Thermal-Aware Scheduling for Integrated CPUs–GPU Platforms

YOUNGMOON LEE and KANG G. SHIN, University of Michigan, Ann Arbor
HOON SUNG CHWA, DGIST, South Korea

As modern embedded systems like cars need high-power integrated CPUs–GPU SoCs for various real-time applications such as lane or pedestrian detection, they face greater thermal problems than before, which may, in turn, incur higher failure rate and cooling cost. We demonstrate, via experimentation on a representative CPUs–GPU platform, the importance of accounting for two distinct thermal characteristics—the *platform's temperature imbalance* and *different power dissipations of different tasks*—in real-time scheduling to avoid any burst of power dissipations while guaranteeing all timing constraints. To achieve this goal, we propose a new *Real-Time Thermal-Aware Scheduling* (RT-TAS) framework. We first capture different CPU cores' temperatures caused by different GPU power dissipations (i.e., *CPUs–GPU thermal coupling*) with core-specific thermal coupling coefficients. We then develop *thermally-balanced* task-to-core assignment and *CPUs–GPU co-scheduling*. The former addresses the platform's temperature imbalance by efficiently distributing the thermal load across cores while preserving scheduling feasibility. Building on the thermally-balanced task assignment, the latter *cooperatively* schedules CPU and GPU computations to avoid simultaneous peak power dissipations on both CPUs and GPU, thus mitigating excessive temperature rises while meeting task deadlines. We have implemented and evaluated RT-TAS on an automotive embedded platform to demonstrate its effectiveness in reducing the maximum temperature by $6-12.2°C$ over existing approaches without violating any task deadline.

CCS Concepts: • **Computer systems organization** → **Embedded hardware**; **Embedded software**;

Additional Key Words and Phrases: Thermal management, embedded systems, GPU, real-time systems

ACM Reference format:
Youngmoon Lee, Kang G. Shin, and Hoon Sung Chwa. 2019. Thermal-Aware Scheduling for Integrated CPUs–GPU Platforms. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 90 (October 2019), 25 pages.
https://doi.org/10.1145/3358235

1 INTRODUCTION

As modern embedded systems like cars increasingly use integrated CPUs–GPU system-on-chips (SoCs) with growing power dissipations, thermal challenges therein have become critical. Hardware cooling solutions like fans have been used to lower chip temperature, but the cooling cost

This article appears as part of the ESWEEK-TECS special issue and was presented at the International Conference on Embedded Software (EMSOFT) 2019.
Authors' addresses: Y. Lee and K. G. Shin (corresponding author), University of Michigan, Ann Arbor, 2260 Hayward St. Ann Arbor, MI, 48109-2121; emails: {ymoonlee, kgshin}@umich.edu; H. S. Chwa (corresponding author), DGIST, South Korea, 333, Techno Jungang Daero, Hyeonpung-Myeon, Dalseong-Gun, Daegu, South Korea, 42988; email: chwahs@dgist.ac.kr.

1539-9087/2019/10-ART90 $15.00
https://doi.org/10.1145/3358235

has been increasing rapidly, which is estimated to be $3 per watt of heat dissipation [26]. Chip overheating not only incurs higher cooling cost but also degrades its reliability [27], which may, in turn, risk physical safety. Thus, reducing on-chip temperature while meeting the application timing constraints has become a key system design objective.

There are two key thermal characteristics to consider for integrated CPUs–GPU platforms: (1) the platform's temperature imbalance and (2) different CPU and GPU power dissipations by different tasks. Our experimentation on a representative CPUs–GPU SoC has shown 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 show higher temperatures than others due to the heat conduction from GPU. Besides this platform's temperature imbalance, our experimentation with automotive vision tasks has demonstrated an up to $1.35\times$ difference of CPU's power dissipations and an up to $2.68\times$ difference of GPU power dissipations by different tasks; some (*hot*) tasks dissipate more power than others when executed on a CPU/GPU.

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 temperature $5.6°C$ higher than assigning it to a cold core, significantly increasing cooling costs and/or severely degrading app performance by drastic hardware throttling (to be detailed in Section 3.2, Section 7). 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, and a scheduling decision on CPUs and 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 [14, 16] and multiprocessor systems [2, 6, 15]. They usually employ idle-time scheduling [14], DVFS scheduling [15], or thermal-isolation server [2] 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 didn't consider 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 [10, 21, 24, 25]. Prior studies recognized thermally-efficient cores [24] and demonstrated the platform's thermal imbalance through infrared imaging [10]. However, they are not suitable for safety/time-critical systems like in-vehicle vision systems. To the best of our knowledge, there is no prior work 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 paper, 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-schedule CPUs and GPU to reduce the maximum chip temperature while meeting task/job deadlines.

We first capture the different CPUs' temperatures caused by 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 validate 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 considering CPUs–GPU thermal coupling. We then develop a thermally-balanced task-to-core assignment, called T-WFD, that 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, so as to avoid simultaneous executions of hot tasks on both CPUs and GPU, thus mitigating excessive temperature increase without missing any task deadline. Finally, we have implemented RT-TAS on a representative CPUs–GPU platform [1] and evaluated it with automotive vision workloads [22], demonstrating its effectiveness in reducing the maximum temperature by $6-12.2°C$ compared to existing approaches without violating timing constraints, thus providing a reliable response time under a given chip temperature limit. This $6°C$ reduction translates to a 1.52× improvement of chip lifetime reliability [27] or savings of cooling cost by $15.6 per chip [17, 26].

This paper makes the following contributions:

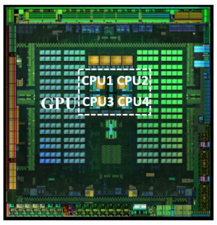
- Demonstration of the importance of co-scheduling CPUs and GPU while accounting for their thermal coupling (Section 3.2);
- Empirically capturing CPUs–GPU thermal coupling effect and temperature differences among CPU cores (Section 5);
- Development of thermally-balanced task-to-core assignment and CPUs–GPU co-scheduling (Section 6);
- Implementation and evaluation of RT-TAS on a popular CPUs–GPU platform with automotive vision workloads (Section 7).

2 RELATED WORK

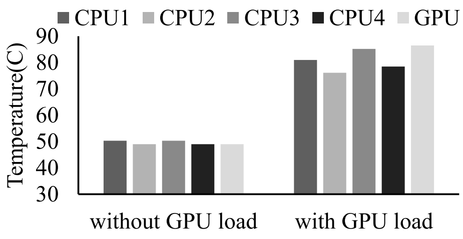
Prior research in the field of real-time systems focused on thermal-aware task and DVFS scheduling while meeting timing constraints for uni-processor [14, 16], multiprocessor platforms [2, 6, 15]. Kumar et al. [14] proposed a thermal shaper to regulate the runtime chip temperature by inserting idle periods. Lampka et al. [15] proposed a history-aware dynamic voltage/frequency scaling (DVFS) scheme that raises core frequency only in case of potential timing violations. A thermal-isolation server [2] was proposed to avoid the 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.

