Thermal-aware Adaptive Platform Management for Heterogeneous Embedded Systems

SRIJEETA MAITY, ANIRBAN GHOSE, and SOUMYAJIT DEY, Indian Institute of Technology Kharagpur, India
SWARNENDU BISWAS, Indian Institute of Technology Kanpur, India

Recent trends in real-time applications have raised the demand for high-throughput embedded platforms with integrated CPU-GPU based Systems-On-Chip (SoCs). The enhanced performance of such SoCs, however, comes at the cost of increased power consumption, resulting in significant heat dissipation and high on-chip temperatures. The prolonged occurrences of high on-chip temperature can cause accelerated in-circuit ageing, which severely degrades the long-term performance and reliability of the chip. Violation of thermal constraints leads to on-board dynamic thermal management kicking-in, which may result in timing unpredictability for real-time tasks due to transient performance degradation. Recent work in adaptive software design have explored this issue from a control theoretic stand-point, striving for smooth thermal envelopes by tuning the core frequency.

Existing techniques do not handle thermal violations for periodic real-time task sets in the presence of dynamic events like change of task periodicity, more so in the context of heterogeneous SoCs with integrated CPU-GPUs. This work presents an OpenCL runtime extension for thermal-aware scheduling of periodic, real-time tasks on heterogeneous multi-core platforms. Our framework mitigates dynamic thermal violations by adaptively tuning task mapping parameters, with the eventual control objective of satisfying *both* platform-level thermal constraints and task-level deadline constraints. We consider multiple platform-level control actions like task migration, frequency tuning and idle slot insertion as the task mapping parameters. To the best of our knowledge, this is the first work that considers such a variety of task mapping control actions in the context of heterogeneous embedded platforms. We evaluate the proposed framework on an Odroid-XU4 board using OpenCL benchmarks and demonstrate its effectiveness in reducing thermal violations.

CCS Concepts: • **Computer systems organization → Real-time systems**; **Parallel architectures**; • **Hardware → Thermal issues;**

Additional Key Words and Phrases: Heterogeneous computing, thermal violation, adaptive thermal management

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT), 2021. The authors acknowledge generous grant received from "IHUB NTIHAC Foundation - IIT Kanpur" for partially supporting this work.

Authors' addresses: S. Maity, A. Ghose, and S. Dey, Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, West Bengal, 721302, India; emails: srijeeta.maity@iitkgp.ac.in, {anirban.ghose, soumya}@cse.iitkgp.ac.in; S. Biswas, Department of Computer Science and Engineering, Indian Institute of Technology Kanpur, Uttar Pradesh, 208016, India; email: swarnendu@cse.iitk.ac.in.

1539-9087/2021/09-ART97 $15.00
https://doi.org/10.1145/3477028

ACM Reference format:
Srijeeta Maity, Anirban Ghose, Soumyajit Dey, and Swarnendu Biswas. 2021. Thermal-aware Adaptive Platform Management for Heterogeneous Embedded Systems. *ACM Trans. Embedd. Comput. Syst.* 20, 5s, Article 97 (September 2021), 28 pages.
https://doi.org/10.1145/3477028

1 INTRODUCTION

Modern embedded platforms like automotive systems, mobile platforms, and gaming consoles often need to execute a variety of workloads with different computational characteristics under stringent power and timing budgets. Due to this, Multiprocessor System-on-Chips (MPSoCs), commonly used for low-power but high-performance embedded systems, have become increasingly heterogeneous in nature. Such MPSoCs are comprised of multiple processing elements (PEs) like general purpose CPU cores, accelerators such as graphic processing units (GPUs), and special function cores like digital signal processors (DSPs) and neural processing units (NPUs). Co-existence of such cores with complex functionalities, coupled with the fact that shrinking technologies increase chip density, results in higher chip-level power density leading to high on-chip temperatures in modern MPSoCs. Repeated occurrences of high on-chip temperatures (i.e., a thermal violation) shorten the lifetime of such MPSoCs and accelerate in-circuit ageing [2]. Also, this may severely degrade the performance and reliability of the chip, sometimes even risking the safety of the system leading to catastrophic consequences for driving, medical and wearable applications, where such embedded heterogeneous MPSoCs are extensively used.

The problem: Dynamic Thermal Management (DTM) methods provide different power capping capabilities to the MPSoCs. When the temperature of any PE exceeds some predefined threshold, the power consumption of the PE is forcibly reduced by DTM techniques [49]. Common DTM methods include processor level techniques like clock or fetch gating [44] and software-directed Dynamic Frequency/Voltage Scaling (DVFS) [5, 21, 28]. Such techniques are supported at the hardware level by almost all modern processors having multiple power domains and the ability to operate at different frequency/voltage settings. DVFS support is an intrinsic feature of all popular operating system distributions for server, desktop as well as mobile systems. Henceforth, we refer to this as default DTM. However, for such default DTM techniques, the focus remains solely on reducing the chip temperature which can affect the application performance resulting in degraded Quality of Service (QoS). This is also true for other high-level DTM techniques like temperature-oriented task migration approaches proposed for many-core systems [13, 26]. For latency-sensitive applications such as real-time tasks, latency-unaware DTM can thus be fatal. Hence, there is a need for platform resource managers that allocate compute resources with suitably tuned configurations such that *both* application level latency and platform level thermal requirements are met.

There is an increasing demand for supporting dynamic task scheduling primitives in adaptive software systems. This essentially requires accommodating new tasks (e.g., task injection) or new instances of existing tasks with reduced periodicity (i.e., job injection) in the current schedule. As an example, the real-time workloads executing in an automotive platform can be a mix of time-driven periodic tasks and event-driven sporadic tasks. Automobiles that implement perception systems consist of inferencing pipelines for detecting traffic signs and pedestrians [16]. The periods of such tasks need to be adjusted to accommodate dynamic frame sampling rates so that the detection accuracy remains above a threshold irrespective of the lighting condition and the locality [45]. Alternatively, the frame resolutions may be adaptive leading to varying execution time of the inference pipeline when the period is kept constant [48]. Vehicular control loops can also have

scenario-based modification in sampling rates based on control performance and co-scheduling requirements [18]. Monitoring tasks can also get triggered by correlating multiple sensor values in an event driven manner for attack detection, mitigation, general diagnostics, and prognostics [17].

Dynamic modification of executing task sets, changes in periodicity and execution time demand adaptive resource management strategies for satisfying both thermal and real-time constraints. The problem of task-to-core allocation under such multi-dimensional constraints is complex. Firstly, the task-to-core mapping configuration space is overly large due to the heterogeneous nature of modern embedded multi-cores [46]. Also, the nonlinear relationship among the performance of concurrent tasks due to the interference created by shared resource accesses (e.g., shared L2 cache) makes performance modeling complex [47]. Trying to figure out resource mappings which ensure schedulability at peak system load can lead to thermal envelope violation causing reliability issues in the long-term [6]. Another solution is to calculate suitable resource mapping options for all possible changes in the workload scenario in an offline phase and applying it online. But such static resource mapping fails to scale for large scale dynamic systems with multiple applications with different occurrence rate, mapping choices and multiple constraints. Such static scheduling solutions are also overly pessimistic [36]. So, there is an inherent need for adaptive, lightweight scheduling mechanisms that can perform dynamic mapping by online resource analysis and re-configure scheduling decisions as and when required in a thermally aware manner for heterogeneous many-cores.

Our Approach: Designing an efficient thermal-aware, adaptive resource manager, that is cognizant of the application performance, as well as the thermal profile, is a challenging problem for heterogeneous multi-cores. Different dynamic resource mapping methodologies for CPU-GPU platforms have been explored that optimize performance mainly in terms of latency and power. These approaches employ heuristic based, greedy, ML and control theoretic techniques [34, 35, 39, 46]. However, most approaches ignore potential thermal violations in the system and the effects of thermal throttling. Prior work does not address the issue of real-time task scheduling in CPU-GPU integrated systems while handling thermal violations in case of dynamic workload modifications.

Our work provides a run-time platform manager which aims precisely at filling up these lacunae. We try to address the aforementioned problem by formulating lightweight heuristic-guided scheduling solutions which perform online thermal management of heterogeneous CPU-GPU platforms executing multiple real-time tasks. We consider a combination of runtime task partitioning, migration, idleness insertion, and control theoretic DVFS as scheduling moves. Existing work consider only one of these approaches for CPU-GPU systems [38, 42], or a subset of these approaches but for homogeneous multi-cores [30].

