Thermal-Aware Servers for Real-Time Tasks on Multi-Core GPU-Integrated Embedded Systems

Seyedmehdi Hosseinimotlagh and Hyoseung Kim
University of California, Riverside
shoss007@ucr.edu, hyoseung@ucr.edu

***Abstract*—The recent trend in real-time applications raises the demand for powerful embedded systems with GPU-CPU integrated systems-on-chips (SoCs). This increased performance, however, comes at the cost of power consumption and resulting heat dissipation. Heat conduction interferes the execution time of tasks running on adjacent CPU and GPU cores. The violation of thermal constraints causes timing unpredictability to real-time tasks due transient performance degradation or permanent system failure. In this paper, we propose a thermal-aware server framework to safely upper bound the maximum temperature of GPU-CPU integrated systems running real-time sporadic tasks. Our framework supports variants of real-time server policies for CPU and GPU cores to satisfy both thermal and timing requirements. In addition, the framework incorporates two mechanisms, miscellaneous-operation-time reservation and pre-ordered scheduling of GPU requests, which significantly reduce task response time. We present analysis to design thermal-server budget and to check the schedulability of CPU-only and GPU-using sporadic tasks. The thermal properties of our framework have been evaluated on a commercial embedded platform. Experimental results with randomly-generated tasksets demonstrate the performance characteristics of our framework with different configurations.**

I. Introduction

High temperature in embedded systems with modern systems-on-chips (SoCs) causes several major issues. An increase in temperature has a considerable impact in the growth of leakage current which leads to rise in static power consumption. An increase in power consumption levels up the device temperature, thereby resulting in again an increase in power consumption. This detrimental loop causes not only rapid battery drain but also “thermal runaway” [3]. Furthermore, studies show that operating in high temperature reduces the system reliability substantially [42]. For instance, 10°C to 15°C increase in temperature doubles the probability of failure of the underlying electronic devices [44]. Thermal violation not only reduces the reliability in real-time implantable medical devices, but also causes physical harm [6]. Therefore, bounding the maximum temperature is an important issue, especially when real-time requirements have to be satisfied.

Thermal management on today’s multi-core CPU-GPU integrated platforms with real-time requirements is a challenging problem. Dynamic Thermal Management (DTM) is triggered when thermal violation occurs so that it forces frequency throttling or shutdown of the SoC for cooling purpose. This unwanted performance degradation leads to timing unpredictability in task execution, and real-time tasks may miss their deadlines. Thermal violation avoidance in uni-processor systems has been studied extensively in the literature of real-time systems [6, 46] but it cannot be directly used for multi-core GPU-integrated devices due to heat conduction between processor units. Dynamic Voltage Frequency Scaling (DVFS) techniques to mitigate the heat and power dissipation of processors also has been widely studied in the literature [8, 46]. However, aside from a considerable reduction in system reliability over time due to continuous frequency changes [16, 24, 47], not all embedded devices support DVFS, especially for integrated GPUs.

Despite the popularity of integrated GPUs in modern multi-core SoCs, state-of-the-art approaches are incapable of simultaneously addressing thermal management and real-time schedulability issues. On the one hand, the GPU access segment of a real-time task has been modeled as a critical section to ensure schedulability [11, 12, 30]. These approaches, however, are oblivious of thermal constraints so that the system may suffer from heat dissipation and intermittent performance drops by DTM. On the other hand, there are previous studies [1, 2] introducing the concept of thermal servers for real-time uni-processor and multi-core platforms. However, their schemes are not ready to use for a multi-core SoC with an integrated GPU. The unique characteristics of GPU operations, e.g., kernel execution on the GPU and interactions between CPU and GPU cores for data transfer, introduce new challenges to thermal server design and schedulability analysis. To the best of our knowledge, there is no prior work that offers both thermal safety and real-time schedulability in multi-core GPU-integrated embedded systems.

In this paper, we propose a thermal-aware CPU-GPU framework to handle both real-time CPU-only and GPU-using tasks. Our framework enhances the notion of thermal servers to satisfy the given thermal constraint on CPU and GPU cores and to offer bounded response time to real-time tasks. The framework also introduces two mechanisms, miscellaneous-operation-time reservation and pre-ordered waiting queue for GPU requests, to reduce task response time. We will show with experimental results that our framework is effective in satisfying both thermal and temporal requirements.

Contributions. The contributions of this paper are as follows:

- We propose a thermal-aware CPU-GPU server framework for multi-core GPU-integrated real-time systems. We characterize different timing penalties and present a protocol for CPU and GPU thermal servers.

- For real-time predictability on a GPU, we propose a GPU server design with a variant of the sporadic server policy, where the GPU segments of tasks execute with no thermal violation.
- We propose an enhancement to the waiting queue of the GPU server to mitigate the pessimism of a priority-based queue.
- We introduce a *miscellaneous operation time reservation* mechanism for deferrable and sporadic CPU servers to reduce CPU-GPU handover delay and remote blocking time.
- We extensively analyze the thermal safety and task schedulability of CPU and GPU servers with various budget replenishment policies.

II. Related Work

Real-time GPU management has been studied to handle GPU requests with the goal of improving timing predictability [17, 18, 19, 49]. To guarantee the schedulability of GPU-using real-time tasks, synchronization-based approaches [11, 12, 30] that model GPU access segments as critical sections have been developed. The work in [20, 21] introduces a dedicated GPU-serving task as an alternative to the synchronization-based approaches. While these prior studies have established many fundamental aspects on real-time task schedulability with GPUs, thermal violation issues have not been considered.

There exist extensive studies on bounding the maximum temperature in real-time uni-processor systems [6, 48] and non-real-time heterogeneous multi-core systems [14, 31, 32, 36, 39]. In [6], the authors proposed a novel scheme that bounds the maximum temperature by analyzing task execution and cooling phases in uni-processor real-time systems. Real-time thermal-aware resource management for uni-processor systems has been developed with the consideration of varying ambient temperature and diverse task-level power dissipation [48]. For non-real-time heterogeneous systems, Singla et. al [39] proposed a dynamic thermal and power management algorithm to adjust the frequency of GPU and CPU as well as number of active cores by computing the power budget. Prakash et al. [32] proposed a control-theory based dynamic thermal management technique for mobile games. Gong et al. [14] presented a thermal model on a real-life heterogeneous mobile platform. In [31] and [36], the authors proposed a proactive frequency scheduling for heterogeneous mobile devices to maintain their performance. However, these studies cannot be directly applied to real-time multi-core GPU-integrated systems where the GPU is shared among tasks.

The notion of periodic thermal-aware servers was proposed in [1] for uni-processors. In this work, the optimal server utilization has been proved and the budget replenishment period is determined by a heuristic algorithm. Similar to the thermal server, the notion of cool shapers was proposed in [23] to satisfy the maximum temperature constraint by throttling task execution. The notion of hot tasks was introduced in [15] to partition lengthy tasks into several chunks to avoid continuous task execution and thermal violation while maximizing throughput. Although all of these aforementioned studies have brought valuable contributions, they have been proposed for uni-processor platforms. In contrast, our framework addresses the temperature bounding problem for real-time tasks with GPU segments running on modern CPU-GPU integrated SoCs.

Recently, the authors of [10] introduced a novel technique for periodic tasks executing on multi-core platform. This technique introduces an Energy Saving (ES) task that runs with the highest priority and captures the sleeping time of CPU cores. The technique can be seen as an alternative to a thermal server because the ES task effectively models the budget-depleted duration of a thermal server. The authors of [2] proposed thermal-isolation servers that avoid the thermal interference among tasks in temporal and spatial domains with thermal composability. These techniques, however, cannot address the challenges of scheduling GPU-using real-time tasks.

III. Background on Thermal Behavior

In this section, we briefly introduce the thermal model used in this paper. It depends on power consumption, heat dissipation, and the conductive heat transfer between adjacent power-consuming resource components, which includes CPU cores, CPU peripherals, GPU, caches, and other IPs.

Uni-processor thermal model. With respect to the power function [7, 26], the thermal model of a uni-processor is modeled as an RC circuit in the literature [7, 37, 40]. According to the Fourier's Law [45] and considering $t_0 = 0$, the temperature of a core $\theta(t)$ after t time units operating at a fixed clock frequency is given by:[1]

$$\theta(t) = \alpha + (\theta(t_0) - \alpha)e^{\beta t} \tag{1}$$

where $\alpha > 0$ and $\beta < 0$ are constants.

Most operating systems in embedded platforms transition the processor to the sleep state when there is no task execution. Therefore, the power consumption can be assumed to be negligible in such a case. The thermal function in the sleep state (*aka* cooling phase) where the frequency is switched off can be modeled as:

$$\theta(t) = \theta(t_0)e^{\beta t}. \tag{2}$$

because in the cooling phase, $\alpha = 0$.