There exist studies on GPU thermal management for non-real-time systems [10, 21, 24, 25]. 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 [25]. Prakash et al. proposed CPU–GPU cooperative frequency scaling for a mobile gaming app [21]. The notion of thermally-efficient core was proposed in [24], where the CPU core less impacted by GPU heat dissipation was identified in offline and tasks were assigned in the order of thermally-efficient cores. The infrared imaging characterized the CPUs–GPU thermal coupling, introducing scheduling challenges [10]. Although all of these studies made valuable contributions, they did not deal with the timing constraint when applying DVFS or scheduling tasks, rendering them infeasible for time-critical embedded systems.

To the best of our knowledge, there is no prior work that addresses the challenges of thermal-aware assignments and scheduling of real-time tasks on CPUs–GPU platforms while accounting



(a) CPUs–GPU SoC [1]



(b) Platform's temperature imbalance

Fig. 1. CPUs surrounded by the GPU cluster on a SoC (Tegra X1) where GPU's power dissipation affects CPUs' temperatures, creating temperature imbalance across CPUs.

for the platform's temperature imbalance. Unlike 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 have implemented and evaluated RT-TAS, demonstrating its effectiveness on a representative CPUs–GPU platform with automotive vision workloads.

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.

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. Typical vision apps include feature detector, object tracker, and motion estimator [22]. The feature detector captures features to detect various objects, such as cars, road signs, pedestrians, etc. The object tracker maps and tracks moving/standing objects in consecutive input frames. The 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 GPU, CPU transfers data to GPU memory and calls GPU functions. GPU then performs the required computation and returns the result back to the CPU. See [11] for GPU operation details for real-time apps.

3.2 Thermal Characteristics of CPUs–GPU Platforms

To understand the thermal characteristics of CPUs–GPU platforms, we conducted experiments on Nvidia Tegra X1 [1] equipped with 4 CPUs and a GPU with representative vision workloads [22]. We highlight two key findings from this experimentation. First, GPU power dissipation raises CPUs' temperatures significantly at different rates, creating a large temperature imbalance on the platform cores. Second, different tasks dissipate different amounts of power on 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 (Figure 1(a)) for cost/power/communication efficiency [27]. CPU cores are usually surrounded by the GPU cluster [10], and hence the GPU's power dissipation greatly affects CPU cores' temperatures because of heat transfer. To understand the GPU's thermal impact on CPU cores, we measured the temperatures of CPU and GPU cores

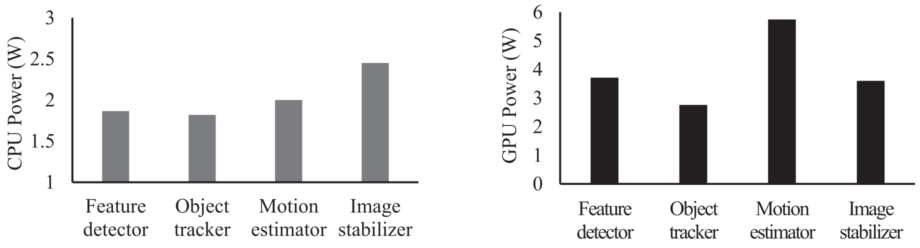


Fig. 2. Average power dissipations of (a) CPU and (b) GPU vary greatly by application tasks.

with and without GPU workload.[1] Figure 1(b) corroborates CPUs–GPU thermal coupling where the GPU workload raises CPU cores' temperatures from $50°C$ to $77°C$ on average without any CPU workload. More importantly, we observed a significant temperature difference across CPU cores up to $10°C$ (CPU2 vs. CPU3) in the presence of GPU workload. This imbalance is due to CPU3's close proximity to the GPU cluster, and thus the significant impact of GPU power dissipation (Figure 1(a)). 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.

We conducted the same experiments on other SoCs—Snapdragon 810 and Exynos 5420—with different chip layouts and observed similar trends of temperature imbalance (see Appendix A for the experimental results). Existing studies also reported large temperature imbalances in various integrated CPUs–GPU platforms, such as MD A10-5700 [10] and Trinity APU [19].

Power dissipations of app tasks. Besides the underlying platform's temperature imbalance, different tasks incur significantly different amounts of power dissipations on CPUs and GPU. Figure 2 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 6.2). On CPU, the image stabilizer is the hottest, dissipating 1.35× more power than the coldest, the object tracker. On GPU, the motion estimator is the hottest, dissipating 2.68× more power than the coldest, the object tracker.

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 in 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) simultaneous execution of both CPU and GPU workloads; ii) execution of CPU workload alone; and iii) no execution (idling). For cases i), ii), and iii), the maximum temperature was recored at $79.6°C$, $55.6°C$, and $50.3°C$, respectively. Simultaneous CPU and GPU executions make the CPU temperature $24°C$ higher than the case of CPU execution alone. Moreover, assigning the CPU workload to a *hot* core results in $85.2°C$, a CPU temperature increase by $5.6°C$ over the case of assigning it to a *cold* core. Such an excessive temperature increase/imbalance may result in i) high cooling cost, ii) poor

[1]Note that in this motivational experiment, we used the GPU thermal benchmark [9], i.e., CPU remains idle, to minimize the impact of each CPU's own power dissipation.
[2]We chose the CPU and GPU workloads from the thermal benchmark [9, 23] designed for CPU/GPU stress tests with high power dissipations of 4.67W and 9.89W, respectively.

reliability, and/or iii) performance degradation due to 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. So, 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* as they do not consider 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.

Besides the platform's temperature imbalance, tasks' different CPU/GPU power-dissipation variations make the scheduling of app tasks on CPUs and GPU important. Scheduling CPUs and GPU *independently* may adversely affect the peak temperature if both CPUs and GPU simultaneously run *hot* tasks, both dissipating a large amount of heat. So, we need to *cooperatively* schedule CPU and GPU computations to avoid the burst of power dissipations on the CPUs–GPU platform, while meeting all timing constraints.

4 PROBLEM STATEMENT AND SOLUTION OVERVIEW

We want to solve the following thermal-aware scheduling problem for real-time CPUs–GPU systems. Let $\pi = \{\pi_1, \dots, \pi_m, \pi_g\}$ be the set of m CPU cores and a shared GPU, and let their temperatures at time t be $T_{\pi_x}(t)$, $\pi_x \in \pi$.

Definition 4.1. Given a task set τ running on a CPUs–GPU platform π, determine (i) task-to-core assignment, (ii) schedule of jobs to be executed on CPU and GPU cores such that

G1. the maximum chip temperature, $\max_{\pi_x \in \pi} T_{\pi_x}(t)$, is minimized; and
G2. all the jobs of $\tau_i \in \tau$ meet their deadlines for all possible legitimate job arrival sequences (timing constraints).

For a schedule, we need to determine not only the core state (active or idle) but also the order of executing jobs in the active state.

We consider the partitioned fixed-priority preemptive scheduling policy, which is widely used in various real-time systems. We develop a Real-Time Thermal-Aware Scheduling (RT-TAS) framework that accounts for the platform's temperature imbalance and diverse power dissipations of tasks on CPUs and GPU. The main challenges are: (C1) how to capture the platform's temperature imbalance and power dissipation variations of tasks; (C2) efficiently map tasks to CPU cores; and (C3) schedule task/job executions on CPU and GPU cores to reduce the maximum chip temperature while meeting timing constraints.