We have implemented an adaptive framework capable of supporting dynamic workload scenarios that keep on changing the scheduling status-quo of an existing real-time task set. The framework is generic and can be extended to any heterogeneous multi-core platform equipped with core level temperature sensors and OpenCL programming support. The efficacy of the proposed heuristics used by the framework is established on a real embedded CPU-GPU platform (Section 7.1) through extensive experimentation involving a variety of workload scenarios with different choices of periodic and event-driven job injections and different arrival rates of constituent tasks. We have observed that our approach exhibits positive results in terms of dynamically reducing peak temperature and deadline violations in comparison to other existing approaches.

2 BACKGROUND

In the following, we briefly discuss concepts that are relevant to this work.

Heterogeneous Computing: Modern embedded computing platforms have seen a paradigm shift in their architectures and have become increasingly heterogeneous. For example, recent mobile and automotive platforms consist of multiple processing elements, such as high-throughput power-consuming 'big' CPU cores, low-power low-throughput 'LITTLE' CPU cores, and GPU cores integrated on the same die with a shared memory unit. Instead of using fixed (i.e., static) roles for each kind of device (e.g., CPUs for sequential and management tasks and GPUs for parallel work), heterogeneous programming models, such as OpenCL (by Khronos [37]), oneAPI (by Intel), and HetCompute (by Qualcomm), allow for efficient execution of a single data parallel kernel using all the available devices.

OpenCL Programming Model: OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms [37]. It provides a standard interface for parallel computing using task and data level parallelism on different compute devices such as CPU, GPU, FPGA and DSP. An OpenCL program typically consists of two distinct entities: (i) the *host* code which is a single-threaded serial program that executes on the host device (a CPU) and orchestrates the entire process of data transfer and issuing directives for parallel execution, and (ii) *kernel* code which executes the actual data parallel computation on compute devices.

Each compute device comprises multiple compute units (e.g., shader cores in case of GPUs) and each compute unit consists of multiple processing elements (e.g., arithmetic pipelines). A processing element executes a kernel instance called work-item that operates on a single data point. A group of work-items form a work-group and these items execute concurrently on the processing elements of a single compute unit. The OpenCL memory model demands memory consistency among work-items within the same work-group but not across different work-groups. Different work-groups of the same kernel can be launched on different devices without worrying about maintaining memory consistency among compute devices. This allows scheduling of tasks on a heterogeneous platform to be done asynchronously at the granularity of work-groups.

OpenCL Task partitioning: Partitioning tasks in the context of OpenCL applications entail splitting the data space to be processed by the task/kernel into segments and processing the segments concurrently using the devices of the heterogeneous system. In contrast to the traditional approach of executing tasks on a single accelerator device in its entirety, partitioning allows for optimum resource utilization of the different processing elements in a heterogeneous platform leading to reduced execution times. Several partitioning based methodologies have been proposed over the years that advocate such collaborative execution of data parallel tasks on CPU/GPU platforms [14, 15, 20, 27, 29, 46]. Such approaches typically decide upon a partitioning ratio that would determine how the data space can be distributed for processing across the CPU and GPU devices concurrently such that overall execution time is minimized.

Task partitioning in this context occurs at the work-group granularity and is orchestrated by the host program. The host is configured to launch a subset of the total number of work-groups on the CPU device while the remaining work-groups are launched on the GPU device. As an illustrative example consider the OpenCL vector addition kernel depicted in Figure 1 executing on a CPU-GPU platform. The kernel takes as input two buffers `input1` and `input2`, performs element wise addition and returns the result in `output`. We assume the number of elements of the buffers to be $n = 256$. The total number of work-items launched would be thus 256 where each work-item i is designated with a single addition operation: `output[i]=input1[i]+input2[i]`. Assuming a work-group size of 4, the total number of work-groups required for executing the kernel is $256/4 = 64$. The example depicted in Figure 1 considers a partitioning ratio of $1 : 3$, i.e., $1/4 * 64 = 16$ out of 64 work-groups (work-groups with ids $0 - 15$) are mapped to the CPU device. The remaining 48 (work-groups with ids $16 - 63$) are mapped to the GPU device. We note

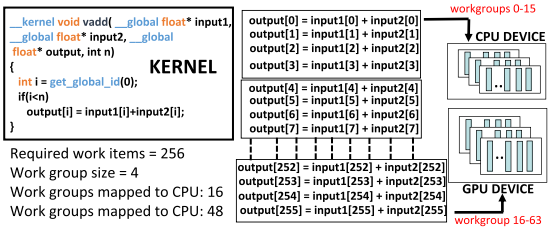

Fig. 1. Partitioning OpenCL Vector Addition.

that partitioning OpenCL tasks only require configuring the kernel launch commands from the host program and does not require modifying the source code of the kernels. The overhead of partitioning is thus in issuing extra kernel launch directives, which is negligible when compared to the overall execution time of the kernel. However, kernels with inter-thread data dependencies (e.g., reduction kernels) may require more careful choice of partitioning ratio and extra merging steps in the host program.

3 MOTIVATING EXAMPLE

We first conduct experiments that analyse the performance of different thermal management schemes which involve scheduling moves/actions like DVFS, task migration, and idle slot insertion. Our experiments constitute iteratively executing multiple instances of the data parallel SIMD style OpenCL task, GESUMMV (Scalar Vector and Matrix Multiplication), on the heterogeneous Odroid XU-4 platform. Figure 2(a) illustrates the working principle of different thermal management schemes that leverage the aforementioned moves/actions (both task and device level) for scheduling OpenCL tasks on CPU-GPU platforms.

Each small sub-figure in Figure 2(a) pictorially depicts the latency of the task subject to different schemes used in our experiments. The OpenCL task GESUMMV is represented as a collection of multiple threads, where the thread length is proportional to the task latency. The rectangular blocks represent the device to which the task is mapped (red for CPU and green for GPU). The impact of these different schemes on the chip temperature (peak and average) and the task latency is illustrated in Figure 2(b). Here, the x-axis shows the latency and the y-axis plots the temperature. The orange crosses in Figure 2(b) represent the peak temperature versus the latency plot. The blue circles represent the average temperature versus the latency plot for each case.

We first describe our observations on three DVFS-based DTM schemes (top row of Figure 2(a)).
Aggressive thermal throttling (Case A): In our platform, if the peak temperature of the CPU exceeds 90°C (trip point), the OS applies thermal throttling [46] that automatically underclocks the processor. The frequency reduces from 2GHz to 0.9GHz and stays there until the temperature drops to 82°C. The frequency is raised to 2GHz thereafter. This action is irrespective of task deadline characteristics. In the figure, the high/low frequency regions of thread execution are denoted by corresponding frequency modulation of the threads.
Constant low frequency (Case B): Given a (task, device) pair, this scheme sets the processor frequency to a constant low value (1.2GHz in our experiments) for the entire lifetime of the application so that default thermal throttling is not triggered. In the figure, thread execution is denoted by a constant frequency modulation with increased length of the threads due to increased latency.
Online frequency tuner (Case C): In this scheme, a simple control logic observes the core-level temperatures at regular intervals while executing tasks on the CPU. If the peak temperature reaches 88°C, it decreases the frequency in steps of 0.2 GHz in each sensor sampling interval. If the temperature goes below a threshold 84°C, the frequency is increased in steps of 0.1 GHz. For

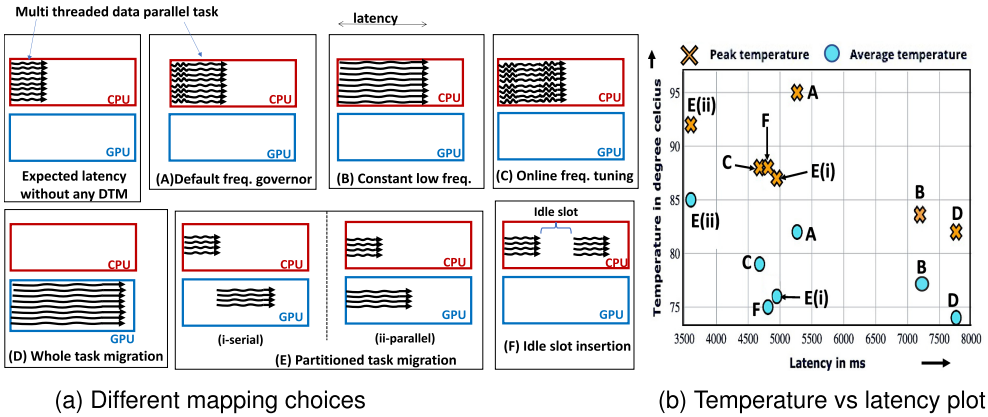

Fig. 2. Variation in task mapping and their effect on temperature and latency.

our example, the task executes on the CPU initially set at frequency 2GHz and gradually reduces and remains in the range 1.5–1.8GHz.