Heterogeneous multi-core thermal model. With the presence of multiple cores and other power-consuming resources, there is heat dissipation not only to ambient but also between nodes. In this paper, we only consider CPU and GPU cores as power-consuming nodes because other IPs consume much less power than them, thereby causing negligible thermal effects.

In such a CPU-GPU integrated system with the lateral thermal conductivity between processing cores, the temperature of a core depends not only on its current temperature and power consumption, but also on those of adjacent cores. Prior work [13] showed that the temperature of each core at time $t + \Delta$ can be modeled with an acceptable accuracy as follows:

$$\Theta(t + \Delta) = A \times P(t + \Delta) + \Gamma \times \Theta(t)$$

[1]Details are available in [10] and [2].

where A is $m \times m$ matrix, and Γ, P and Θ are $m \times 1$ matrices, respectively. One can interpret the thermal model as a function of the current temperature and heat produced by execution and the thermal conductivity between cores. Hence, according to the thermal composability characteristics of heat transfer [2] with the initial temperature of $\theta(t_0)$, the temperature after t time units is given by:

$$\theta_i(t_0 + t) = \alpha + (\theta(t_0) - \alpha)e^{\beta t} + \sum_{j=1}^{m} \gamma_{ij}\theta_j(t_0 + t). \quad (3)$$

IV. System Model

In this section, we describe the thermal-aware server as well as the task models used in this paper and explain the procedure of a kernel launch on a GPU. Then, we characterize the scheduling penalties that arise from the use of a GPU with thermal-aware servers.

A. Computing platform

We consider a temperature-constrained embedded platform equipped with an integrated CPU-GPU SoC. The SoC has multiple CPU cores and one GPU core, each running at a fixed clock frequency. Note that such SoC design with a single GPU is popular in today's embedded processors, such as Samsung Exynos 5422 and NVIDIA TX2. The assumption on the fixed operating frequency is particularly suitable for the GPU as DVFS capabilities are not widely supported on embedded on-chip GPUs. The thermal behavior of CPU and GPU cores follows the model described in Section III. For simplicity, we assume that the amount of temperature generated by other SoC components, such as peripherals and caches, is either negligible or acceptably small.

B. Thermal-aware servers

We consider one thermal-aware server for each CPU and GPU core.[2] Each server is statically associated with one core and does not migrate to another core at runtime. However, unlike prior work [1, 2], we do not limit the server to follow only the polling server policy. We will show in the later section that this flexibility brings significant benefit in the schedulability of tasks accessing the GPU.

To bound the temperature of each core, its corresponding server v_i is modeled as $v_i = (C_i^v, T_i^v)$ where C_i^v is the maximum execution budget and T_i^v is the budget replenishment period of v_i. For brevity, we will use $v_g = (C^g, T^g)$ to denote the GPU server and $v_c = (C^c, T^c)$ for the CPU server. For budget replenishment policies, we consider *polling* [38], *deferrable* [43], and *sporadic servers* [41]. Under the polling server policy, the corresponding server activates periodically and executes ready tasks until its budget is depleted. The budget is fully replenished at the start of the next period. If there is no task ready, the remaining budget is immediately depleted. In contrast, under the deferrable server, any unused budget is preserved until the end of the period. Hence, a

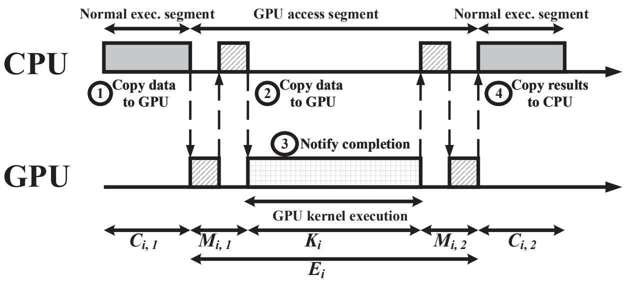


Figure 1: Task execution with a GPU segment.

task can execute at any time during the server period while the budget is available. The sporadic server also preserves remaining budget, but replenishes the budget sporadically; only the amount of budget consumed is replenished after T^v time units from the time when that budget is used. Let J^v denote the task release jitter relative to the server release. The value of J^v is T^v under the polling server policy and $T^v - C^v$ under the deferrable and sporadic server policies [4].

C. Task model

This work considers sporadic tasks with implicit deadlines under *partitioned fixed-priority preemptive scheduling*, which is widely used in many real-time systems. Each task τ_i has been statically allocated to one CPU core (thus to the server of that core) with a unique priority. Tasks are labeled in an increasing order of priority, i.e., $i < j$ implies τ_i has lower priority than τ_j. Without loss of generality, each task can contain at most one GPU segment, but it can be easily extended to multiple GPU segments. A task τ_i is modeled as $\tau_i = ((C_{i,1}, E_i, C_{i,2}), T_i, s_i)$, where $C_{i,1}$ and $C_{i,2}$ are the worst-case execution time (WCET) of the normal execution segments of task τ_i, and E_i is the worst-case time of the GPU access segment. The normal execution segments run entirely on the CPU core and the GPU segment involves GPU operations. Let T_i denote the the minimum inter-arrival time of τ_i and s_i indicate whether τ_i has a GPU segment, i.e., $s_i = 1$ means a GPU-using task. In case τ_i executes only on the CPU, s_i, E_i and $C_{i,2}$ are all zero. Thus, the accumulated sum of the WCETs of τ_i is denoted as

$$C_i = s_i \times (C_{i,1} + E_i + C_{i,2}) + (1 - s_i) \times C_{i,1}.$$

Furthermore, $V(\tau_i)$ represents the CPU server where τ_i is assigned. Tasks are considered as fully compute-intensive and independent from each other during normal segment execution. The only resource shared among tasks is the GPU and it is modeled as a critical section protected by a suspension-based mutually-exclusive lock (mutex). Note that this approach follows the well-established locking-based real-time GPU access schemes [11, 12, 30]. We will later present how pending GPU requests are queued in our proposed framework.

D. GPU execution model

The GPU has its own memory region, which is assumed to be sufficient enough for the tasks under consideration. We do not consider the concurrent execution of GPU requests from different tasks because of the resulting unpredictability

[2]Running multiple thermal servers on each CPU/GPU core is left for future work.

in kernel execution time [29, 30]. Once a task acquires the GPU lock, its GPU segment is handled through the following steps (see Fig. 1):

1) *Data Transfer to the GPU:* The task first copies data needed for the GPU computation, from CPU memory to GPU memory. This can be done by Direct Memory Access (DMA), which requires minimal CPU intervention. If the GPU uses a unified memory model, this step can be omitted.
2) *Kernel Launch:* Kernel launches on the GPU. Meanwhile, the task on the CPU side self-suspends and waits for the GPU computation to complete.
3) *Kernel Notification Signal:* The GPU signals the CPU to notify the completion of kernel execution.
4) *Data Transfer to the CPU:* The task wakes up and transfers the results from GPU memory to CPU memory.

It is worth noting that a GPU kernel *cannot self-suspend* on the GPU in the middle of execution.[3] On the other hand, since there is no CPU intervention during kernel execution, the task on the CPU side *self-suspends* to save CPU cycles. As a result, other tasks have a chance to execute or the CPU core sleeps during the kernel execution. For a task τ_i, the total time for the above four steps consists of two major parts:

- Miscellaneous operations that require CPU intervention. Let $M_{i,1}$ denote the time for data transfer before the kernel execution and $M_{i.2}$ denote that after the kernel execution.
- Pure GPU kernel operations that do not require any CPU intervention denoted as K_i.

In such aspect, the GPU segment time of a task τ_i is modeled as $E_i = M_{i,1} + K_i + M_{i,2}$. As the GPU is non-suspendable during kernel execution, the GPU server should have enough budget larger than or equal to E_i. The CPU server only needs to have budget larger than $M_{i,1}$ or $M_{i,2}$.

E. Challenges of thermal-aware servers with an integrated GPU

Aside from the blocking delays coming from the locking-based GPU access approach, e.g., local and remote blocking to acquire the GPU lock [11, 12, 30], there are other new challenges faced by thermal-aware servers in a CPU-GPU integrated system.

- *Server budget depletion*: Task execution in a server is scheduled with respect to the available budget. When the server budget is depleted, a task has to wait until the budget is replenished.
- *Mutual budget availability*: If a task τ_i issues a GPU request, both CPU and GPU servers must have enough budget to handle this request. It is worth noting that server budget needs for the CPU and GPU servers are different ($M_{i,1}$ or $M_{i,2}$ vs. E_i).
- *CPU-GPU handover delay*: Even if both servers are designed to have enough budget, each server may have to wait for the other's budget to be replenished when their interactions are needed, e.g., data transfer between the CPU and the GPU.
- *Back-to-back heat generation*: In case of the deferrable server, some tasks can use up all the budget at the end of the budget replenishment period, and at the very beginning of the next period, some other tasks can start to consume the replenished budget. This causes the server to run longer than its budget and generate heat in a back-to-back manner.