To address C1, we develop a CPUs–GPU thermal coupling model that can capture different rates of temperature increase of CPUs due to the power dissipation on GPU by using core-level thermal coefficients (in Section 3.2). Based on this thermal coupling model, we analyze different thermal effects of different tasks with distinct power dissipations on CPUs and GPU. We empirically identify the thermal parameters and validate the CPUs–GPU thermal coupling model on a representative embedded platform for automotive vision apps.

To address C2, we propose the concept of *thermally-balanced assignment* to gauge the heat distribution across cores on the CPUs–GPU platform. We then develop a new thermal-aware

[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.

task-to-core assignment, called T-WFD, by considering the platform's temperature imbalance and different power dissipations of tasks (in Section 6.1). We prove that T-WFD always yields a thermally-balanced assignment that efficiently reduces the temperature difference across cores. Note that a task-to-core assignment dictates the heat distribution across CPU cores, which, in turn, affects their steady-state temperatures. Therefore, it is important to find a mapping from tasks to cores such that the maximum steady-state temperature is minimized.

To address C3, building on the thermally-balanced task-to-core assignment derived by addressing C2, we develop an online CPU–GPU co-scheduling algorithm (in Section 6.2). While we assign tasks to cores in a thermally-balanced manner in terms of the steady-state temperature, an actual schedule of jobs on CPU and GPU cores may affect the *transient* temperature, potentially overheating the chip before reaching its steady-state temperature. Since the transient temperature is increased by both CPU and GPU power dissipations by currently running tasks, scheduling hot tasks on both CPU and GPU at the same time causes the transient temperature to rise. To avoid this, we need to schedule a cold task or insert an idle-time on CPU when a hot task is running on GPU, and vice versa. We thus develop a thermal-aware CPU–GPU co-scheduling algorithm that finds candidate tasks whose executions do not violate timing constraints at each scheduling instant of a CPU and then determines which task to schedule by considering the currently running task on its counterpart (GPU), and vice versa, to achieve the desired thermal behavior.

Specifically, at each CPU/GPU scheduling time instant, the scheduler selects a job so that the difference between the steady-state and the transient temperatures due to the execution of the selected job and the currently running job on CPU and GPU is minimized without compromising timing constraints. This way, our co-scheduling algorithm effectively mitigates any excessive transient temperature rise without violating any timing constraint.

5 CPUs–GPU SYSTEM MODEL

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

5.1 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 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 lock (mutex) because most contemporary GPUs perform their assigned computation non-preemptively. The GPU access is then made with the MPCP protocol, well-known

[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 [13].

locking-based GPU access scheme [12, 18]. 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 up and granted access to 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 in the following expression:

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

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 [18]. 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 [18]. 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 $\lceil \frac{w_i + w_h - e_h^C}{p_h} \rceil$. Then, I_i^a is derived as

$$I_i^a = \sum_{\tau_h \in hpp(\tau_i)} \left\lceil \frac{w_i + w_h - e_h^C}{p_h} \right\rceil \cdot e_h^C. \tag{2}$$

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 [18], B_i^a is derived as

$$B_i^a = B_i^{dr} + B_i^{pr}, \tag{3}$$

where

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

$$B_i^{pr} = \sum_{\tau_h \in lpp(\tau_i)} \left\lceil \frac{w_i + d_l - e_l^C}{T_l} \right\rceil \cdot e_l^G.$$

A detailed proof of this can be found in [18]. Then, we can check the schedulability of a task set as presented in the following lemma:

Lemma 5.1. *[18] A task set τ is schedulable if*

$$\forall \tau_i \in \tau, \ w_i \le d_i. \tag{4}$$

5.2 CPU and GPU Power-dissipation Model

As shown in Figure 2, CPU and GPU power dissipations are found to vary significantly with the executing task. Since 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 CPU execution (P_i^C) and GPU execution (P_i^G) of τ_i. Since 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:

$$P_{\pi_c}(\Lambda) = \sum_{\tau_i \in p(\pi_c)} P_i^C \cdot u_i^C, \quad P_{\pi_g}(\Lambda) = \sum_{\tau_i \in \tau} P_i^G \cdot u_i^G, \tag{5}$$

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

5.3 Platform's Thermal Model

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

$$T_\pi(t) = T_A + R \cdot P_\pi(t) + R \cdot C \cdot \frac{dT_\pi(t)}{dt} \tag{6}$$

where $T_\pi(t) = [T_{\pi_1}(t), \ldots, T_{\pi_m}(t), T_{\pi_g}(t)]$ is an $m+1$ element vector of CPUs 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 diagonal matrix with the thermal capacitance of each component. With the thermal circuit model shown in Equation (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:

$$T_\pi(t) = e^{(R\cdot C)^{-1}\cdot t} \cdot T_\pi(0) + (1 - e^{(R\cdot C)^{-1}\cdot t}) \cdot (T_A + R \cdot P_\pi(t)). \tag{7}$$

where $T_\pi(0)$ is the initial temperature of the processor. One can observe from Equation (7) that the temperature will increase/decrease towards 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

$$T_\pi = T_A + R \cdot P_\pi. \tag{8}$$

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:

$$T_{\pi_x}(\Lambda) = T_A + R_x \cdot P_{\pi_x}(\Lambda) + \sum_{\pi_y \in \pi \setminus \pi_x} R_{x,y} \cdot P_{\pi_y}(\Lambda) \tag{9}$$

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

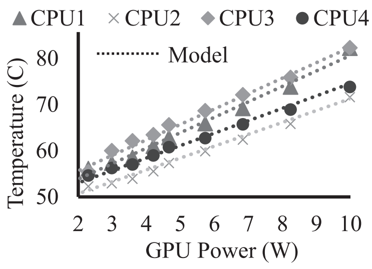
5.4 Parameter Identification and Validation

Parameter identification. The thermal resistance R and capacitance C are SoC-specific parameters in the platform's thermal model in Equation (6). While considering the CPUs–GPU thermal coupling, we identify these SoC-specific parameters using a typical thermal parameter identification process [7] as follows. We run a GPU benchmark [9] on GPU with CPU cores kept idle, and measure the power dissipation of GPU and the steady-state temperatures of CPU and GPU cores. We then determine each core's thermal coefficient w.r.t. GPU heating impact by using Equation (8). Likewise, we run a CPU benchmark [23] on each CPU core, one at a time, and determine each

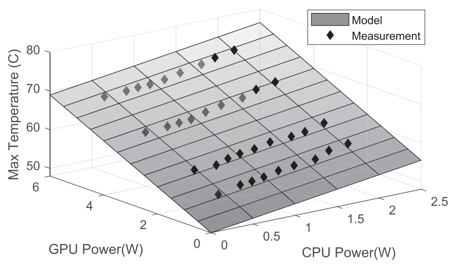
[5]Note that this thermal model has been shown to be reasonably accurate [16, 25].