In Figure 2(b), we observe the peak CPU temperature in Case A reaches 95°C and then drops quickly leading to increased latency due to thermal throttling, whereas the average temperature is 82°C. For Case B and Case C, the peak temperature reaches 84°C and 88°C (below the trip point) respectively, whereas the average temperatures are 77 and 79°C. Even though the task is mapped at the highest frequency on the CPU for Case A, its observed latency is more than Case C (12%) where the task executes at a comparatively lower frequency. The reason is, once the temperature reaches the trip point, the frequency drops drastically due to thermal throttling in Case A.

Next, we summarise some of the schemes that exploit task level actions like device choice, task partitioning, and tuning of idle slots as depicted in the bottom row of Figure 2(a).

Whole task migration (Case D): This scheme explores task migration by launching all SIMD threads pertaining to the task computation on a cooler device and keeping the hotter device idle. Thus, the chip temperature can be reduced without relying on DVFS. In our example, the task is initially mapped to the CPU, and then mapped to the cooler GPU in the next iteration of the task.

Partitioned task migration (Case E): This scheme partitions a data parallel task into smaller *sub-tasks* by appropriately distributing SIMD threads of the data parallel task across multiple heterogeneous processors. In this example, the task is partitioned into two sub-tasks, one executing on the CPU and the other on the GPU, in two possible ways. In **Case E(i)**, the partitioned sub-tasks are executed serially but on different devices. Once the first sub-task finishes its execution on the CPU, the second sub-task starts execution on the GPU. In **Case E(ii)**, the two sub-tasks are executed concurrently on CPU and GPU devices. The frequency setting is set to maximum for both devices.

Idleness insertion (Case F): Instead of executing the task continuously on a device, the scheme partitions the task into sub-tasks and inserts an idle slot between consecutive sub-tasks. Idle slots refer to time intervals when the device is kept idle. In our example, the task is partitioned into two sub-tasks and both execute on the CPU serially with an idle slot of 1 second between them.

Note from Figure 2(b) that Case D and Case E(ii) exhibit two extreme cases. The peak and average temperatures in Case D are the least and latency is maximum compared to all other cases. Case E(ii) exhibits the minimum latency but maximum average temperature and second-highest peak temperature. It has both cores occupied during task execution, leading to lesser latency but high power dissipation due to thermal coupling of cores. In our target platform, ARM Cortex A15 cores

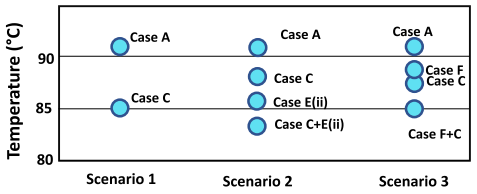

Fig. 3. Peak temperatures for different scenarios.

have greater compute capability than ARM Mali GPU and the maximum allowed frequency for the CPU and GPU are 2GHz and 0.6 GHz respectively. Case D is the only mapping option where we have the CPU completely left idle as the task instance has been migrated to the GPU. This explains the increased execution time. In Case B also, due to continuous use of low operating frequency, execution time is quite high. Case C, E(i) and F have similar performance in terms of both latency and reducing peak temperature, and also have better trade-offs between these two parameters. We observe Case E(i) and F have similar average temperature characteristics also. So, we consider these three schemes in our proposed framework as control actions for reducing thermal violations.

Given the promising results for Case C (frequency tuning), Case E (task migration) and Case F (idle slot insertion), we next investigate how synergistically executing combinations of these three schemes impact the reduction of peak temperatures. For this, we construct three relevant scheduling scenarios, each with different initial chip temperature, device utilization and target latency goals. For each scenario, our objective is to maintain the peak temperature below an upper threshold (85°C). If temperature is above 85°C (thermal violation), the schemes discussed should reduce the temperature below the threshold without violating the target latency goal. The peak temperatures achieved by different combinations of the schemes in each scenario is highlighted in Figure 3 where the x-axis plots the three scenarios and y-axis plots the peak temperature.

Scenario 1: We map the task to the CPU at the highest frequency setting while keeping the GPU idle. The target latency goal is relaxed. In such a scenario, frequency tuning (Case C) alone can reduce the peak temperature below the given threshold, as depicted in Figure 3.

Scenario 2: We map the task to the CPU at the highest frequency setting while keeping the GPU idle. The target latency goal is highly conservative. In such a scenario, sufficient frequency reduction (Case C) or task shifting (Case F) in isolation are not feasible for meeting deadline constraints. In contrast, partitioned task migration to GPU (Case E) and then applying frequency tuning (Case C) results in lower peak temperatures compared to individual schemes as depicted in Figure 3.

Scenario 3: We map the task to CPU at the highest frequency setting while keeping the GPU heavily engaged. The target latency goal is relaxed, but the initial chip temperature before the starting of the CPU bound task is high. Here, migration (Case E) is not an option since the GPU is busy. We observe from Figure 3, that rather than applying frequency tuning (Case C) alone, a combination of idle slot insertion (Case F) and frequency tuning (Case C) ensures better thermal management. This is because the idle slot inserted before executing the CPU task helps to reduce the initial high chip temperature such that it is manageable by frequency reduction.

The above scheduling experiments thus motivate scenario-based usage of DTM schemes for thermal-aware scheduling while maintaining target latency goals.

4 PROBLEM FORMULATION

In the following, we discuss the platform model, the task model, and the thermal model for our proposed framework, and then present our problem statement.

Platform Model: There exist multiple processing elements or devices with different compute capabilities in a heterogeneous integrated processing platform $\mathcal{P}$. Each device $\delta_i \in \mathcal{P}$ is potentially distinct from another device $\delta_j \in \mathcal{P}, j \neq i$, in terms of the architecture, computational power, and memory structure. The devices reside on the same die and share a common DRAM. Without loss of generality, we have considered one CPU and one GPU device on our platform, i.e., $\mathcal{P} = \{\delta_C, \delta_G\}$, where δ_C and δ_G denote the CPU and the GPU devices, respectively. The trip point at which OS-governed thermal throttling takes place for the device δ_j is denoted as T_{δ_j}. In our work, the trip point for both δ_C and δ_G are considered the same and denoted by T_{trip}.

Task Model: We assume the input task set $\mathcal{T} = \mathcal{T}^p \cup \mathcal{T}^s$ comprises both time-triggered periodic tasks ($\mathcal{T}^p$) and event-triggered sporadic tasks ($\mathcal{T}^s$). Once started, any sporadic task executes for a bounded number of instances with some periodicity. Each periodic task $\tau_i \in \mathcal{T}^p$ executes with some period $\in [p_i^l, p_i^h]$ depending on the dynamic performance requirement of the subsystems (e.g., detection pipelines and control modules pertaining to various vehicle domains). At any point of time, the currently executing job set $\mathcal{J}$ thus comprises multiple instances (jobs) of both periodic and sporadic tasks inside a hyper-period following their respective specification of period values. A new scheduling *epoch* starts due to possible modifications in $\mathcal{J}$. The currently executing job set $\mathcal{J}$ can get modified to some $\mathcal{J}'$ in a new scheduling epoch in the following ways.

(1) The period specification for some periodic task $\tau_i \in \mathcal{T}_p$ can change from period p_i to some p_i' as dictated by the performance requirements of the underlying subsystems. Note that both $p_i, p_i' \in [p_i^l, p_i^h]$.
(2) The set of sporadic tasks $\mathcal{T}_s' \subseteq \mathcal{T}_s$ to be run may change. Job instances corresponding to each new sporadic task $\tau_i \in \mathcal{T}_s'$ are then added to $\mathcal{J}'$ following their period specification, and sporadic tasks that expire are removed from the job set.

A real-time data parallel OpenCL task τ_i is characterized by the tuple $(W_i, w_i, E_i, [p_i^l, p_i^h], d_i)$. The quantities W_i and w_i represent the number of OpenCL work-groups and the number of work-items per work-group to be launched, respectively. The set E_i represents the set of worst-case execution times (WCET) of τ_i for all possible combinations of devices in $\mathcal{P}$ and their respective frequency settings. An element $e_{i,j,f} \in E_i$ represents the maximum time to complete τ_i on device δ_j at frequency f. In our framework, each element $e_{i,j,f}$ is measured offline over multiple runs while running suitable tasks in other devices so that maximum possible memory interference is created for τ_i. We maintain a data structure denoted by *WCET lookup table* for all the OpenCL tasks considered in the framework in an offline stage such that for any executing OpenCL task, the corresponding set of worst-case execution times (WCET) can be readily accessed during online scheduling. For any task, a choice of period $p \in [p_i^l, p_i^h]$ is associated with a related choice of deadline d ($d \leq p$).