[3]Although GPU kernel preemption is available on some recent GPU architectures, e.g., Nvidia Pascal [9], to the best of our knowledge, the self-suspension of a kernel is not supported in any of today's GPU architectures.

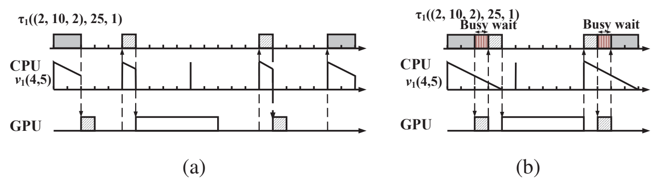


Figure 2: Example task scheduling with an unlimited GPU server budget and a CPU server following the polling server policy. a) No busy waiting: τ_1 finishes at 22. b) Busy waiting: τ_1 finishes at 14.

V. Framework Design

In this section, we present our proposed framework. We first give a protocol for CPU and GPU thermal servers, and then explain thermal server design to address the challenges discussed in the previous section. We lastly describe a *miscellaneous operation time reservation* mechanism to trade-off between server budget and GPU waiting time.

A. Thermal-aware CPU-GPU server protocol

Our framework simultaneously bounds the worst-case blocking time for GPU access and the maximum temperature of CPU and GPU cores. The thermal server designed with our framework isolates the thermal conductive effects of each compute node to other nodes. To achieve these properties, we establish the following rules in our framework.

Shared GPU server

1) Pending GPU requests are inserted to a priority queue that orders the requests based on the priorities of the corresponding tasks. This rule assures that the GPU request of a high-priority task is blocked by at most one lower-priority request in the queue.
2) To handle the GPU request of a task τ_i, there must exist at least E_i budget available on the GPU server. This rule is due to the preemptive and non-self-suspending characteristics of the GPU.
3) If there is an insufficient amount of budget to launch a GPU segment, the GPU is locked until having enough budget for that GPU segment. This rule assures that starvation does not happen and the critical section of a higher-priority task waits for just a single critical section of a lower-priority task. With these rules, remote blocking time can be bounded. Later, we will propose an enhancement to these rules.

CPU Core Server

1) Servicing a GPU request boosts the priority of the corresponding task to the highest-priority level. As a result, the normal segments of higher-priority tasks are blocked until the GPU segment completes. If there is not enough budget

on the corresponding CPU server (e.g., $M_{i,1}$) or on the GPU server (e.g., E_i), no other task can execute on the CPU or the GPU.

2) During data transmission to/from the GPU, the CPU server "busy-waits". The reasoning behind this rule is to reduce the total response time of a task. Fig. 2 illustrates task scheduling with and with or without this rule on a CPU server using the polling server policy. As one can see in this example, without this rule, it takes one additional replenishment period for transferring data to/from the CPU, because during processing the transferring request on the GPU, the CPU server has no other workload to execute; hence it deactivates until the beginning of the next replenishment period.

B. GPU server design

We now discuss budget replenishment policies for the GPU server and their implications. One may consider both the CPU and GPU servers following the polling server policy. In this case, the CPU-GPU handover delay can be at least three complete replenishment periods of a CPU server. For instance, consider a task holding a GPU lock and its kernel being executed on the GPU. Once the kernel execution completes, the GPU has to wait for the task on the CPU to transfer the results and release the lock. It is possible that at this time, the CPU server budget has been already depleted. Thus, the GPU has to wait for the next period of the CPU server, and this kind of extra delay happens for each sub-segment of the CPU segment, i.e., $M_{i,1}$, K_i and $M_{i,2}$. Moreover, since operations on the GPU are non-preemptive and non-suspendable, the GPU server budget has to be large enough that at least one entire GPU sub-segment of $M_{i,1}$, K_i or $M_{i,2}$ can complete its execution within the same period. However, having a large replenishment period woudl exacerbate the response time of GPU segments especially for small kernels.

One may consider busy waiting between GPU sub-segments so as to fill their execution gaps because such gaps may deactivate the GPU's polling server. This approach, however, not only requires over-provisioning of the GPU budget, but also produces a considerable amount of heat. In the worst case, the extra heat generation due to the busy-waiting on the GPU server may continue over two periods of the CPU server because the CPU server with the polling server policy can be deactivated between GPU sub-segments.

One may suggest the deferrable server policy for the GPU to mitigate the heat generation issue and to minimize the response time of a GPU segment. This approach, however leads to the thermal back-to-back execution phenomenon. Fig. 3a illustrates an example of possible drawbacks of the deferrable server chosen for the GPU. The first jobs of τ_1 and τ_2 arrive at the latest moments in the first GPU server period, and the second job of τ_1 arrives at the beginning of the second GPU server period. Although the tasks are schedulable by the given server budgets, it causes burst heat generation by back-to-back execution, which can potentially lead to thermal violation. In order to avoid this, the budget of the deferrable

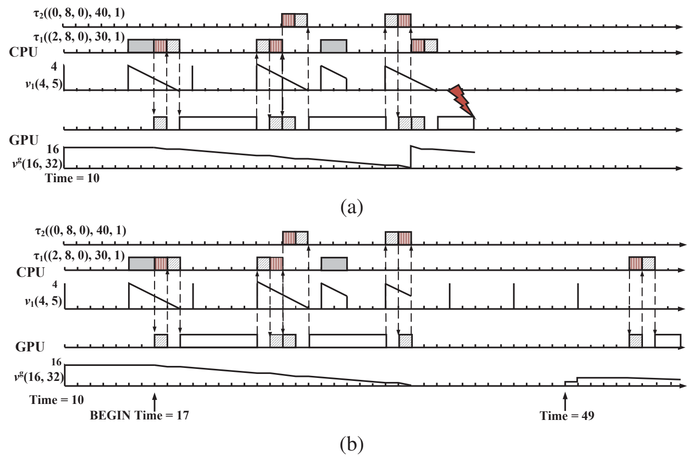


Figure 3: Example task scheduling with the GPU server under a) the deferrable server policy and b) the sporadic server policy. For both tasks τ_1 and τ_2, $E_i = 8$ and $M_{i,1} = M_{i,2} = 2$.

server for the GPU has to be halved to avoid thermal violation but it can drastically lower task schedulability.

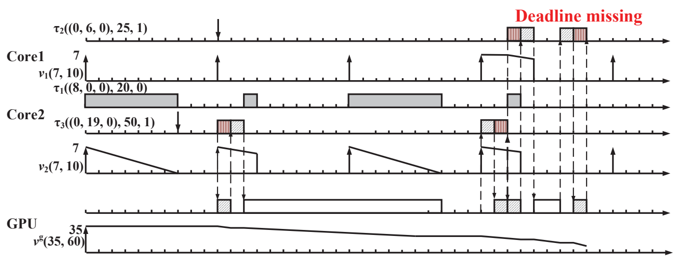
In contrast, the sporadic server policy can take the merits of both the polling server and deferrable server policies. If the sporadic server policy is used for the GPU, a GPU segment can execute at any time as long as there is enough budget, and back-to-back heat generation does not occur. Fig. 3b illustrates the previous example with the sporadic server on the GPU. As can be seen, unlike the deferrable server case, the budget replenishment of the GPU server is one period apart from its consumption time, thereby preventing potential thermal violation. The sporadic server on the GPU is also practically effective because the GPU server needs to have a relatively large budget with a long replenishment period due to its non-preemptive nature whereas the CPU server typically has a short replenishment period to reduce task response time.

In summary, due to the aforementioned reasons, our framework specifically uses the sporadic server policy for the GPU, while all the three policies (polling, deferrable, and sporadic) are allowed for CPU cores. We also set the budget of the GPU sporadic server to be at least as large as one complete GPU segment of any task, i.e., $\max(E_i)$, because this can reduce the response time of a GPU segment and remote blocking time. The detailed analysis on these delays will be presented in Section VI.

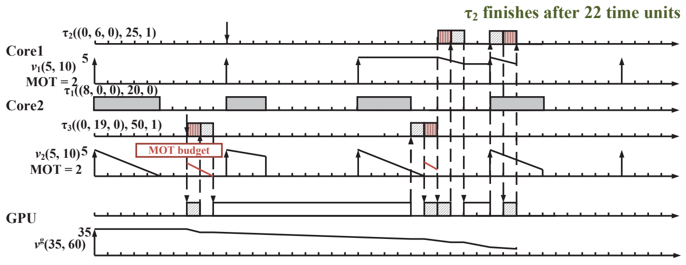
The thermal server for the GPU is a resource abstraction managed on the CPU side, similar to other real-time GPU management schemes [11, 12, 20, 21, 30]. One can implement the GPU thermal server as part of GPU drivers or application-level APIs.