Table 1. Thermal Coupling Coefficients for Tegra X1 (° C/W)

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



(a) Varied GPU power



(b) Varied CPU/GPU power

Fig. 3. (a) CPU temperatures resulting from varied GPU power dissipations and (b) maximum CPU temperature resulting from varied CPU and GPU power dissipations.

core's thermal coefficient w.r.t. CPU heating impact. From these results, we identify the thermal-coupling coefficients by using $R_{x,y} = \Delta T_{\pi_x}/P_{\pi_y}$ as in Table 1. Note that the thermal coefficient values shown in Table 1 are for Nvidia Tegra X1. By taking the above parameter identification process for 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 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 Figure 3.

Figure 3(a) plots the measured CPU temperatures resulting from the varied GPU's power dissipation. CPU temperatures linearly increase with GPU's power dissipation at different rates that is captured by core-level thermal coupling coefficients As GPU's power dissipation increases from 2.2W to 8.2W, the temperature of CPU3 increases at most by 14.3$°C$ while that of CPU2 increases by 9.4$°C$. Figure 3(b) plots the maximum chip temperature while varying CPU and GPU power dissipations. The result shows that the maximum chip temperature linearly increases with both CPU and GPU power dissipations as in Equation (9).

6 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 present a thermally-balanced assignment, called T-WFD, that equilibrates the platform's thermal imbalance by considering different power dissipations of tasks while preserving feasibility (Section 6.1). 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 CPU and GPU and thus effectively reduces the peak chip temperature while meeting task deadlines (Section 6.2). While our task-to-core assignment algorithm minimizes the maximum steady-state temperature of CPU cores, our co-scheduling policy regulates CPUs' and GPU's power dissipations to mitigate the increase of transient temperature.

6.1 Thermally-balanced Assignment

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

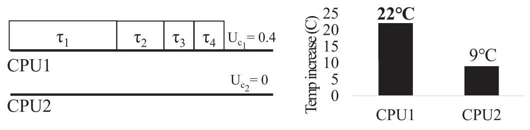
Definition 6.1 (Task-to-core Assignment). Given a task set τ and a CPUs–GPU platform π, find a mapping from the tasks of τ to the CPU cores in π (i.e., task-to-core assignment Λ) such that the maximum steady-state temperature of the cores is minimized while all tasks mapped onto each core meet their deadlines under fixed-priority scheduling with MPCP.

The task-to-core assignment problem is NP-hard, since finding a feasible mapping is equivalent to the bin-packing problem which is known to be NP-hard in the strong sense [3]. Thus, we need to 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) [8]. 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 there are multiple cores 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.

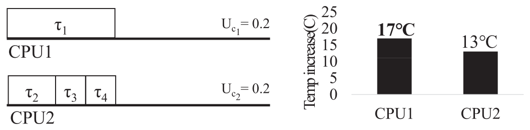
Example 6.2. Let us consider a set of four vision tasks shown in Figure 2 and a platform consisting of two CPU cores and a single GPU. The CPU utilizations of individual tasks, i.e., $u_i^C = e_i^C/p_i$, are $u_1^C = 0.2$, $u_2^C = 0.1$, $u_3^C = 0.05$, and $u_4^C = 0.05$. The CPU's power dissipations 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 Figure 1(b) where CPU1 heats up faster than CPU2 due to the CPUs–GPU thermal coupling. We consider four possible task-to-core assignment algorithms as shown in Figure 4: (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 shows lower maximum steady-state temperature than FFD/BFD, it is not thermally-optimal. In fact, there exists a thermally-optimal assignment shown in Figure 4(c).

Note that FFD and BFD try to pack as many tasks as possible on one core while keeping the other cores empty to accommodate other unassigned tasks. On the other hand, WFD tends to distribute the workloads evenly across all cores. In general, FFD and BFD have shown better feasibility than WFD [4]. However, they may result in higher temperatures than WFD since 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 Figure 4(b). As shown in Figure 4(c), a thermally-optimal assignment in this example turns out to be the one that assigns slightly more CPU workloads to CPU2 than

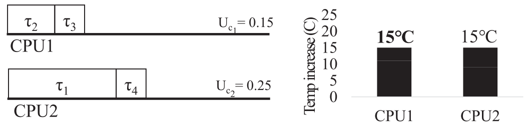
[6]Note that the temperature rise of CPU2 in Figure 4(a) is due to the indirect effect of the execution of workloads on GPU, i.e., CPUs–GPU thermal coupling.



(a) FFD and BFD



(b) WFD



(c) Thermally-optimal assignment

Fig. 4. Task-to-core assignment algorithms and their corresponding temperature increases.

CPU1. This is because CPU2 provides a more thermally-efficient operation than CPU1 due to 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 6.3 (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 a 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 1, in task-to-core assignment, we assume that the difference in thermal conduction rate from one CPU to other CPUs is negligible ($R_{c1,c2} \simeq R_{c2,c1} \simeq R_{c1,c3}$ where $\forall 1 \leq c1, c2, c3 \leq m$).[7]

Lemma 6.4. *A task-to-core assignment Λ is thermally-balanced if for every pair (π_p, π_q) s.t. $\pi_p, \pi_q \in \{\pi_1, \dots, \pi_m\}$ and every task $\tau_i \in p(\pi_p)$ satisfy*

$$T_{\pi_p}(\Lambda) - T_{\pi_q}(\Lambda) \leq R_p \cdot P_i^C \cdot u_i^C. \tag{10}$$

Proof. Suppose that a task-to-core assignment Λ satisfies Equation (10). Without 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. There are two possibilities: i) $T_{\pi_p}(\Lambda) > T_{\pi_q}(\Lambda)$, and ii) $T_{\pi_p}(\Lambda) \leq T_{\pi_q}(\Lambda)$.

[7]Note that we still consider a different thermal coefficient value for other elements in Table 1, such as R_c, R_g, $R_{g,c}$, and $R_{c,g}$.

ALGORITHM 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 $\Lambda$
```

Case i): according to Equations (5) and (9), $T_{\pi_p}(\Lambda') = T_{\pi_p}(\Lambda) - R_p \cdot P_i^C \cdot u_i^C$ and $T_{\pi_q}(\Lambda') = T_{\pi_q}(\Lambda) + R_q \cdot P_i^C \cdot u_i^C$. 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, we are moving a task τ_i from a low temperature core to a high temperature core. The resulting assignment Λ' can be easily 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 1. Unlike the previous algorithms presented in Example 6.2, tasks are sorted in 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 their temperature taking the CPUs–GPU coupling into account. Then, T-WFD allocates each task to the core with lowest temperature on which the allocation can preserve feasibility by the schedulability test in Lemma 5.1. This way, it is possible to find a thermally-balanced assignment.

We now prove that T-WFD never produces a thermally-unbalanced assignment in the following theorem.

Theorem 6.5. *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 the statement by induction. Clearly, after assigning the first task τ_1 to the core with the lowest temperature on which it fits,

ALGORITHM 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**
```

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 $p \neq c$. There are two possible cases: 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,

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

Due to 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 6.4, the pair (π_c, π_p) cannot be unbalanced.