We note that these periodic OpenCL tasks (new or existing) are non-preemptive in nature, i.e. once the task is dispatched to the command queue, the device choice cannot be changed by issuing directives from the host program. The problem of non-preemption can be handled by partitioning data parallel OpenCL tasks into suitable smaller sized *sub-tasks*. Each *sub-task* is a subset of work-groups used in the original task, operating on a subset of the input data space. Let $\tau_{i_p}^k$ denotes the k^{th} sub-task of the p^{th} job instance of the task τ_i. If the number of sub-tasks for a task τ_i is denoted by K_i where $1 \leq K_i \leq W_i$, then the value of K_i decides how frequently scheduling decisions for a task can be modified. The *local deadline* of each such sub-task $\tau_{i_p}^k$ is denoted as $d_{i_p}^k$. This is estimated from the relative deadline d_i of the whole task τ_i and the fraction ($frac_{i_p}^k$) of total work-groups in τ_i present in the sub-task ($d_{i_p}^k = frac_{i_p}^k \times d_i$). Let $e_{i,j,f}^k$ denote the maximum time

for executing sub-task $\tau_{i_p}^k$ on the device δ_j at frequency f. This is estimated from $e_{i,j,f}$ and the fraction of work-groups present in $\tau_{i_p}^k$.

Definition 4.1. The *task mapping* triplet $m_{i_p}^k = (\delta_j, f, t_s)$ denotes the sub-task $\tau_{i_p}^k$ is mapped to the device δ_j at frequency f where t_s is the relative start time of the sub-task, i.e. after the arrival of $\tau_{i_p}^k$, the execution starts after time t_s. By default, it is set to 0. The mapping is *valid* if $t_s + e_{i,j,f}^k \leq d_i$ where $e_{i,j,k}^k$ is the estimated worst-case execution time for finishing sub-task $\tau_{i_p}^k$.

Definition 4.2. The *schedule* $\mathcal{S}$ for the job set $\mathcal{J}$ in some hyper-period H on the platform $\mathcal{P}$ is the set of all task mappings for all sub-tasks of all jobs in $\mathcal{J}$. $\mathcal{S}$ is said to be *valid* if the mapping decision $m_{i_p}^k$ is valid for every sub-task in every job in $\mathcal{J}$.

The mapping triplet for a (sub)task can be altered by the following well-defined control actions.

(1) *Frequency tuning* of sub-task $\tau_{i_p}^k$ can change the core frequency from f to some f' resulting in a modified mapping, $m_{i_p}^k = (\delta_j, f, t_s) \rightarrow m'^k_{i_p} = (\delta_j, f', t_s)$.

(2) *Task migration* of sub-task $\tau_{i_p}^k$ can alter the assigned device δ_j to some cooler device δ_j' resulting in a modified mapping, $m_{i_p}^k = (\delta_j, f, t_s) \rightarrow m'^k_{i_p} = (\delta_j', f, t_s)$.

(3) *Idle slot insertion* for sub-task $\tau_{i_p}^k$ can alter the start time by inserting some delay t resulting in a modified mapping, $m_{i_p}^k = (\delta_j, f, t_s) \rightarrow m'^k_{i_p} = (\delta_j, f, t_s')$ such that $t_s' = t_s + t$.

We have used device utilization as a quantitative measure of the idleness present at a given time interval during a schedule. It gives a fair idea about the amount and type of computation handled by each device. For any given time interval I_a, we have classified the utilization ($U(I^a, j)$) of a device δ_j as either low (*Lo*) if $U(I^a, j) < \alpha_j$ or high (*Hi*) if $\alpha_j \leq U(I^a, j) \leq 1$, where $\alpha_j \in [0, 1]$ is an experimentally derived threshold parameter for device utilization calculated offline. Similarly, we have also defined a threshold parameter β_j by conducting extensive experimentation, which would dictate whether the frequency setting is high (*Hi*) or low (*Lo*) for the device δ_j.

Thermal Model: We have set two different thermal thresholds for each device: (i) *major* threshold ($T_{major} < T_{\delta_j}$) where T_{δ_j} is the trip point of the device δ_j and (ii) *minor* threshold ($T_{minor} < T_{major}$). For simplicity, we have considered the same value for T_{major} for both δ_C and δ_G and the same value for T_{minor} for both devices. The values of T_{major}, T_{minor} and T_{δ_j} considered for our platform is specified in Section 7.1. During run time, the temperature of both the devices are monitored at regular intervals of time (h) using the on-board temperature sensors to check whether core temperatures exceed the thermal thresholds. Whenever the temperature exceeds T_{minor} but not T_{major}, it is considered as a *minor* thermal violation. In case the temperature exceeds T_{major}, it is considered as a *major* thermal violation. A sub-task is marked *unstable* if *major* thermal violation is detected during its execution.

Our proposed framework targets to reduce the operating temperature for both kinds of violations. The primary reason for setting a range of lower values for thermal thresholds instead of considering the OS specified trip point is to ensure that immediate recovery actions can be taken before reaching the trip point. This would prevent the OS-governed thermal throttling routine from kicking in which would drastically alter core frequency leading to potential deadline violations.

In the steady state of system operations, we assume that the platform is thermally stable until thermal violations are detected due to some change(s) in the current job set $\mathcal{J}$. A platform $\mathcal{P}$ is said to be *thermally stable* with respect to a given job set $\mathcal{J}$ iff for a valid schedule $\mathcal{S}$ for a

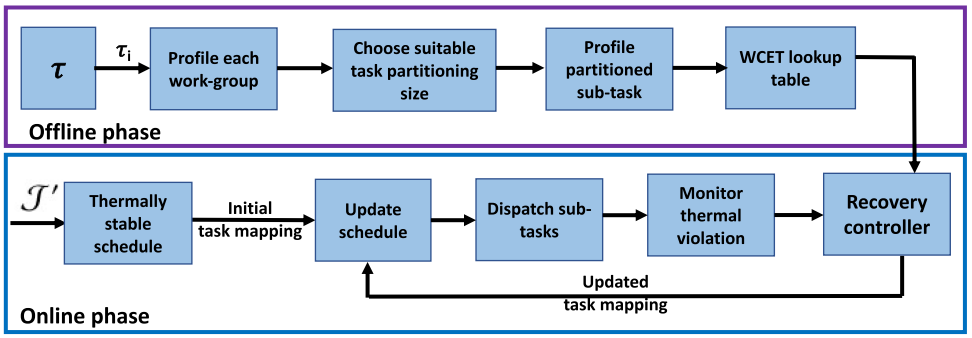

Fig. 4. Workflow overview.

hyper-period H, the peak temperature in all the devices in $\mathcal{P}$ remains below a given threshold value T_{major} throughout the schedule.

Problem Statement: Our objective is to solve the following design problem in this work: *Given a job set $\mathcal{J}$ running on a heterogeneous embedded platform $\mathcal{P}$ equipped with thermal sensors and following a feasible thermally stable schedule $\mathcal{S}$, in case of thermal violations caused due to change in $\mathcal{J}$ to $\mathcal{J}'$ at run time, our proposed framework decides how a new feasible thermally stable schedule $\mathcal{S}'$ can be reached adaptively.*

We propose the design of an intelligent scheduler that runs periodically and minimizes thermal violations adaptively using control theoretic approaches and heuristics. The thermal-aware control actions executed by the scheduler are task migration, frequency tuning and idle slot insertion, performed iteratively in a feedback loop.