C. Miscellaneous operation time (MOT) reservation

In order to reduce the CPU-GPU handover delay, we propose an MOT reservation mechanism. With this mechanism, a small portion of the CPU server budget is reserved only for miscellaneous operations in a GPU segment, e.g., transferring data to/from the GPU. The MOT reservation is feasible with the deferrable and sporadic server policies but not with the polling server policy because the polling server is unable to



(a)



(b)

Figure 4: Data transfer in a GPU request a) without and b) with the MOT reservation mechanism.

keep unused budget by design. Although this MOT reservation reduces the amount of CPU budget for regular task execution, it guarantees that the GPU does not need to wait for the budget replenishment of the CPU server during the data transmission phase of a GPU segment. It can also reduce the remote blocking time of other tasks. This reserved budget for MOT has to be the largest amount of the CPU intervention time in all GPU requests. The trade off between the MOT reservation and the reduced budget for regular task execution will be extensively investigated in the evaluation.

Fig. 4 illustrates an example to highlight the benefit of the MOT reservation mechanism. As one can see in Fig. 4a, τ_3 has to wait until 10 to transfer its data to the GPU because τ_1 already has consumed all budget of v_2. Similarly, although the result of the GPU kernel of τ_3 is ready at 27, its result begins to be transferred at 30. These delays cause the remote blocking of τ_2 lasts for 22 time units although there exists an available amount of the server budget on v_1. Designating 2 time units as the MOT budget (see Fig. 4b) leads τ_3 to transfer its data to the GPU at 7 and finishes its kernel launch at 27; hence τ_2 initiates its kernel launch accordingly. Consequently, designating some amount of the server budget as MOT reservation leads to reduction in the remote blocking of a GPU-using task from other CPU core.

VI. Thermal and Schedulability Analysis

In this section, we present the schedulability analysis of our framework. We first design our thermal-aware servers for multi-core GPU-integrated platforms to avoid thermal violation. Then, we analyze task schedulability with and without the MOT reservation mechanism.

A. Design of server budget

In our framework, task execution is performed within thermal-aware servers. As discussed, the notion of servers is to isolate each compute node from others in terms of the thermal aspect. Accordingly, under any circumstance of task execution, the thermal violation avoidance has to be guaranteed in the server design. The specifications of servers are independent of their running tasks (except for the design of the MOT reservation), whereas task schedulability does depend on them. In our proposed framework, introducing the thermal-aware servers for both the CPU and the GPU makes the physical characteristics of underlying platforms be transparent of the task schedulability test.

1) Single-core platforms: First, we calculate the "maximum" budget that a server can have while limiting the temperature not to exceed the given thermal constraint in a single-core platform under a given replenishment period. In the worst case of a polling server, the server exhausts all of its budget at the beginning of its period and then sleeps until the beginning of the next replenishment period. Let t_{wk} and t_{slp} denote the active time (i.e., the budget-consuming phase) and the sleeping time (i.e., the cooling phase) of a CPU core server, respectively. Hence, the server period T is

$$T = t_{wk} + t_{slp}. \tag{4}$$

In the steady state of the system, we are interested in bounding the server's maximum temperature. According to Eq. 1,

$$\alpha + (\theta_s - \alpha)e^{\beta t_{wk}} \leq \theta_M$$

where θ_s and θ_M are the steady state temperature and the thermal constraint, respectively. Therefore,

$$e^{\beta t_{wk}} \geq \frac{\theta_M - \alpha}{\theta_s - \alpha} \Longrightarrow t_{wk} \leq \frac{1}{\beta} \ln \frac{\theta_M - \alpha}{\theta_s - \alpha}. \tag{5}$$

On the other hand, in the cooling phase, according to Eq. 2 to respect the steady state, $\theta_M e^{\beta t_{slp}} = \theta_s$. Hence,

$$t_{slp} = \frac{1}{\beta} \ln \frac{\theta_s}{\theta_M}. \tag{6}$$

By substituting Eqs. 5 and 6 by Eq. 4, we have

$$\frac{1}{\beta} \ln \frac{\theta_M - \alpha}{\theta_s - \alpha} + \frac{1}{\beta} \ln \frac{\theta_s}{\theta_M} \leq T$$

$$\frac{\theta_M - \alpha}{\theta_s - \alpha} \times \frac{\theta_s}{\theta_M} \leq e^{\beta T}.$$

Therefore, the worst-case steady state temperature at the beginning of each period is

$$\theta_s = \frac{\alpha \theta_M e^{\beta T}}{\theta_M (e^{\beta T} - 1) + \alpha}. \tag{7}$$

Accordingly, the maximum budget for the period T is

$$t_{wk} = T - \frac{1}{\beta} \ln \frac{\theta_s}{\theta_M}. \tag{8}$$

Consequently, for a given replenishment period, a server $v_i = (T - \frac{1}{\beta} \ln \frac{\theta_s}{\theta_M}, T)$ can bound the maximum temperature to θ_M.

As one can figure out from the analysis, the maximum budget converges because of α. This means that after some point, an increase in the replenishment period has no effect on the maximum feasible budget. As discussed earlier through our framework design, the budget has to be considered as $\frac{t_{wk}}{2}$ for a deferrable server due to the *thermal back-to-back* phenomenon. This phenomenon does not occur in a sporadic server, and it can use the computed budget as is.

2) Homogeneous multi-core platforms: The worst case for the budget of polling servers on a multi-core CPU happens when all of them exhaust their budget completely. Therefore, according to Eq. 3 for the composability characteristics of the heat transfer, we have

$$\alpha + (\theta_s - \alpha)e^{\beta t_{wk}} + \sum_{\substack{j=1 \\ j\neq i}}^{m} \gamma_{i,j}\theta_M^j \leq \theta_M^i$$

where θ_M^i is the maximum temperature for the ith node. In the worst case, every core may reach its maximum temperature at the same time. Hence,

$$\underbrace{(1 + \sum_{\substack{j=1 \\ j\neq i}}^{m} \gamma_{i,j})}_{\lambda_i}[\alpha + (\theta_s - \alpha)e^{\beta t_{wk}}] \leq \theta_M^i.$$

However, the geographic location of cores on the chip results in different values of the conduction coefficients although it remains symmetric (i.e., $\gamma_{i,j} = \gamma_{j,i}$). Denoting $\lambda = \max_{1\leq i\leq m} \lambda_i$, θ_M is given by $\theta_M = \lambda[\alpha + (\theta_s - \alpha)e^{\beta t_{wk}}]$. Similar to the single-core analysis given in the previous subsection, the steady state temperature is

$$\theta_s = \frac{\alpha \frac{\theta_M}{\lambda} e^{\beta T}}{\frac{\theta_M}{\lambda}(e^{\beta T} - 1) + \alpha} \tag{9}$$

Therefore, the server budget for each compute node i is

$$C^c = t_{wk} = T - \frac{1}{\beta} \ln \frac{\theta_s \times \lambda}{\theta_M}. \tag{10}$$

3) Heterogeneous multi-core GPU-integrated platforms: Hereby, we will determine the budget of servers in the presence of an integrated GPU. Similar to the homogenous multi-core platform, the worst case happens when all servers exhaust their budgets completely. There is also a GPU segment execution on the GPU which causes extra heat dissipation. Since the GPU runs at a different frequency and its architecture is different from the CPU cores, its heat generation parameters differ from those of the CPU cores. Hence,

$$\begin{cases} \theta_M^i = \lambda_i[\alpha + (\theta_s - \alpha)e^{\beta t_{wk}}] + \gamma_{i,g}[\alpha^g + (\theta_s^g - \alpha^g)e^{\beta^g t_{wk}^g}] \\ \theta_M^g = \alpha^g + (\theta_s^g - \alpha^g)e^{\beta^g t_{wk}^g} + \underbrace{\sum_{j=1}^{m} \gamma_{g,j}}_{\gamma^g}[\alpha + (\theta_s - \alpha)e^{\beta t_{wk}}] \end{cases}$$

$$\Longrightarrow \begin{cases} \lambda\theta_M^i e^{\beta t_{slp}} + \gamma_{i,g}\theta_M^g e^{\beta^g t_{slp}^g} = \theta_s \\ \theta_M^g e^{\beta^g t_{slp}^g} + \gamma^g \theta_M^i e^{\beta t_{slp}} = \theta_s^g \end{cases} \tag{11}$$

where the symbols with a superscript g represent the corresponding parameters of the GPU.

4) Miscellaneous operation time reservation: To reduce the remote locking time, some portion of the CPU server budget can be reserved for data transferring from/to the GPU that needs CPU intervention. The MOT budget has to be large enough to handle the longest data transferring time, therefore

$$C^c = t_{wk} - \max_{\forall\tau_i}(M_{i,1}, M_{i,2}) \tag{12}$$

It is noteworthy that acquiring a lock on the GPU happens when there is enough budget on the GPU server to execute the whole GPU request; hence, no budget reservation is needed on the GPU side.