Case ii): after assignment of τ_{k+1} to π_c, let π_q is 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. Algorithm 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 by 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 $O(\max(n \cdot logn, n \cdot m))$.

6.2 CPU–GPU Co-scheduling

So far, we have discussed the task assignment to handle the platform's temperature imbalance. Building on the thermally-balanced assignment, we now show 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 6.6 (schedule-generation). Given the task-to-core assignment, determine a schedule of job executions and idle-times on both CPU and GPU such that the maximum transient temperature across CPU and GPU cores is minimized while all the jobs of all tasks $\tau_i \in \tau$ meet their deadlines.

Addressing 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 preserving feasibility under fixed-priority scheduling with MPCP [18]. However, a job schedule on CPU and GPU cores may affect the *transient* temperature, potentially leading to chip overheating before reaching the steady-state temperature, due to the following two key issues: 1) different power dissipations of tasks on CPU and GPU cores, and 2) CPUs–GPU thermal coupling. Specifically, due to the different power dissipations by different tasks (as shown in Figure 2), the temperatures of CPU and GPU vary greatly depending on the tasks currently running on their cores. We observe from Equations (7) and (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 CPU and GPU. In addition, because of heat conduction by CPUs–GPU thermal coupling, the tasks scheduled on 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 CPU and GPU together. One may slow down 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 deadline miss of the hot task. This calls for cooperative scheduling of CPU and GPU computations, i.e., scheduling CPU job while considering GPU schedules, and vice versa, to effectively mitigate excessive temperature rise without missing any task deadline.

We develop a thermal-aware CPU–GPU co-scheduling mechanism that determines which tasks to run on CPU and 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 CPU when its counterpart (GPU) is running a hot task, and vice versa—subject to schedulability constraints. Such a mechanism avoids simultaneous executions of hot tasks on both CPU and GPU and thus reduces the peak temperature while ensuring that all tasks still meet their deadlines. Algorithm 2[8] presents our thermal-aware CPU–GPU co-scheduling which consists of two steps: (i) *candidate selection* and (ii) *job selection*. Whenever a scheduling decision is to be made on CPU and 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 so that the difference between the steady-state temperature and the transient temperature caused by the execution of the selected and current jobs on CPU and GPU is minimized (lines 8–10).

Finding candidate jobs. To prevent any deadline miss due to the priority inversions, we calculate the *worst-case maximum inversion budget* V_i for each task τ_i allowed for 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 [28]. 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, due to the additional interference by the deferred executions (also known as back-to-back hit) [28]. To

[8]Algorithm 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 Section 7.1.

taking into account such deferred executions when calculating V_i, we derive a pessimistic upper-bound on the worst-case response time w_i^* by 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

$$V_i = d_i - w_i^*. \tag{11}$$

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) release of a new job, (ii) completion of a job, or (iii) no remaining inversion budget of a job. Upon each invocation, our scheduling algorithm find 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, let $e_{i,cur}^C$ denote the remaining CPU section execution time at time t_{cur}.

LEMMA 6.7. *For a task τ_i, if $\forall \tau_h \in hpp(\tau_i)$ satisfies $v_h - e_{i,cur}^C \geq 0$ 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, preemption delay of CPU execution by $hpp(\tau_h)$, blocking time to acquire a GPU access, and further delay of CPU execution due to priority inversions by our co-scheduling policy. Suppose that at time t_{cur}, our co-scheduling policy decides to execute τ_i which is lower priority than τ_h at time t. Then, the worst-case busy interval of τ_h is bounded by

$$e_h^C + e_h^G + I_h + B_h + e_{i,cur}^C \leq w_h^* + e_{i,cur}^C \leq w_h^* + v_h = d_h,$$

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

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

LEMMA 6.8. *If $\forall \tau_h \in hp(\tau_i)$ satisfies $v_h - e_{i,cur}^G \geq 0$ or τ_i is the highest-priority task, τ_i is a candidate for GPU execution.*

PROOF. This lemma can be proved similarly to Lemma 6.7. □

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

$$\bar{P}_{tot} = \sum_i \left(P_i^C \cdot \frac{e_i^C}{p_i} + P_i^G \cdot \frac{e_i^G}{p_i} \right). \tag{12}$$

Let P_{cur}^C and P_{cur}^G denote the power dissipation by the current running job on CPU 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 ($P^C_s + \sum_{\pi \setminus \pi_c} P^C_{cur} + P^G_{cur}$) by the selected and currently running jobs on CPU and GPU is minimized, that is,

$$\min_{\tau_s \in \Gamma_{\pi_c}} \left| \bar{P}_{tot} - \left(P^C_s + \sum_{\pi \setminus \pi_c} P^C_{cur} + P^G_{cur} \right) \right|. \tag{13}$$

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

$$\min_{\tau_s \in \Gamma_{\pi_g}} \left| \bar{P}_{tot} - \left(\sum_{\pi_c} P^C_{cur} + P^G_s \right) \right|. \tag{14}$$

With the proposed CPU–GPU co-scheduling algorithm, we can reduce the variation in the steady-state and transient temperatures, thus effectively mitigating any excessive rise in transient temperature while guaranteeing all deadlines to be 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 by Algorithm 2 with $O(n)$ complexity, where n is the number of tasks. Then, our algorithm selects a task based on Equations (13)–(14) with $O(1)$ complexity. Thus, the total complexity is $O(n)$.

7 EVALUATION

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

- Maximum temperature is reduced by $6°C$ and $12.2°C$ w.r.t. the state-of-the-art [24] and a default OS scheduler, respectively.
- Our thermally-balanced assignment reduces the maximum temperature by $3.9°C$ and CPU–GPU co-scheduling reduces 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 (Figure 4(b)).

7.1 Methodology

Our experimental platform is Nvidia Tegra X1 equipped with 4 CPU cores and a shared GPU [1] rated at the maximum power dissipation of $15W$. To avoid chip overheating, each CPU/GPU is equipped with a thermal sensor. The built-in hardware temperature management kicks in when one of its cores reaches the temperature threshold, and lowers CPU frequency to cool down the temperature. According to the thermal specification [1], chip thermal resistance is $1.15°C/W$. To evaluate the benefit of RT-TAS under a realistic setup, we have implemented a real-time vision system running representative vision workloads [22]: (i) feature detector, (ii) object tracker, (iii) motion estimator, and (iv) image stabilizer. An in-vehicle camera video is given to these tasks as input. Specifically, we have implemented RT-TAS on top of Linux kernel as an user-level application that 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 by using `sched_setaffinity` and `CPU_SET`;
- Priority-based scheduling using `sched_setscheduler` under the `SCHED_FIFO`; and

Table 2. Vision Tasks Used in 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

- Implementing a GPU lock using `pthread_mutex`, and the highest priority task waiting for the lock will grab the lock.

Throughout the evaluation, we compare following approaches:

- BASE: default OS scheduler (completely fair scheduling) [20];
- TEA: thermally-efficient allocation [24] assigning tasks from the most thermally-efficient core first[9];
- RT-TAS: the proposed thermally-balanced assignment (Section 6.1) and CPU–GPU co-scheduling (Section 6.2).