5 METHODOLOGY OVERVIEW

Figure 4 depicts the workflow involved in both offline and online phases of our proposed framework. In the offline phase, our objective is to populate the WCET lookup table for the set of tasks in $\mathcal{T}$. This is done by profiling every k^{th} sub-task of $\tau_i \in \mathcal{T}$ for every possible combination of device j and frequency setting f to yield $e^k_{i,j,f}$. To determine what should be a suitable task-size for each sub-task and consequently the number of sub-tasks for any given task, a single work-group of τ_i is initially profiled in all the devices at both the highest and lowest frequency settings. The average execution time (ex_{avg}) for a single work-group is computed after multiple profiling runs. Based on ex_{avg}, τ_i is suitably partitioned into K_i sub-tasks. We set $K_i = W_i/((z \cdot h)/ex_{avg})$ where W_i is the number of OpenCL work-groups in τ_i, h is the sampling period of thermal sensors for the platform and $z \in \mathbb{Z}^+$. Note that for $z = 1$, K_n will be such that each sub-task will take (approximately) one sensor sampling period to execute. For larger z (i.e., smaller K_i), sub-tasks will be lesser but each sub-task will take around $z \cdot h$ amount of time to execute, empirically speaking. Since individual sub-tasks take longer time to execute, control actions in the recovery mode cannot be applied fast enough for avoiding thermal violation. For smaller z (i.e. larger K_i), the system will have higher task dispatch overhead. After partitioning, sub-tasks are profiled on each device at all possible frequencies. We follow the co-degradation-based approach [50] that ensures maximum memory interference while profiling the k^{th} sub-task of τ_i on device j at frequency f to obtain $e^k_{i,j,f}$. The profiling information thus derived is used in populating the WCET lookup table discussed earlier and are leveraged in subsequent hyper-periods for refining the task mapping of the task set when required.

In the online phase, let us consider that the currently executing job set $\mathcal{J}$ gets modified to $\mathcal{J}'$ in a new scheduling epoch. Let $\mathcal{J}_{rel}$ and $\mathcal{J}_{new}$ denote the job instances released and added in the system so that $\mathcal{J}' = \mathcal{J} \setminus \mathcal{J}_{rel} \cup \mathcal{J}_{new}$. If $\mathcal{J}_{rel}$ is nonempty, device utilization is reduced, thus allowing jobs in $\mathcal{J}_{new}$ more resource bandwidth to exploit. As depicted in Figure 4, an *initial mapping* is first constructed to update the schedule of the job set $\mathcal{J}'$.

For each job instance job_i in $\mathcal{J}_{new}$, the initial mapping is chosen greedily using the Worst-Fit Decreasing (WFD) algorithm [7]. This is done by mapping job_i in the hyper-period H to the device (δ_l) with the lowest device utilization in the time interval between the arrival and deadline of the job_i. The frequency of the device under consideration is then greedily chosen as the maximum value (f_M) to minimize the latency requirement so that the deadline of the job is satisfied. If the deadline constraint is not satisfied at the maximum frequency, then it cannot be scheduled at any lower configuration.

The relative start time (t_s) of the job_i is set to 0 by default, i.e., job_i can be dispatched right after its arrival. Here, start times of consecutive sub-tasks of the same task mapped to the same device and frequency setting are set with the assumption that the variation in latency of such sub-tasks is negligible. For all sub-tasks of job_i , the initial task mapping needs to be set to the greedily chosen device and frequency setting.

The initial task mapping ($m^k_{job_i}$) for each sub-task (job^k_i) is denoted by the triplet (δ_l, f_M, ($t_s + k \cdot e^k_{n,l,f_M}$)) where k is the index of the sub-task and e^k_{n,l,f_M} is the estimated worst-case execution time to finish k^{th} sub-task of job_i on δ_l at frequency f_M. If $\mathcal{M}$ denotes the set of the initial mappings of all sub-tasks of all job instances in $\mathcal{J}_{new}$ in a hyper-period H, the updated schedule $\mathcal{S}'$ over each hyper-period is given as $\mathcal{S}' = \mathcal{S} \setminus \mathcal{S}_{rel} \cup \mathcal{M}$ where $\mathcal{S} \setminus \mathcal{S}_{rel}$ is the schedule at the end of last epoch without the mapping of each job in $\mathcal{J}_{rel}$. The framework will next *dispatch* sub-tasks following $\mathcal{S}'$ and monitor the thermal profile at the end of executing each sub-task. In case of thermal violations, our proposed framework triggers a *recovery controller* which adaptively tries to mitigate the thermal violations. The recovery controller chooses appropriate thermal recovery actions (discussed in the next section) that modify the task mapping to reduce peak temperature while satisfying the deadline constraints of the tasks with the help of the WCET lookup table created offline. The new mappings are updated in the schedule and the process continues for the whole epoch or until the platform is thermally stable again.

6 THERMAL VIOLATION AND RECOVERY MECHANISM

The work flow of the thermal recovery mechanism is depicted in Figure 5. Let us consider that in a given hyper-period a sub-task $\tau^k_{i_p}$ is dispatched with task mapping $m^k_{i_p} = (\delta_j, f, t_s)$ in platform $\mathcal{P}$ following schedule $\mathcal{S}$. The thermal profile during the execution of the sub-task $\tau^k_{i_p}$ is analysed to identify the peak temperature T_j in the device δ_j. If there is no thermal violation, i.e. $T_j \leq T_{minor}$, the framework does not take any action and continues with the existing task mapping decisions in $\mathcal{S}$. If there is any kind of thermal violation, the temperature is reduced by the recovery controller that work in two distinct modes - local and global recovery mode.

If a major thermal violation is observed (i.e., $T_j > T_{major}$), the *global recovery* mode is activated and the sub-task is marked *unstable*. A closed feedback loop is used to adaptively change the task mapping of the unstable sub-task $\tau^k_{i_p}$ by taking suitable control actions discussed earlier in Section 4. In this context, feedback refers to the difference in expected and the actual performance with respect to peak temperature and latency of $\tau^k_{i_p}$. Based on the observed feedback, the current thermal behaviour and the resource availability of the platform $\mathcal{P}$, a control action is selected

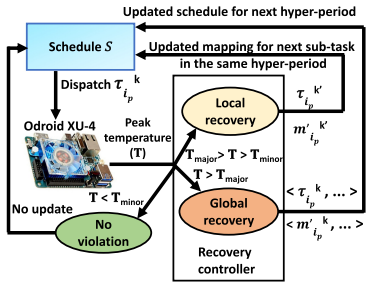

Fig. 5. Thermal Recovery Mechanism.

such that the peak temperature drops below T_{major} while ensuring there is no deadline violation. The resultant task mapping $m'^{k}_{i_p}$ is updated in schedule $\mathcal{S}$ to yield a new schedule $\mathcal{S}'$ which is followed in the next hyper-period. We note that while such task mapping modification decisions are adaptive and accurate in mitigating thermal violation, the changes are reflected at an interval of one hyper-period. In the case of a minor thermal violation (i.e, $T_{major} \geq T_j > T_{minor}$), the recovery controller activates the *local recovery* mode in the device δ_j. It changes the task mapping of the next sub-task ($\tau^{k'}_{i_p}$) scheduled in δ_j after $\tau^{k}_{i_p}$. In contrast to global recovery mode, changes made in local recovery mode are reflected immediately in the same hyper-period, i.e., it does not get updated in the existing schedule $\mathcal{S}$. We note that such task mapping modification decisions may not be accurate in mitigating thermal violation. However, this mode is lightweight when compared to global recovery and initiates an instantaneous cooling effect on the relevant device. The recovery controller thus chooses local recovery mode when the peak temperature is high but not at the risk of overshooting the trip point that will trigger OS induced thermal throttling leading to increase in deadline violations.

6.1 Global Recovery Mode

The following three steps are performed in global recovery mode to reduce the peak temperature in the upcoming hyper-period.

Interval detection: At the end of a hyper-period H, the operating regions of the recovery controller in global recovery mode is identified by constructing a set of intervals $\mathcal{I}_H$ as follows. Initially, for each unstable sub-task identified, an *interval* $I_q = (s_q, e_q)$ is created where s_q denotes the starting time and e_q denotes the local deadline of the sub-task. A pair of intervals $I_q = (s_q, e_q)$ and $I_r = (s_r, e_r)$ can be overlapped if $s_r \leq e_q$. Since there can exist multiple such intervals for different sub-tasks that can overlap, we merge such intervals so that the number of operating regions is reduced. The set $\mathcal{I}_H$ contains the set of such merged intervals only.