B. *Task schedulability analysis*

The thermal analysis in the previous subsection gives the maximum budget of each CPU/GPU server that satisfies the thermal constraint of the system. Hereby, we present the schedulability analysis of a task τ_i in our framework.

Before introducing our analysis, we review the existing response time test for independent tasks with no thermal constraints and no shared GPU under hierarchical scheduling [35], which is

$$W_i^{n+1} = C_i + \sum_{\substack{\tau_h \in V(\tau_i) \\ h>i}} \left\lceil \frac{W_i^n + J^c}{T_h} \right\rceil C_h + \left\lceil \frac{W^n + C^c}{T^c} \right\rceil (T^c - C^c) \tag{13}$$

where J^c is the jitter of a task running in a server (see Section IV) and $W^0 = C_i$. The recursion terminates successfully when $W^{n+1} = W^n$ and fails when $W^{n+1} > T_i$. This equation considers the budget depletion of a CPU server. The first term is the amount of CPU time used by the task τ_i, the second term captures the back-to-back execution of each higher-priority task τ_h, and the third term captures the amount of interference that the server can generate due to the periodic budget replenishment. However, this equation cannot be used directly for the thermal constraint problem with a shared GPU.

Our analysis extends the existing response time test by considering the factors discussed in Section V: (i) local blocking time, (ii) remote blocking time, (iii) back-to-back execution due to remote blocking, (iv) mutual budget availability, (v) CPU-GPU handover delay, (vi) multiple priority inversions, and (vii) CPU and GPU server budget co-depletion. We take into account the factors (iv) and (v) together in the analysis of the handover delay, the factor (vi) as part of the local blocking time analysis, and the factor (vii) as part of the remote blocking time analysis. By considering all these factors, the following recurrence equation bounds the worst-case response time of a task τ_i:

$$\begin{aligned} W_i^{n+1} &= C_i + B_i^l + B_i^r + H_i^{gc} + \\ &\left\lceil \frac{W_i^n + C^c - s_i(H_i^{gc} + K_i)}{T^c} \right\rceil (T^c - C^c) + \\ &\sum_{\substack{\tau_h \in V(\tau_i) \\ h>i}} \left\lceil \frac{W_i^n + J^c + (W_h - C_h) - s_i(H_i^{gc} + E_i)}{T_h} \right\rceil C_h \end{aligned} \tag{14}$$

where C_i is the worst-case execution time of τ_i, B_i^l is the local blocking time, B_i^r is the remote blocking time, H_i^{gc} is the CPU-GPU handover delay. We will later discuss each of them in details. The recurrence equation terminates when $W_i^{n+1} = W_i^n$, and the task τ_i is schedulable if its response time does not exceed its implicit deadline (i.e., $W_i^n <= T_i$).

The second line of the equation captures the delay due to the server budget depletion on the CPU side (an extension of the last term of Eq. 13). It is worth noting that during the pure GPU kernel execution of τ_i (K_i), no CPU budget is consumed by τ_i. Any other tasks can execute on the CPU core if there is a remaining budget. The task τ_i only consumes the CPU server budget when it executes normal segments or the miscellaneous operations (e.g. data copy) of GPU segments. Therefore, H_i^{gc} and K_i, which already exist in W_i^n for a GPU-using task τ_i, are excluded such that only the CPU-consuming parts of this task is affected by the CPU server budget depletion. Any additional delay due to CPU budget replenishments during GPU segment execution will be captured by H_i^{gc}.

The last line captures the preemption time by the normal execution segments of higher-priority tasks (an extension of the second term of Eq. 13). The fact that a higher-priority task τ_h can only preempt the normal execution segments of τ_i leads to the deduction of H_i^{gc} and E_i from W_i^n. There can be at most one additional job of τ_h that has arrived during τ_i's GPU segment execution and interfere τ_i's normal segments. This holds true if τ_h is schedulable, i.e., $W_h \leq T_h$, because other arrivals of τ_h during τ_i's GPU segment will finish their executions while τ_i is self-suspended for its kernel execution on the GPU. The interference from this additional carry-in job of τ_h is taken into account by modeling it as a dynamic self-suspending model and adding $W_h - C_h$ to W_i^n [5].

Now, we present the detailed analysis of the delay factors used in our response time test.

*1) **CPU-GPU handover delay**:* It captures an extra delay a task can experience after acquiring the GPU lock because of the factors (iv) and (v). Fig. 5 illustrates this type of delay decomposed into three parts when the polling server policy is used for a CPU server. ① When there is not enough budget available in the GPU server for the execution of a GPU segment, the task has to wait at most the jitter of the GPU server ($T^g - C^g$). ② After the GPU budget is replenished, if the CPU server is inactive, the task has to wait for additional T^c time units for the next period of the CPU server. ③ After the completion of kernel execution on the GPU, at most T^c time units are needed for the CPU server to be activated in order to transfer the results back from the GPU to the CPU. Hence,

$$H_i^{gc} = s_i(T^g - C^g + 2T^c). \tag{15}$$

For a CPU server with the deferrable and sporadic server policies, ② and ③ change to the jitter of the CPU server (i.e., $T^c - C^c$) because a GPU segment needs to wait for the replenishment of its corresponding CPU server's budget. The handover delay is then given by

$$H_i^{gc} = s_i[T^g - C^g + 2(T^c - C^c)]. \tag{16}$$

*2) **Local blocking**:* It occurs when a task τ_i is blocked by the lower-priority task on the same core. As in the case of MPCP [33, 34], a task can be blocked by each of its lower-priority task τ_l's GPU segment at most once due to the priority boosting of our framework. To obtain a tight bound, we analyze local blocking time from two different perspectives.

On the one hand, the worst-case local blocking time of a task τ_i happens when each normal execution segment of τ_i is blocked by the GPU segment of each lower-priority

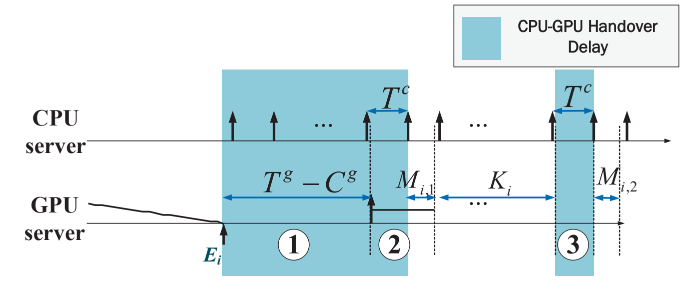


Figure 5: The worst-case scenario for CPU-GPU handover delay with a CPU polling server. When a GPU request with E_i is chosen from the GPU waiting queue for execution, it experiences the delay highlighted in blue boxes.

task τ_l with the amount of $E_l - K_l = M_{i,1} + M_{i,2}$, which is the maximum CPU time used by the GPU segment of τ_l. Hence, the total local blocking time of a task τ_i is $(1 + s_i)\sum_{l<i \ \& \ \tau_l \in V(\tau_i) \ \& \ s_i>0} E_l - K_l$, where $(1 + s_i)$ indicates the number of normal execution segments of τ_i. It is worth noting that under the RM policy, the total blocking time can be bounded by just $\sum_{l<i \ \& \ \tau_l \in V(\tau_i) \ \& \ s_i>0} E_l - K_l$ because each task has only one GPU segment and the period of any lower-priority task τ_l is larger than that of τ_i, which leads to only one blocking time from each τ_l during the job execution of τ_i.

On the other hand, the worst-case local blocking time of τ_i can also be bounded by the amount of GPU budget available during one period of τ_i. The reasoning behind this approach is that lower-priority tasks cannot execute GPU segments more than the available budget on the GPU. Thus, the maximum total blocking time of τ_i is bounded by $(1 + s_i)(\lceil \frac{T_i}{T^g} \rceil + 1)C^g$ where "+1" is due to the carry-in effect.