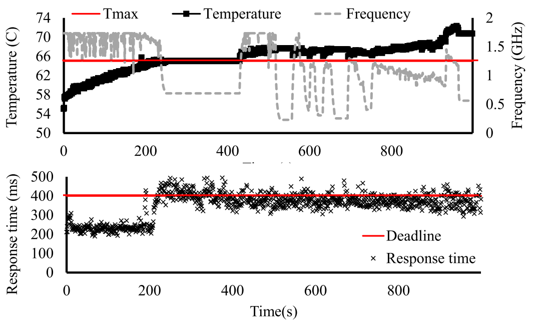
To avoid external influences, the external fan is turned off. Unless otherwise specified, the temperature threshold is set to $65°C$, and CPU and GPU cores are running at the maximum frequency. During our experiments, the WCET of each job is recorded to check if the job deadlines are met. The CPU/GPU execution time, power dissipation, and period of tasks are provided in Table 2.

7.2 Effectiveness in Reducing Temperature

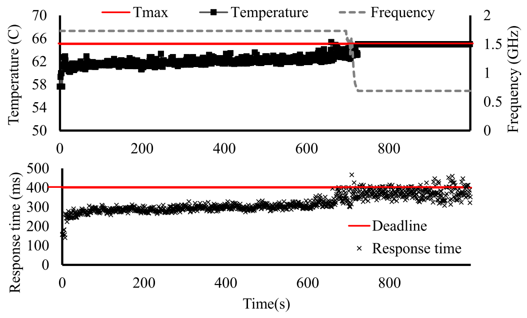
We first demonstrate RT-TAS's effective reduction of the maximum chip temperature, thus achieving reliable performance of a real-time vision system. Figure 5 plots the maximum transient temperature among cores, CPU frequency, and task response time for different schemes. With BASE (Figure 5(a)), temperature exceeds the threshold at around $200s$ and hardware thermal throttling was triggered to reduce the processor frequency from $1.7GHz$ to $0.5GHz$. As a result, the maximum response time increases from $309ms$ to $493ms$ violating the deadline ($400ms$). However, the chip temperature increases further, reaching up to $73°C$. With TEA (Figure 5(b)), the maximum temperature increases less rapidly than with BASE, but temperature exceeds the threshold at around $800s$, and experiences thermal throttling thereafter. With RT-TAS (Figure 5(c)), temperature remains 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.

Figure 6 compares the CPUs' and GPU's peak temperatures, demonstrating how RT-TAS mitigates the temperature imbalance. With BASE, tasks were assigned without considering the temperature imbalance, which leads 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 better across CPU cores than BASE. However, tasks are 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 assigns tasks to the core in a thermally-balanced manner by capturing the core-level different GPU heating impact and power variations of tasks. Therefore, RT-TAS reduces the max-min difference to $1°C$, and the maximum temperature to $60.7°C$. Figure 7 compares the maximum

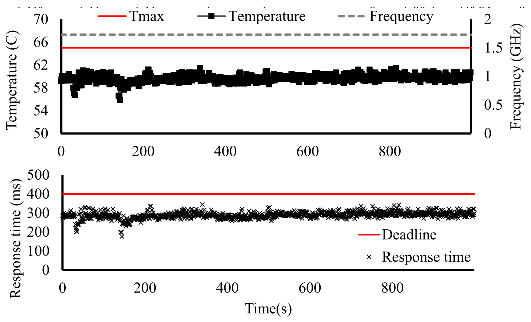
[9]TEA identifies thermally-efficient cores depending on the CPU power dissipation offline and then sequentially bind the task with the highest CPU usage to the next most thermally-efficient core.



(a) BASE

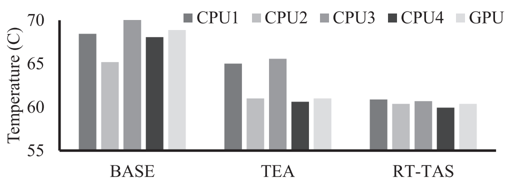


(b) TEA



(c) RT-TAS

Fig. 5. RT-TAS by reducing maximum temperature avoids thermal throttling, thus achieving reliable response time.



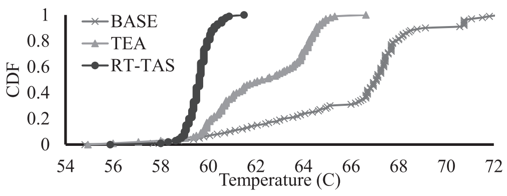


Fig. 6. Core-level peak temperature under different scheduling policies.

Fig. 7. Maximum temperature CDF under different scheduling policies.

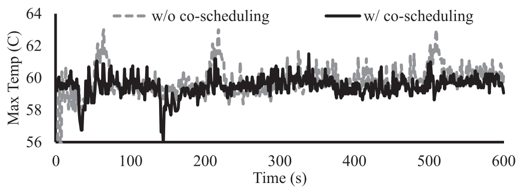
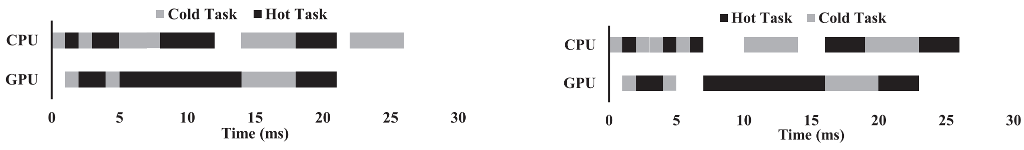


Fig. 8. Transient temperatures w/o and w/ CPU–GPU co-scheduling.



(a) w/o co-scheduling (b) w/ co-scheduling

Fig. 9. Job schedule (a) w/o and (b) w/ co-scheduling.

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

We analyzed the impact of co-scheduling by running the same experiment without and with co-scheduling. Figure 8 compares the transient temperature variation over time without and with co-scheduling. The peak temperature under CPU–GPU co-scheduling was $61.5°C$ at 217 seconds while that under no co-scheduling was $63.6°C$. Figure 9 shows the actual schedule of jobs in a specific time interval without and with co-scheduling. Without co-scheduling, tasks were scheduled in fixed-priority order on CPU and GPU independently (Figure 9(a)). In such a case, CPU and GPU may simultaneously perform peak computations (running hot tasks), thus incurring a peak total power dissipation. With co-scheduling CPU and GPU together, RT-TAS avoids the overlap of simultaneous peak computations on CPU and GPU to reduce the peak power dissipation (Figure 9(b)). As a result, co-scheduling effectively mitigates any excessive rise in transient temperature by avoiding bursts of peak power dissipation. We also analyzed the effectiveness of task assignment and co-scheduling in reducing the peak temperature, respectively. See Appendix B for results. Overall, the thermally-balanced assignment and CPU–GPU co-scheduling reduce the maximum temperature by $3.9°C$ and $2.1°C$, respectively, resulting in a total temperature reduction of $6°C$ over TEA. This $6°C$ reduced maximum temperature translates to the 1.52× longer chip lifetime[10] [27] and cooling cost savings of \$15.6 per chip[11] [1, 26].