Resource Analysis: In the resource analysis phase, the resource availability, device utilization, frequency settings and peak temperatures are analysed for each identified interval in $\mathcal{I}_H$. Device utilization gives a fair idea about the amount and type of computation handled by the device, and a quantitative measure of idleness present in the given interval. For each interval $I^a = (s^a, e^a)$, we denote device utilization for each device δ_j as $U(I^a, j)$. This represents the fraction of time δ_j was busy executing sub-tasks in the duration of I^a. For any given interval I^a, the utilization and frequency setting of a device δ_j is classified as either low (*Lo*) depending on experimentally derived threshold parameters α and β_j calculated offline. This binary classification with respect to frequency and utilization is done to simplify run-time decisions. Each device in the platform can

Table 1. Recovery Action Selection

<table>
<tr><th rowspan="2">Thermal violation in</th><th rowspan="2">(Utilization, Frequency) in δ_G</th><th colspan="4">(Utilization, Frequency) in δ_C</th></tr>
<tr><th>(Lo, Lo)</th><th>(Lo, Hi)</th><th>(Hi, Lo)</th><th>(Hi, Hi)</th></tr>
<tr><td rowspan="4">δ_C</td><td>(Lo, Lo)</td><td rowspan="4">No action</td><td rowspan="4">Frequency tuning in δ_C</td><td colspan="2" rowspan="2">Migrate task to δ_G</td></tr>
<tr><td>(Lo, Hi)</td></tr>
<tr><td>(Hi, Lo)</td><td>Insert idle slot in δ_C</td><td>Frequency tuning in δ_C</td></tr>
<tr><td>(Hi, Hi)</td></tr>
<tr><td rowspan="4">δ_G</td><td>(Lo, Lo)</td><td colspan="4">No action</td></tr>
<tr><td>(Lo, Hi)</td><td colspan="4">Frequency tuning in δ_G</td></tr>
<tr><td>(Hi, Lo)</td><td colspan="2" rowspan="2">Migrate task to δ_C</td><td colspan="2">Insert idle slot in δ_C</td></tr>
<tr><td>(Hi, Hi)</td><td colspan="2">Frequency tuning in δ_G</td></tr>
<tr><td rowspan="4">Both</td><td>(Lo, Lo)</td><td>No action</td><td>Frequency tuning in δ_C</td><td colspan="2">Migrate task to δ_G</td></tr>
<tr><td>(Lo, Hi)</td><td>Frequency tuning in δ_G</td><td>Frequency tuning in both</td><td>Frequency tuning in δ_G</td><td>Frequency tuning in both</td></tr>
<tr><td>(Hi, Lo)</td><td colspan="2" rowspan="2">Migrate task to δ_C</td><td>Insert idle slot in both</td><td>Insert idle slot in δ_G,
Frequency tuning in δ_C</td></tr>
<tr><td>(Hi, Hi)</td><td>Insert idle slot in δ_C
Frequency tuning in δ_G</td><td>Insert idle slot in both</td></tr>
</table>

have a total of 2×2 possible configurations of device utilization and frequency settings. For our case, since there are two devices, we require examining $2^{2\times2} = 16$ different scenarios.

Control action selection: We design a controller routine that takes control theoretic scheduling decisions to choose appropriate *recovery actions* to be applied for the *next* hyper-period. These are chosen based on the observed values of device utilization and frequency settings identified by the resource analysis phase in each of the intervals in $\mathcal{I}_H$. Table 1 summarises the recovery actions taken by the controller for each of the sixteen different scenarios discussed above. The table layout is designed such that each control action will be chosen depending on the device where the major thermal violation is detected. There can be three possible thermal violation options, occurring in δ_C or δ_G or both. They are given in the first column of the table. For each of the three possible Utilization violation scenarios, the GPU configuration can be selected from sub-rows in Column 2, four possibilities of (Utilization, Frequency) for each violation scenario. The corresponding CPU violation scenario (again four possibilities) can be selected from the rest of columns. Thus, for any of the three thermal violation conditions (in Column 1), each intermediate cell in the table represents a recovery action taken for a pair of possible configurations pertaining to the devices δ_G and δ_C.

In all scenarios, the final objective is to reduce the peak temperature and thus mitigate thermal violations while respecting deadline constraints. We next explain the intuitions behind the choice of each recovery action taken. If both device utilization and frequency settings in interval I are low for a device where a major thermal violation has occurred, we do not change the current task mapping, since its effect on the peak temperature would be negligible.

If one device has high utilization whereas another device has low utilization i.e. the load distribution is not uniform, then irrespective of the frequency setting the controller opts for task migration and suitably changes the task mapping to balance the device utilization across devices. Task migration is preferred over frequency tuning in this case, since there is an opportunity of sharing the load on the device in the interval where the utilization is low. Decreasing the frequency of the device in this case might result in a potential deadline violation.

If the device utilization is high in all available devices, i.e., we cannot perform task migration, we either perform frequency tuning or idle slot insertion. If the frequency setting is high for the device where thermal violation occurred, the controller applies frequency tuning. The choice of frequency tuning over idle slot insertion here may be attributed to the fact that reducing frequency leads to more temperature reduction compared to task shifting for similar latency performance. If the frequency setting is already low, an *idle slot* is inserted, allowing the heated device some time for cooling down.

ALGORITHM 1: Frequency Tuning Mode

1 **Input:** $\tau_{i_p}^k, m_{i_p}^k = (\delta_j, f, t_s), ex\_time_{cur}$
2 **Output:** $m'^k_{i_p}$
3 $speed_{goal} \leftarrow size(\tau_{i_p}^k)/(local\_deadline(\tau_{i_p}^k) - t_s)$
4 $speed_{cur} \leftarrow (size(\tau_{i_p}^k)/ex\_time_{cur})$
5 error $\leftarrow speed_{goal} - speed_{cur}$
6 $speedup_{cur} \leftarrow speed_{cur}/speed_{base}$
7 $speedup_{nxt} \leftarrow speedup_{cur} + ((1-\rho)/speed_{base}) \times error$
8 $f_2 \leftarrow get\_frequency(speedup_{nxt}, \tau_i, \delta_j)$
9 **if** $f_2 < f_1$ **then**
10 $\quad m'^k_{i_p} = (\delta_j, f', t_s)$

Next, we describe how these three recovery actions have been designed.

A. Frequency Tuning: For each unstable sub-task $\tau_{i_p}^k$ in global recovery mode where frequency tuning is chosen as the recovery action, the frequency tuner runs once at the end of each hyper-period. It chooses a suitable alternate frequency for the demarcated sub-task to be used in the next hyper-period. It again iterates in the immediate next hyper-period and continues until $\tau_{i_p}^k$ is marked stable. The algorithm used for frequency tuning uses the well known pole-placement based self-tuning control techniques [23, 35] which dynamically model *speedup* of a task as a function of task latency at different clock frequency values. This is depicted in Algorithm 1 which takes as input an unstable sub-task $\tau_{i_p}^k$, its mapping $m_{i_p}^k = (\delta_j, f_1, t_s)$ and the actual execution time (ex_time_{cur}) recorded for $\tau_{i_p}^k$ in the current hyper-period. The output is the modified task mapping $m'^k_{i_p}$ which would be used in the next hyper-period. We define speed of a task τ_i as the ratio between the total number of work-items ($size(\tau_i) = W_i \cdot w_i$) for the task and the time taken to execute that task, where W_i is the total number of work-groups and w_i is the number of work-items per work-group for the task τ_i as defined earlier. Since, we want the sub-task $\tau_{i_p}^k$ to finish within its local deadline, the required speed i.e. $speed_{goal}$ is calculated by considering the number of work-items launched for the sub-task ($size(\tau_{i_p}^k)$) and the time left to finish the sub-task since it arrived within the required local deadline (line# 3 of Algorithm 1). The current speedup of $\tau_{i_p}^k$ is calculated by considering the actual time it took for execution, i.e. ex_time_{cur} (line# 4). The difference in required and actual speeds indicate the term *error* that needs to be adjusted (line# 5). Since speedup of a task is an abstraction of the relative changes in the performance with respect to frequency variation, we define it as the ratio between the speed of the task at frequency f and the speed of the task when it is executing at the minimum frequency (base speed). Based on this, the current speedup is calculated considering $speed_{cur}$ and $speed_{base}$ (line# 6). Our objective is to calculate $speedup_{nxt}$ which is the required speedup for executing the sub-task in the next hyper-period. This is obtained by using the control theoretic speedup equation $speedup_{nxt} \leftarrow speedup_{cur} + ((1-\rho)/speed_{base}) \times error$ where ρ is the pole of the controller. Given that we have access to the WCET lookup table, the required frequency f' for attaining $speedup_{nxt}$ is then obtained. If we observe that f' is less than f, the task mapping is modified to, $m'^k_{i_p} = (\delta_j, f', t_s)$ which would be used for executing the sub-task in the next hyper-period. This process of calculating speedup and obtaining the required processor frequency setting is repeated iteratively for consecutive hyper-periods in order to minimize the difference in expected and actual performance, i.e. *error*.