Using these two approaches, the total local blocking time of τ_i is bounded by

$$B_i^l = (s_i + 1) \cdot \min\left(\left(\left\lceil \frac{T_i}{T^g} \right\rceil + 1 \right) C^g, \sum_{\substack{l<i \\ \tau_l \in V(\tau_i) \\ s_i>0}} E_l - K_l \right). \tag{17}$$

*3) **Remote blocking**:* It occurs when the GPU segment of a task is blocked in the GPU waiting queue due to other GPU requests. Recall that the GPU segments of tasks are ordered in a priority queue according to their tasks' original priorities. The response time of a GPU segment of τ_i is given by $W_i' = H_i^{gc} + E_i$. The reasoning is that after a task acquires the GPU lock, it has to wait H_i^{gc} for the handover delay of data transferring and mutual server synchronization (Eq. 15). There is no other delay than H_i^{gc} added to the GPU segment length E_i because our framework sets the GPU budget to be large enough to perform any GPU segment in one GPU period and boosts the priority of the task executing a GPU segment. Hence, the remote blocking time of τ_i is bounded by the following recurrence equation:

$$B_i^{r,n+1} = \max_{\substack{l<i \\ s_i>0}} W_l' + \sum_{\substack{h>i \\ s_i>0}} \left(\left\lceil \frac{B_i^{r,n}}{T_h} \right\rceil + 1 \right) . W_h'. \tag{18}$$

where the base is $B_i^{r,0} = \max_{\substack{l<i \\ s_i>0}} W'_l$. The first term of the equation captures the waiting time for acquiring the GPU lock due to the currently-running of one GPU segment of a lower-priority task and the second term represents the waiting time for the GPU segments of higher-priority tasks. This analysis is pessimistic because it assumes that the GPU budget is exhausted after the completion of each GPU segment and the GPU segment of the next task has to wait for the amount of H^{gc}. It is noteworthy that because of the non-preemptive GPU resource, the GPU server budget has to be large enough that the GPU segment of any task is able to execute without interruption which leads to enormous remote blocking time due to a considerable data handover delay. Let Γ^g denote the set of GPU-using tasks in a taskset. For the lowest-priority GPU-using task, $|\Gamma^g - 1| \times (T^g - C^g)$ is the amount of delay only from the GPU server budget replenishment of higher-priority GPU-using tasks. To mitigate this issue, we will present another design of the waiting queue for GPU requests in the later part of this section.

C. Improvement

*1) **Miscellaneous operation time reservation policy**:* Recall our MOT reservation mechanism presented in Section V-C. When enabled, it ensures that there is always an enough amount of budget for miscellaneous operations (e.g. data copy from/to GPU); thus, the GPU does not have to wait until the start of the next CPU budget replenishment. Hence, the CPU-GPU handover delay with the MOT mechanism is

$$H_i^{gc} = s_i(T^g - C^g). \tag{19}$$

The improvement in the handover delay has also a profound impact on the remote blocking delay by reducing the worst-case response time of a GPU segment. However, it is worth noting that for deferrable CPU servers, their budget needs to be halved as discussed earlier in Section VI-A to avoid the thermal back-to-back phenomenon.

*2) **Remote blocking enhancement**:* To address the problem of enormous remote blocking time due to CPU-GPU handover delay, we propose an alternative approach to the GPU waiting queue. This approach implements the queue based on a variant of the first-come first-served (FCFS) policy with a pre-defined bin-packing order. To be more precise, a bin-packing heuristic is employed to determine the number of bins, where the size of each bin is the GPU budget and the length of a GPU segment is the size of an item to be packed. The total number of items in the bins is $|\Gamma^g|$ and the number of bins is related to the waiting time for a GPU segment. The reason for employing the FCFS policy is to avoid starvation of jobs with large period because under this policy, jobs with shorter period get serviced only once at any time. If a small-period job arrives meanwhile, it waits until the rest of waiting jobs get serviced and after finishing all other jobs, it gets serviced according to its position in the bins. Since in this approach all tasks have the same amount of waiting time, it leads to a significant reduction in the waiting time of low-priority GPU-using tasks but a moderate increase in that of high-priority ones. It is worth reminding that the replenishment period of the GPU server is typically much larger than that of the CPU server. The remote blocking time for a GPU-using task τ_i under the polling server policy for CPU cores is

$$B_i^r = s_i \left[(|bins| + 1)T^g + 2(|\Gamma^g| - 1)T^c \right]. \tag{20}$$

In this approach, missing activation points of the CPU polling server can still happen in the transferring time of data from/to the CPU server and due to the FCFS characteristics of the queue, the total amount is $2(|\Gamma^g| - 1)T^c$. The remote blocking time of τ_i under the deferrable and sporadic CPU server policies without the MOT mechanism is

$$B_i^r = s_i \left(|bins| + 1 \right) T^g + 2(|\Gamma^g| - 1)(T^c - C^c). \tag{21}$$

This is because in the worst case, the GPU server waits for the amount of CPU server jitter. The remote blocking time with the MOT mechanism is

$$B_i^r = s_i \left(|bins| + 1 \right) T^g. \tag{22}$$

To this end, our framework takes a *hybrid* scheduling scheme that chooses one of the proposed queue implementations which successfully passes the schedulability analysis.

VII. Evaluation

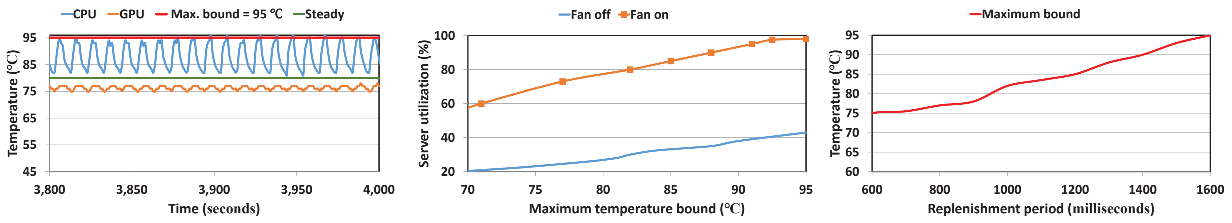
This section gives the experimental evaluation of our framework. First, we explain our implementation on a real platform. Then, we explore the impact of proposed approaches on task schedulability with randomly-generated tasksets based on practical parameters.

A. Implementation

We did our experiments on an ODroid-XU4 development board [28] equipped with a Samsung Exynos5422 SoC. There exist two different CPU clusters of little Cortex-A7 and big Cortex-A15 cores, where each cluster consists of four homogeneous cores. There exists an integrated Mali-T628 GPU on the chip which supports OpenCL 1.1. Built-in sensors with sampling rate of 10 Hz with the precision of 1° C are on each big CPU core and also the GPU to measure the temperature[4]. The DTM throttles the frequency of the big CPU cluster to 900 MHz when one of its cores reaches the pre-defined maximum temperature. During experiments, the CPU fan is always either turned off or on at its maximum speed and the CPU is set to run at its maximum frequency.

We stressed the CPU cores of the big cluster and the GPU with different settings by executing `sgemm` program of the Mali SDK benchmark [27] to measure the system parameters used in Section VI. We observed that without launching any kernel on the GPU, because of the heat conduction, the temperature on the GPU rises from 40° C (the ambient temperature) to 70° C. However, the GPU has less thermal effect on CPU cores due to the low heat dissipation. The kernel execution on the GPU in the presence of CPU workloads raises the CPU temperature by 5-10° C.

[4]There are no temperature sensors on little cores since the power consumption and heat generation of the little cluster is considerably low.



(a) (b) (c)

Figure 6: a) Server design with the given maximum temperature of 95° C when the CPU fan is off. b) Server utilization w.r.t the given maximum temperature. c) The maximum observed temperature w.r.t the server replenishment period when the CPU fan is off and CPU utilization is 30%.

Fig. 6a illustrates the result of the implementation of the CPU polling server with the replenishment period of 1 second[5] and the maximum temperature bound of 95° C when the CPU fan is off. As one can see, the CPU temperature oscillates between 78° C to 95° C after a long time of the system operating in the steady state.

Figures 6b depicts the server utilization with respect to the maximum temperature bound. The maximum achievable utilization of the CPU server is only 43% when the fan is off and almost 95% when the fan is on. The gap in server utilization between these two cases (fan on/off) decreases as the value of the maximum temperature bound reduces. The steady state temperatures under different maximum temperature bounds remain almost the same regardless of whether the fan is on/off.

With our proposed thermal-aware server design, it is possible to bound the operating temperature to any thermal constraint. It is worth noting that the same server utilization value can give different temperature bounds depending on the value of replenishment period used. Fig. 6d shows the temperature bounds at the invariant server utilization of 30% and the replenishment period in the range of $[600, 1600]$ milliseconds. The length of 1600 milliseconds is the maximum feasible server replenishment period at the server utilization of 30% that the board can reach when the fan is off. This is because with a larger period value, the CPU exceeds its maximum temperature bound during the active phase.

With these results, we have shown that it is possible to satisfy the maximum temperature constraint based on our proposed server design. Next, we will use the measured parameters in our analysis and discuss its effect in taskset schedulability.