7.3 Evaluation with Different Task Sets

Next, we evaluate RT-TAS with different task sets by using the parameters measured via the experiments. The base parameters in Table 3 are acquired from the above platform and sample vision tasks. We randomly generate 1,000 task sets, and task execution parameters ($\eta_i, e_i^C, e_i^G, P_i^G$) are set to be uniformly distributed within their maximum bounds. Next, the task utilization is determined according to the UUniFast algorithm [5], and the task period is set to $p_i = (e_i^C + e_i^G)/u_i$. We also

[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 [27]. 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 $3\$/W$[26] and the saving is $5.2 \times 3 = 15.6\$$.

Table 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

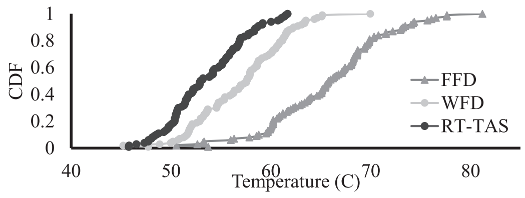


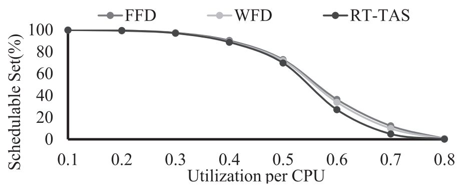
Fig. 10. Maximum temperature CDF for different task sets.

used the identified platform thermal parameters in Table 1 and the thermal model in Equation (6) to estimate the chip temperature. We compare RT-TAS against two baseline algorithms, FFD and WFD in Section 6.1.

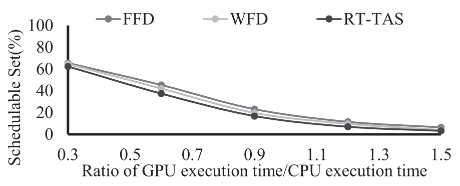
Figure 10 plots the maximum temperature dynamics for different task sets with base parameters. The maximum temperature reduction by RT-TAS is up to 8.3°C and 5.0°C on average. From the base task set parameters, we vary i) utilization ii) GPU execution time, and iii) task-level power variation for each experiment setting. We highlight three observations. The maximum temperature reduction by RT-TAS becomes 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 diminishes as the utilization increases, because a task assignment can no longer avoid assigning tasks to hot CPU cores. As the utilization per CPU increases from 0.3 to 0.6, the maximum temperature reduction by RT-TAS decreases from 5.0°C to 2.1°C on average. As the GPU execution time increases, the temperature reduction by RT-TAS becomes more pronounced due to 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 increases from 0.1 to 1, maximum temperature reduction by RT-TAS increases from 1.1°C to 4.6°C on average.

When the variation on task-level power dissipation is large, temperature decrease by RT-TAS becomes more pronounced. When the maximum difference between task-level power dissipations increases 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 task-level different power dissipations and the platform's temperature imbalance into account in task-to-core assignment and scheduling. See Appendix C for the detailed results.

We finally discuss RT-TAS's impact on schedulability. Figure 11 plots the schedulability as the percentage of the schedulable task sets out of 1,000 task sets with varied (a) utilization per CPU, (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



(a) Varied utilization



(b) Varied GPU/GPU execution time

Fig. 11. Schedulability for varied utilization and GPU execution time.

comparable to FFD. Across various configurations, T-WFD yielded only 3.6% less schedulable sets than FFD on average.

8 CONCLUSION

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

In this paper, we have considered partitioned fixed-priority scheduling with the MPCP protocol as a baseline. In 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.

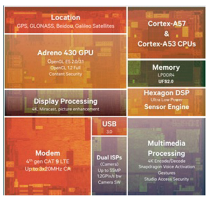
APPENDIX

A TEMPERATURE IMBALANCE ON OTHER SOCS

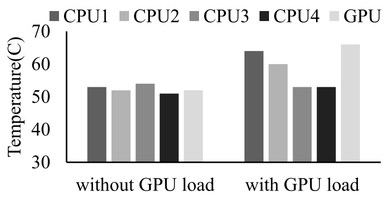
We conducted the same experiments as in Figure 1 on other SoCs—Snapdragon 810 (Figure 12(a)) and Exynos 5420 (Figure 13(a))—with different chip layouts. Figures 12(b) and 13(b) plot the maximum temperature of CPU and GPU cores with and without GPU workloads. We found the temperature difference among CPU cores with GPU workload can be up to 10°C for Snapdragon 810 and 7°C for Exynos 5422.

B IMPACT OF TASK ASSIGNMENT AND SCHEDULING

We also analyze the impact of thermally-balanced task-to-core assignment and CPU-GPU co-scheduling in detail. We measure the temperature characteristics while applying task-to-core assignment alone, then with co-scheduling together. Figure 14 show CPU and GPU cores' peak temperature highlighting the effect in mitigating the temperature imbalance, and Figure 15 shows the maximum temperature characteristics. Thermally-balanced task assignment assigns thermal loads on the lowest temperature core, thus reducing the maximum temperature from 66.7°C to 62.8°C and the max-min temperature difference from 5.7°C to 3.3°C compared to TEA. Furthermore, co-scheduling algorithm reduces the peak temperature to 60.7°C by cooperatively scheduling hot/cold CPU and GPU sections.

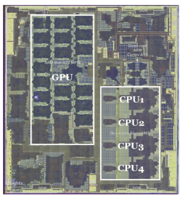


(a) Snapdragon 810

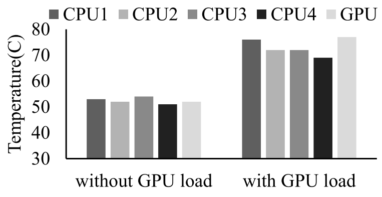


(b) Imbalance on Snapdragon 810

Fig. 12. Snapdragon 810 layout and thermal imbalance.



(a) Exynos 5422



(b) Imbalance on Exynos 5422

Fig. 13. Exynos 5422 layout and thermal imbalance.

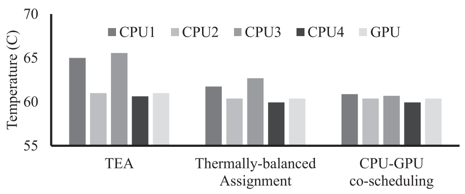


Fig. 14. Core-level peak temperature under task assignment and scheduling.