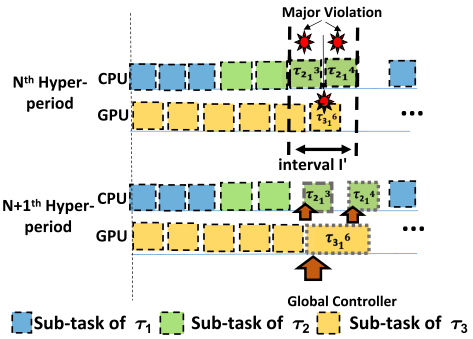

Fig. 6. Example of Global Controller actions.

B. Task Migration: The algorithm used for task migration ascertains whether it is possible to reduce the device utilization of the hotter device facing thermal violation so that the peak temperature could be reduced. For each unstable sub-task $\tau_{i_p}^k$ in global recovery mode where task migration is chosen as the recovery action, the task migration algorithm runs once at the end of each hyper-period to modify the device choice of $\tau_{i_p}^k$. Let the current task mapping of $\tau_{i_p}^k$ be $m_{i,p}^k = (\delta_{hot}, f, t_s)$. Let δ_{cool} be the device with the lowest peak temperature during the time between start time (t_s) and local deadline ($d_{i_p}^k$) of $\tau_{i_p}^k$. The algorithm allows migrating an unstable sub-task $\tau_{i_p}^k$ from device δ_{hot} to δ_{cool} if δ_{cool} can accommodate it. We greedily choose frequency setting for δ_{cool} as the maximum frequency (f_M) to increase the possibility of task migration. Let $avail$ be the available time slot between the local deadline $d_{i_p}^k$ and start time t_s when the device δ_{cool} is idle. The estimated WCET of $\tau_{i_p}^k$ on δ_{cool} at frequency f_M i.e. $e_{i,cool,f_M}^k$ is obtained from the WCET lookup table. If $e_{i,cool,f_M}^k \leq avail$, the task mapping $m_{i_p}^k$ is changed to $(\delta_{cool}, f_M, t_s)$.

C. Idle slot insertion: The algorithm used for idle slot insertion introduces idle slots between consecutive tasks in a device to allow the device to cool down when the device is idle. For each unstable sub-task $\tau_{i_p}^k$ in global recovery mode where idle slot insertion is chosen as the recovery action, the idle slot insertion algorithm runs once at the end of each hyper-period to modify the start time of $\tau_{i_p}^k$. Let the task mapping of $\tau_{i_p}^k$ be $m_{i_p}^k = (\delta_j, f, t_s)$ and the algorithm for idle slot insertion modifies t_s for the next hyper-period. The finish time (t_e) of $\tau_{i_p}^k$ is calculated from the WCET lookup table as $t_s + e_{i,j,f}^k$ where $e_{i,j,f}^k$ is the estimated WCET for the sub-task executed with task mapping $m_{i_p}^k$. The time the device δ_j is idle after $\tau_{i,p}^k$ finishes is denoted by $slack$. Here, $slack = d_{i,p}^k - t_e$ where $d_{i,p}^k$ is the local deadline of $\tau_{i_p}^k$. The sub-task is shifted by offsetting its starting time by this quantity $slack$, resulting in a new task mapping $m'^k_{i,p}$=$(\delta_j, f, t_s + slack)$. Such a choice of modified start time works as our WCET estimate $e_{i,j,f}^k$ bounds the maximum possible time taken by $\tau_{i,p}^k$.

As an example, let us consider the partial schedule for three periodic tasks illustrated in Figure 6 for two consecutive hyper-periods in the scheduling epoch. The lengths of each coloured rectangular box represents the execution time for each sub-task of some job of any task. The temperature monitor samples the platform temperature at a regular interval of h. At the end of each sub-task execution, the peak temperature over the execution period is compared with T_{major} and T_{minor}. We observe that peak temperature during the execution of the three sub-tasks

ALGORITHM 2: Local recovery

```
1 Input:τ^{k'}_{i_p}, m^{k'}_{i_p} = (δ_j, f, t_s) Output: m'^{k'}_{i_p}
2 slot ← d^{k'}_{i_p} − t_s                                   // d^{k'}_{i_p} is local deadline
3 P' ← sort_dev(P, slot)
4 for δ_a in P' do
5     if util_dev(δ_a, slot) = Lo then
6         for f' in F(δ_a) do
7             e^{k'}_{a,f',t_s} ← WCET(τ^{k'}_{i_p}, δ_a, f')
8             if t_s + e^{k'}_{a,f',t_s} < d^{k'}_{i_p} then
9                 m'^{k'}_{i_p} ← (δ_a, f', t_s) return m'^{k'}_{i_p}
```

$\tau^3_{2_1}$, $\tau^4_{2_1}$ and $\tau^6_{3_1}$ exceeds T_{major}. These three sub-tasks are marked unstable and the corresponding working interval I' (depicted in Figure 6) is determined from their start and finishing times. The resource analysis phase for the identified interval I' marks the device utilization in both CPU and GPU as *Hi*. The frequency setting is *Lo* for CPU and *Hi* for GPU. Based on these values of device utilization and frequency setting, one can observe from the rule set in Table 1 that the recovery actions are idle slot insertion in CPU and frequency tuning in GPU. Hence, for unstable tasks $\tau^3_{2_1}$ and $\tau^4_{2_1}$ executing on the CPU, idle slots are inserted. For $\tau^6_{3_1}$ executing on the GPU, frequency tuning is applied. The task mappings are changed and applied in the next hyper-period (depicted by orange arrows in Figure 6). The temperature is successfully reduced below T_{major}.

The global recovery mode runs for multiple hyper-periods until thermal stability is attained. In the worst-case, it may happen that the recovery controller in global recovery mode continues re-configuring the task mappings indefinitely without reaching any thermal stability, leading to an infinite many iterations of global controller. To manage such scenarios of non-convergence, the recovery controller has a permissible upper limit of applying recovery actions on an unstable sub-task in terms of the number of hyper-periods. In case this is exceeded without any observable positive effect on the peak temperature, the global recovery mode is exited and new jobs which are part of $\mathcal{J}_{new}$ are not admitted.

6.2 Local Recovery Mode

The local recovery mode alters the task mapping greedily in order to mitigate minor thermal violations. Let a minor violation be detected in δ_j during the execution of sub-task $\tau^k_{i_p}$. The immediately next sub-task after $\tau^k_{i_p}$ executing on δ_j is identified as $\tau^{k'}_{i_p}$. The local recovery algorithm depicted in Algorithm 2 takes as input this sub-task, its existing mapping $m^{k'}_{i_p}$=(δ_j, f, t_s) and outputs its updated task mapping $m'^{k'}_{i_p}$. We consider the time interval *slot* between the local deadline and the starting time of the sub-task of $\tau^{k'}_{i_p}$ as the working interval for the sub-task in the local recovery mode (line 2). The $sort\_dev(\mathcal{P}, slot)$ function sorts devices in $\mathcal{P}$ in ascending order of peak temperature recorded during the interval *slot* and returns the sorted list $\mathcal{P}'$ (line 3). Let $\mathcal{F}(\delta_a)$ denote the list of available frequency settings for each device δ_a sorted in ascending order of frequency value. For each device $\delta_a \in \mathcal{P}'$ starting with the coolest device, the expected device utilization in the interval *slot* is computed by $util\_dev(\delta_a, slot)$ (line 5). If it is *Lo*, each frequency setting $f' \in \mathcal{F}(\delta_a)$ starting from the lowest frequency setting is checked to see if the finishing time $(t_s + e^{k'}_{i,a,f'})$ after modifying its mapping decision is within the local deadline of the sub-task (line 8). Note, $e^{k'}_{i,a,f'}$

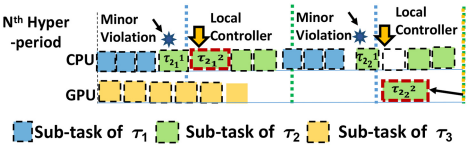

Fig. 7. Local Controller actions.

represents the WCET time for the task obtained from the WCET lookup table(line 7). The modified mapping decision $m'^{k'}_{i_p} = (\delta_a, f', t_s)$ is finally returned (line 9) and the local recovery mode is exited. We note that unlike the global recovery mode, the modified task mapping $m'^{k'}_{i_p}$ is valid for the current hyper-period and is not propagated to the next hyper-period. If no relevant cooler device is found, the mapping remains unchanged.