B. *Schedulability Experiments*

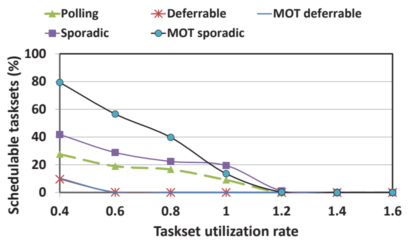
Task Generation. We randomly generate 10,000 tasksets for each experimental setting. The base parameters given in Table I and the measured parameters from the board are used for the taskset generation and the server design, respectively. It is worth noting that the GPU parameters are in compliance with the case study of prior work [18, 19, 20, 22]. Server budgets are determined according to the maximum temperature need by applying the equations of Section VI-A and the measured system parameters. The number of tasks in each taskset is determined based of the uniform distribution in the range of [8, 20]. Then, the utilization of the taskset is partitioned randomly for these tasks in a way that no task has the utilization more than the CPU server utilization. The total WCET of each task (i.e., C_i) is calculated based on the task's utilization and its randomly-chosen period. If the task τ_i is a CPU-only task, the whole C_i is assigned to $C_{i,1}$ otherwise C_i is divided into E_i, $C_{i,1}$ and $C_{i,2}$, according to the random ratio of the GPU segment length to the normal WCET. In this phase, if E_i is more than the GPU server budget, another random ratio is generated. Then, E_i is partitioned randomly into the miscellaneous time ($M_{i,1}+M_{i,2}$) and the pure kernel execution time (k_i) according to the ratio of miscellaneous operations given in Table I. The accumulated miscellaneous-operation time is randomly divided into $M_{i,1}$ and $M_{i,2}$. Finally, tasks are assigned to CPU cores by using the worst-fit decreasing (WFD) heuristic for load balancing across cores. When the MOT reservation is used, the MOT budget is determined by the maximum of $M_{i,1}$ and $M_{i,2}$ for all tasks.

Table I: Base parameters for taskset generation

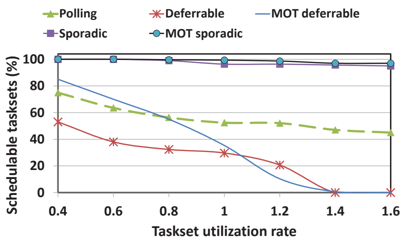
Parameters	Values
Number of CPU cores	4
Number of tasks	[8, 20]
Taskset utilization	[0.4, 1.6]
Task period and deadline	[30, 500] ms
Percentage of GPU-using tasks	[10, 30] %
Ratio of GPU segment len. to normal WCET	[2, 3]:1
Ratio of misc. operations in GPU segment $\frac{M_{i,1}+M_{i,2}}{E_i}$	[10, 20]%
Server Period	10 ms
GPU Period	20 ms

Results. Figures 7a-b depict the percentage of the schedulable tasksets when the CPU fan is on or off with different taskset utilization. The CPU sporadic servers outperform the other CPU server policies as expected because their replenishment budget are as large as the polling server and the remote blocking and CPU-GPU handover delays are as low as those in the deferrable server. Compared to the polling server, the deferrable server with MOT yields a higher percentage of schedulable tasksets especially when taskset utilization is low. Designating some portion of CPU server budget under the deferrable replenishment policy results in improvement in taskset schedulability. However, the percentage of schedulable taskset under the deferrable server drops sharply as taskset

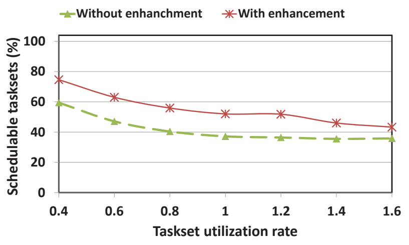
[5]We conducted the experiment with large values of replenishment period because of the coarse granularity of the sampling rate of the on-board temperature sensors.



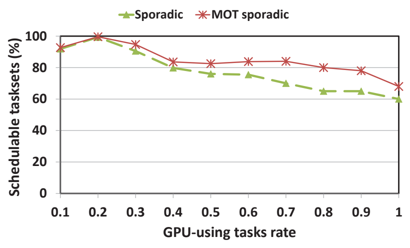
(a)



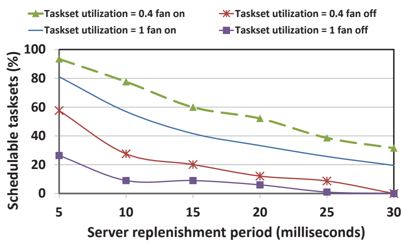
(b)



(c)



(d)



(e)

Figure 7: a) and b) The percentage of schedulable tasksets under the given maximum temperature constraint of 95° C when the CPU fan is off and on, respectively. c) Taskset schedulability results with and without the remote blocking enhancement under the polling server policy while the CPU fan is off. d) The percentage of schedulable tasksets w.r.t the ratio of GPU-using tasks.

utilization increases because of the insufficient amount of server budget (which is just the half of the polling/sporadic server budget). It is worth noting that when the CPU fan is off, there is a large gap between the two sporadic servers with and without MOT at low taskset utilization. This is due to the advantage of the MOT mechanism that also reduces enhanced remote blocking delay.

Fig. 7c shows the effect of the proposed remote blocking enhancement under the polling policy. As one can see, the remote blocking reduces substantially due to our proposed remote blocking enhancement by up to 20% especially when there exists less workload in the system. Fig. 7d depicts the impact of the rate of GPU-using tasks on schedulable taskset rate under the CPU sporadic server policy with and without MOT. As the number of GPU-using tasks increases, taskset schedulability goes decreases as more tasks contend for the shared GPU.

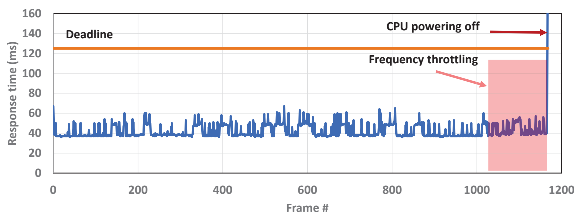
Next, we investigate the effect of server periods on the schedulability rate. In this experiment, the temperature constraint is fixed to the maximum level and the server period is varied from 5 to 30 milliseconds (see Fig. 7e) under the polling server policy. As expected, because of the large CPU-GPU handover delay, the percentage of schedulable tasksets drops significantly as the server replenishment period increases.

VIII. Case study

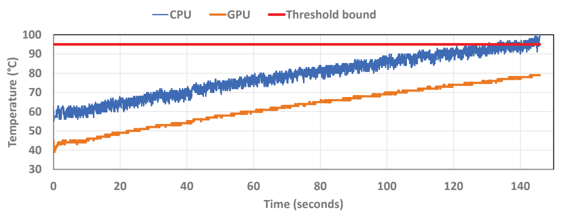
We have implemented a prototype of our framework and conducted a case study on ODroid-XU4. We show that without our framework, a real-time task experiences unexpectedly large delay due to the thermal violation, but with our framework, the operating temperature is safely bounded within a desired range.

In the case study, we used three types of applications: a real-time GPU-using task, and non-real-time CPU-only and GPU-using tasks. The non-real-time CPU-only application is run on each of the four big CPU cores with the lowest priority. The non-real-time GPU-using matrix multiplication task is run on one big CPU core with the medium priority. This task randomly generates two matrices and performs the matrix multiplication repetitively using the GPU-accelerated Mali OpenCL library. The size of matrix is set to 512×512 in order not to cause unnecessarily long waiting time to the real-time task. On the other big CPU core, the highway workzone recognition application for autonomous driving [25] is run as the real-time GPU-using task with the highest priority. This task is configured to process 8 frames per second and has one GPU segment based on OpenCL. A video consisting of around 800 frames is given as input to this task. To avoid unexpected delay in data fetching, video frames are preloaded into memory as a vector during the initialization of the task and the loaded frames are repeatedly processed at runtime. After the initialization, all tasks are signaled to start their execution together and the CPU fan is turned off during the experiment. Other tasks in the system, including system maintenance and monitoring processes, are assigned to the little cluster cores. The thermal-aware servers are implemented based on the polling server policy with the budget replenishment period of 10 milliseconds.

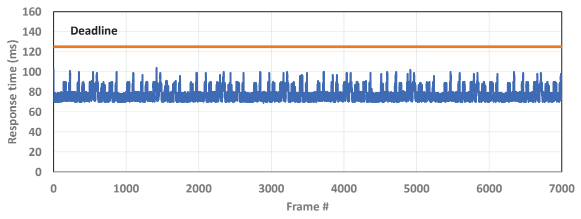
Two scenarios are used to show the effectiveness of our proposed framework. The first one is the baseline system with no thermal-aware server. Fig. 8a shows the response time of each frame (job) of the real-time workzone recognition task, and Fig. 8b depicts the temperature measurements during the experiment. As one can see, after processing around 1040 frames, the DTM was triggered and it started throttling the CPU frequency from 2.0 GHz to 900 MHz since the CPU temperature had reached the threshold of 95° C. However,



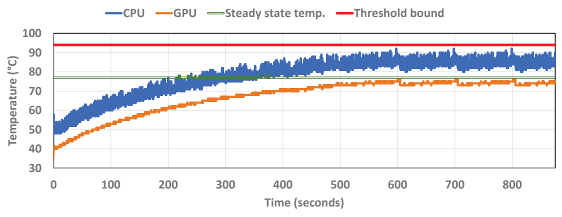
(a)



(b)



(c)



(d)

Figure 8: a) Response time of the real-time workzone recognition task without our framework. b) Temperature of CPU and GPU without our framework. c) Response time of the real-time task with our framework d) Temperature of CPU and GPU with our framework.

the frequency throttling did prevent the operating temperature from rising on both the CPU and the GPU, which caused the OS to power off the big CPU cluster temporarily. This experiment took less than three minutes. In the second scenario, all tasks execute within the thermal-aware servers. As illustrated in Fig. 8c, the observed response time of each frame was larger compared to the first scenario due to the budget of servers, but all frames were processed before the deadline. Fig. 8d depicts that it took around 500 seconds to reach the steady state temperature. Moreover, the operating temperature was tightly bounded by the thermal threshold in any circumstance. This result shows the thermal safety and accuracy of our framework in practical settings.