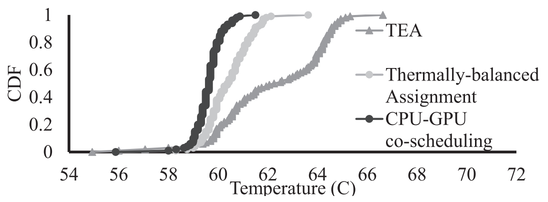
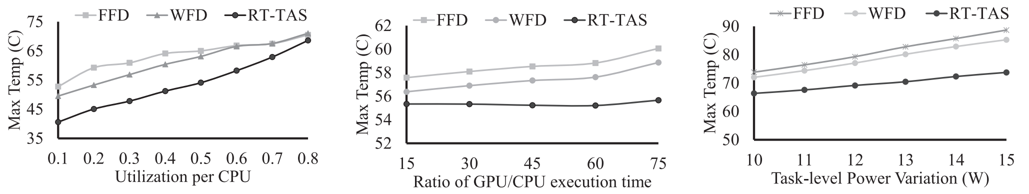


Fig. 15. Temperature reduction by RT-TAS's task assignment and scheduling.



(a) Different utilizaiton (b) Different GPU/CPU execution time (c) Different task-level power variation

Fig. 16. Temperature reduction by RT-TAS for varied utilization, GPU execution time and task-level power variations.

C EVALUATION WITH DIFFERENT TASK SETS

We conducted further experiments with different tasks sets with different characteristics. From the base task set parameters in Table 3, we vary i) utilization ii) GPU execution time, and iii) task-level power variation for each experiment setting. The RT-TAS's temperature reduction diminishes as the utilization increases (Figure 16(a)), because a task assignment can no longer avoid assigning tasks to hot CPU cores and the number of candidates decrease due to the reduced slack time for co-scheduling. 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 increases (Figure 16(b)), the temperature reduction by RT-TAS becomes more pronounced because temperature imbalance across CPU cores increases 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, maximum temperature reduction by RT-TAS increased from 1.1°C to 4.6°C on average. When the variation on task-level power dissipation is large, temperature reduction by RT-TAS becomes more pronounced. When the maximum difference between task-level power dissipations increased from 10W to 15W (Figure 16(c)), maximum temperature reduction by RT-TAS increased from 5.6°C to 11.5°C on average. Such an improvement can be interpreted as the benefit of taking task-level different power dissipations and platform's temperature imbalance into account in task-to-core assignment and scheduling.

ACKNOWLEDGMENTS

This work was supported in part by the US Office of Naval Research (ONR) under Grant No. N00014-18-1-2141, as well as the Bio & Medical Technology Development Program of the National Research Foundation (NRF) funded by the Korean government (MSIT) (No. 2017M3A9G8084463) and in part by the DGIST Start-up Fund Program of the Ministry of Science, ICT and Future Planning (2019010064).

REFERENCES

[1] 2018. *Tegra X1 Thermal Design Guide.* Technical Report TDG-08214-001. Nvidia.
[2] Rehan Ahmed, Pengcheng Huang, Max Millen, and Lothar Thiele. 2017. On the design and application of thermal isolation servers. *ACM Transactions on Embedded Computing Systems (TECS)* 16 (2017).
[3] Tarek A AlEnawy and Hakan Aydin. 2005. Energy-aware task allocation for rate monotonic scheduling. In *RTAS.*
[4] Hakan Aydin and Qi Yang. 2003. Energy-aware partitioning for multiprocessor real-time systems. In *Parallel and Distributed Processing Symposium.*
[5] Enrico Bini and Giorgio C. Buttazzo. 2005. Measuring the performance of schedulability tests. *Real-Time Systems* 30, 1–2 (2005).
[6] Thidapat Chantem, X. Sharon Hu, and Robert P. Dick. 2011. Temperature-aware scheduling and assignment for hard real-time applications on MPSoCs. *IEEE Transactions on Very Large Scale Integration Systems* 19, 10 (2011).
[7] Minki Cho, William Song, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2012. Thermal system identification (TSI): A methodology for post-silicon characterization and prediction of the transient thermal field in multicore chips. In *SEMI-THERM.*
[8] Edward G. Coffman, Gabor Galambos, Silvano Martello, and Daniele Vigo. 1999. Bin packing approximation algorithms: Combinatorial analysis. In *Handbook of Combinatorial Optimization.* 151–207.
[9] David Defour and Eric Petit. 2013. GPUburn: A system to test and mitigate GPU hardware failures. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS).*
[10] Kapil Dev and Sherief Reda. 2016. Scheduling challenges and opportunities in integrated cpu+ gpu processors. In *ESTIMedia.*
[11] Glenn A. Elliott, Bryan C. Ward, and James H. Anderson. 2013. GPUSync: A framework for real-time GPU management. In *RTSS.*
[12] Paolo Gai, Marco Di Natale, Giuseppe Lipari, Alberto Ferrari, Claudio Gabellini, and Paolo Marceca. 2003. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In *RTAS.*
[13] Sharath Kodase, Shige Wang, Zonghua Gu, and Kang G. Shin. 2003. Improving scalability of task allocation and scheduling in large distributed real-time systems using shared buffers. In *RTAS.*
[14] Pratyush Kumar and Lothar Thiele. 2011. Cool shapers: Shaping real-time tasks for improved thermal guarantees. In *DAC.*
[15] Kai Lampka and Bjorn Forsberg. 2016. Keep it slow and in time : Online DVFS with hard real-time workloads. In *DATE.*
[16] Youngmoon Lee, Hoon Sung Chwa, Kang G. Shin, and Shige Wang. 2018. Thermal-aware resource management for embedded real-time systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018).

[17] Sheng-Chih Lin and Kaustav Banerjee. 2008. Cool chips: Opportunities and implications for power and thermal management. *IEEE Trans. Dev.* 55, 1 (2008).
[18] Pratyush Patel, Iljoo Baek, Hyoseung Kim, and Ragunathan Rajkumar. 2018. Analytical enhancements and practical insights for MPCP with self-suspensions. In *RTAS*.
[19] Indrani Paul, Srilatha Manne, Manish Arora, W. Lloyd Bircher, and Sudhakar Yalamanchili. 2013. Cooperative boosting: Needy versus greedy power management. In *ISCA*.
[20] Nick Piggin. [n.d.]. "Linux CFS Scheduler". https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.
[21] Alok Prakash, Hussam Amrouch, Muhammad Shafique, Tulika Mitra, and Jörg Henkel. 2016. Improving mobile gaming performance through cooperative CPU-GPU thermal management. In *DAC*.
[22] Danil Prokhorov. 2008. *Computational Intelligence in Automotive Applications*. Vol. 132. Springer.
[23] Robert Redelmeier. [n.d.]. cpuburn. https://patrickmn.com/projects/cpuburn/.
[24] Onur Sahin, Lothar Thiele, and Ayse K. Coskun. 2018. MAESTRO: Autonomous QoS management for mobile applications under thermal constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).
[25] Gaurav Singla, Gurinderjit Kaur, Ali Unver, and Umit Ogras. 2015. Predictive dynamic thermal and power management for heterogeneous mobile platforms. In *DATE*.
[26] Kevin Skadron, Mircea Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. 2003. Temperature-aware microarchitecture. In *ISCA*.
[27] Liang Wang, Xiaohang Wang, and Terrence Mak. 2016. Adaptive routing algorithms for lifetime reliability optimization in network-on-chip. *IEEE Trans. Comput.* 65, 9 (2016).
[28] Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. 2016. TaskShuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *RTAS*.

Received April 2019; revised June 2019; accepted July 2019