy-minimizing idle listening in wireless networks. In *MobiCom*, 2011.

[129] Yifan Zhu and Frank Mueller. Feedback EDF scheduling exploiting dynamic voltage scaling. In *RTAS*, 2004.