IX. Conclusion

In this paper, we proposed a novel thermal-aware framework to bound the maximum temperature of CPU cores as well as an integrated GPU while guaranteeing real-time schedulability. Our framework supports various server policies and provides analytical foundations to check both thermal and temporal safety. Experimental results show that each CPU server policy provided by our framework is effective in bounding the maximum temperature. We proposed the miscellaneous operation time reservation mechanism for the CPU servers in order to improve task schedulability by reducing the CPU-GPU handover delay. We also introduced a remote blocking enhancement technique that employs the bin-packing strategy to reduce the remote blocking caused by other tasks.

As our future work, we will study task allocation to CPU and GPU cores to further increase the schedulability for a given temperature constraint. Also, the thermal behavior of GPU-using tasks may depend significantly on the type of resources used by their kernels. For instance, a GPU kernel frequently accessing local memory may generate much less heat than those using global memory or being compute intensive. We plan to investigate such issues in the future.

References

[1] M. Ahmed, N. Fisher, S. Wang, and P. Hettiarachchi. Minimizing peak temperature in embedded real-time systems via thermal-aware periodic resources. *Sustainable Computing: Informatics and Systems*, 1(3):226 – 240, 2011.

[2] R. Ahmed, P. Huang, M. Millen, and L. Thiele. On the design and application of thermal isolation servers. *ACM Trans. Embed. Comput. Syst.*, 16(5s):165:1–165:19, Sept. 2017.

[3] R. Ahmed, P. Ramanathan, and K. K. Saluja. Necessary and sufficient conditions for thermal schedulability of periodic real-time tasks under fluid scheduling model. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(3):49, 2016.

[4] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *IEEE Real-Time Systems Symposium*, 1999.

[5] K. Bletsas, N. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. *Leibniz Transactions on Embedded Systems*, 5(1):02–1, 2018.

[6] Y. Chandarli, N. Fisher, and D. Masson. Response time analysis for thermal-aware real-time systems under fixed-priority scheduling. In *IEEE International Symposium on Real-Time Distributed Computing (ISORC)*, 2015.

[7] T. Chantem, R. P. Dick, and X. S. Hu. Temperature-aware scheduling and assignment for hard real-time applications on mpsocs. In *2008 Design, Automation and Test in Europe*, pages 288–293, March 2008.

[8] J. J. Chen, S. Wang, and L. Thiele. Proactive speed scheduling for real-time tasks under thermal constraints. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009.

[9] Pascal Tuning Guide. https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html, 2018.

[10] S. M. D'Souza and R. Rajkumar. Thermal implications of energy-saving schedulers. In *ECRTS*, 2017.

[11] G. A. Elliott and J. H. Anderson. Globally scheduled real-time multiprocessor systems with gpus. *Real-Time Systems*, 48(1):34–74, 2012.

[12] G. A. Elliott, B. C. Ward, and J. H. Anderson. Gpusync: A framework for real-time gpu management. In *IEEE Real-Time Systems Symposium(RTSS)*, 2014.

[13] M. Fan, V. Chaturvedi, S. Sha, and G. Quan. An analytical solution for multi-core energy calculation with consideration of leakage and temperature dependency. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 353–358, Sept 2013.

[14] Y. H. Gong, J. J. Yoo, and S. W. Chung. Thermal modeling and validation of a real-world mobile ap. *IEEE Design Test*, 35(1):55–62, Feb 2018.

[15] H. Huang, V. Chaturvedi, G. Quan, J. Fan, and M. Qiu. Throughput maximization for periodic real-time systems under the maximal temper-

ature constraint. *ACM Trans. Embed. Comput. Syst.*, 13(2s):70:1–70:22, Jan. 2014.

[16] A. Iranfar, M. Kamal, A. Afzali-Kusha, M. Pedram, and D. Atienza. Thespot: Thermal stress-aware power and temperature management for multiprocessor systems-on-chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.

[17] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. Rgem: A responsive gpgpu execution model for runtime engines. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 57–66. IEEE, 2011.

[18] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011.

[19] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt. Gdev: First-class gpu resource management in the operating system. In *USENIX Annual Technical Conference*, pages 401–412. Boston, MA;, 2012.

[20] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar. A server-based approach for predictable gpu access control. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017.

[21] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar. A server-based approach for predictable GPU access with improved analysis. *Journal of Systems Architecture*, 88:97–109, 2018.

[22] H. Kim, S. Wang, and R. Rajkumar. vmpcp: A synchronization framework for multi-core virtual machines. In *2014 IEEE Real-Time Systems Symposium*, pages 86–95, Dec 2014.

[23] P. Kumar and L. Thiele. Cool shapers: Shaping real-time tasks for improved thermal guarantees. In *Proc. of Design Automation Conference (DAC 2011)*, pages 468–473, San Diego, California, Jun 2011. ACM.

[24] C. J. Lasance. Thermally driven reliability issues in microelectronic systems: status-quo and challenges. *Microelectronics Reliability*, 43(12):1969–1974, 2003.

[25] J. Lee, Y.-W. Seo, W. Zhang, and D. Wettergreen. Kernel-based traffic sign tracking to improve highway workzone recognition for reliable autonomous driving. In *Intelligent Transportation Systems-(ITSC), 2013 16th International IEEE Conference on*, pages 1131–1136, 2013.

[26] Y. Liu, R. P. Dick, L. Shang, and H. Yang. Accurate temperature-dependent integrated circuit leakage power estimation is easy. In *2007 Design, Automation Test in Europe Conference Exhibition*, pages 1–6, April 2007.

[27] Mali OpenCl SDK. https://developer.arm.com/products/software/mali-sdks, 2016.

[28] ODROID-XU4. http://www.hardkernel.com/, 2016.

[29] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang. An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*, pages 353–364. IEEE, 2017.

[30] P. Patel, I. Baek, H. Kim, and R. R. Rajkumar. Analytical enhancements and practical insights for mpcp with self-suspensions. 2017.

[31] F. Paterna and T. S. Rosing. Modeling and mitigation of extra-soc thermal coupling effects and heat transfer variations in mobile devices. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 831–838, Nov 2015.

[32] A. Prakash, H. Amrouch, M. Shafique, T. Mitra, and J. Henkel. Improving mobile gaming performance through cooperative cpu-gpu thermal management. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 47:1–47:6, New York, NY, USA, 2016. ACM.

[33] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 116–123. IEEE, 1990.

[34] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 259–269. IEEE, 1988.

[35] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarchical fixed-priority scheduling. In *ECRTS*, 2002.

[36] O. Sahin and A. K. Coskun. Providing sustainable performance in thermally constrained mobile devices. In *2016 14th ACM/IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*, pages 1–6, Oct 2016.

[37] L. Schor, I. Bacivarov, H. Yang, and L. Thiele. Worst-case temperature guarantees for real-time applications on multi-core systems. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 87–96, April 2012.

[38] L. Sha, J. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1986.

[39] G. Singla, G. Kaur, A. K. Unver, and U. Y. Ogras. Predictive dynamic thermal and power management for heterogeneous mobile platforms. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 960–965, March 2015.

[40] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, Mar. 2004.

[41] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[42] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In *Dependable Systems and Networks, 2004 International Conference on*, pages 177–186. IEEE, 2004.

[43] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

[44] R. Viswanath, V. Wakharkar, A. Watwe, V. Lebonheur, et al. Thermal performance challenges from silicon to systems. 2000.

[45] S. Wang, Y. Ahn, and R. Bettati. Schedulability analysis in hard real-time systems under thermal constraints. *Real-Time Systems*, 46(2):160–188, 2010.

[46] S. Wang and R. Bettati. Delay analysis in temperature-constrained hard real-time systems with general task arrivals. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 323–334, Dec 2006.

[47] Y. Xiang, T. Chantem, R. P. Dick, X. S. Hu, and L. Shang. System-level reliability modeling for mpsocs. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 297–306. ACM, 2010.

[48] K. G. S. Youngmoon Lee, Hoonsung Chwa and S. Wang. Thermal-aware resource management for embedded real-time systems. In *Embedded Software (EMSOFT), 2018 International Conference on*. IEEE, 2018.

[49] H. Zhou, G. Tong, and C. Liu. Gpes: A preemptive execution system for gpgpu computing. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 87–97. IEEE, 2015.