As an example, let us consider the partial schedule for some N^{th} hyper-period of three periodic tasks in Figure 7. Similar to Figure 6, the lengths of each coloured rectangular box represent the execution time for each sub-task of every job. For every job of a task marked with a specific colour, we have dotted vertical lines of the same colour representing their deadlines. The temperature monitor samples the platform temperature at a regular interval of T_s. At the end of each sub-task execution, the peak temperature over the execution period is compared with T_{major} and T_{minor}. We observe that *minor* thermal violations are detected during the execution of sub-tasks $\tau^1_{2_1}$ and $\tau^1_{2_2}$ on the device *CPU* in the N^{th} hyper-period of the current scheduling epoch. For the sub-task $\tau^1_{2_1}$, the task mapping of the immediate next sub-task $\tau^2_{2_1}$ is modified in local recovery mode. In this case, even though the GPU device is relatively cooler, it has high device utilization. Hence, there is not enough bandwidth available to facilitate migration of the sub-task. So, the local controller decides to use frequency tuning on the CPU device for sub-task $\tau^2_{2_1}$ in this context. For the sub-task $\tau^1_{2_2}$, the device utilization is less in cooler GPU device. Hence, the immediate next sub-task $\tau^2_{2_2}$ is migrated from CPU to GPU device.

7 EVALUATION

This section evaluates the effectiveness of our proposed framework in reducing the occurrence and intensity of thermal violations in heterogeneous embedded CPU-GPU platforms.

7.1 Test-bed Description

The target heterogeneous platform: We have performed our evaluations on an Odroid XU-4 embedded platform.[1] The platform uses the Exynos SoC which comprises the following three types of compute units: (i) a quad-core ARM Cortex-A7 (LITTLE CPU), (ii) a quad-core ARM Cortex-A15 (big CPU), and (iii) an ARM Mali-T628 GPU with 6 Streaming Multiprocessors (SMs).

The Odroid XU-4 platform supports a set of discrete operating clock frequencies for each compute element (18 for big CPU, 14 for LITTLE CPU, and 7 for the GPU). There are five temperature sensors in the platform: one for each of the 4 cores in the big CPU and one sensor for the GPU.

Runtime Framework: Our proposed framework has approximately 7000 LOC and leverages the asynchronous event driven programming model supported in OpenCL to (i) dispatch tasks to the big CPU and the GPU based on the respective arrival time of the tasks, and (ii) ascertain device availability through asynchronous callback functions which are instantiated when a device becomes free. For an OpenCL application executing on the CPU, the role of the CPU cores switch between host and compute devices. The CPU can act as a host or as an OpenCL compute device as

[1] https://magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual.pdf.

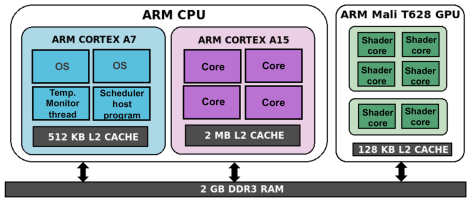

Fig. 8. Our runtime framework on Odroid XU-4.

Table 2. Candidate OpenCL Kernels

Task name	Description
2DCONV (τ_1)	2D Convolution
2MM (τ_2)	2 Matrix Multiplications
3MM (τ_3)	3 Matrix Multiplications
ATAX (τ_4)	Matrix Transpose and Vector Multiplication
GESUMMV (τ_5)	Scalar Vector and Matrix Multiplication
GEMM (τ_6)	General Matrix-multiply

required. This introduces certain limitations in the OpenCL runtime system. For example, in case of executing an OpenCL kernel on CPU, the host program waits for the CPU to be released from the kernel execution and cannot dispatch another kernel on the GPU even if the GPU is available. The GPU has to wait till the CPU cores have completed the current kernel execution. This unwanted overhead cannot be tolerated in real-time scheduling. This is handled by limiting the OpenCL compute device for CPUs to the A-15 cores of the ARM big CPU and using the LITTLE CPUs for all other services such as scheduling and temperature monitoring. We bind the OS (Ubuntu 18.04 LTS) to the first two cores of the quad-core ARM LITTLE CPU by modifying the OS boot file settings. We map our proposed scheduler (a complex host program) exclusively on the fourth core of the LITTLE CPU by setting the thread affinity during execution. We implement an additional thread that continuously monitors the chip temperatures. It executes on the third core of the LITTLE CPU and discretely samples the temperature readings from the sensors with a sampling period of 250 ms. In the rest of the paper, we use the terms CPU device, A15 cluster, or big CPU interchangeably.

Initial framework setup: The thermal threshold values for major and minor thermal violation are set to 87°C and 84°C respectively. For the platform under question, the trip point for CPU and GPU is 90°C. The value of α and β used for binary classification of device utilization and frequency setting in the framework for CPU are set to 0.7 and 1.5 GHz respectively and that of GPU are set to 0.55 and 0.5 GHz respectively. This is obtained by extensively running different task sets and considering only those cases where the peak temperature of the device δ_j exceeds T_{minor}. The quantity α_j represents the average device utilization observed for those cases. For each of the experiments, the drop in peak temperature is recorded whenever we decrease the frequency setting of the device from a particular value to its immediately next lower value. This is done for all consecutive pairs of available frequency setting values. The lower frequency value for the pair which is responsible for the maximum drop in temperature is recorded for that experiment. In such a way, these representative frequency setting values are computed for each experiment. The representative frequency setting that was observed the maximum number of times across all experiments was selected as the value of β_j. We have used representative kernels (depicted in Table 2) from the popular OpenCL benchmark suite Polybench [19] with different input sizes. The task set contains candidate kernels representative of typical real-time computation performed

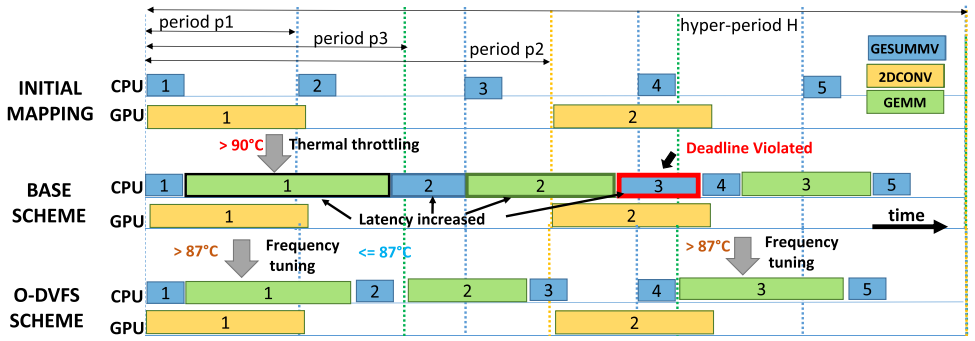

Fig. 9. Mapping schedule for different schemes.

as part of perception systems in automotive domain (neural network pipelines, signal estimators, etc.).

Our proposed framework can automatically design the host code for launching the kernels with the support of a specification file that contains necessary attribute information such as input/output buffers, variables passed as arguments, dimension size, work-group size for each kernel. The specification file has provisions for specifying the arrival time, period and deadline for each task.

7.2 Experimental Results

We compare the following scheduling approaches in our evaluation.

(1) ***BASE***: This approach refers to the default Dynamic Thermal Management (DTM) technique [4] employed by the OS where the processor frequency is drastically reduced by thermal throttling without considering its effect on the performance of the executing tasks. If the CPU temperature goes beyond 90°C on the Odroid XU-4, the CPU frequency setting drops from 2 GHz to 0.9 GHz.
(2) ***Online (O)-DVFS***: In this approach, DTM is done by an online frequency tuning algorithm [25]. The approach initially profiles each task in the task set at the maximum frequency setting in the target platform. The task mapping option for each task that results in minimum latency is computed offline from the profiled data. During online scheduling, the approach always chooses the task mapping with minimum latency for each task. If it is observed that the peak temperature at runtime overshoots 87°C, the frequency level of the A15 cores in the big CPU is reduced by 0.2 GHz. This process of stepwise frequency reduction continues if the peak temperature is still equal to or greater than the threshold, but not below the lowest frequency setting that ensures deadline constraint.
(3) ***Adaptive (A)-DTM***: This is our proposed DTM scheme described in Sections 5–6 that uses different thermal recovery actions discussed earlier to mitigate thermal violations.

We pictorially describe, with the help of Gantt Charts depicted in Figure 9, how task-scheduling decisions are taken for BASE and O-DVFS schemes. We consider the first hyper-period in a given epoch for the case where thermal violation occurs due to modification in the current job set from $\mathcal{J}$ to $\mathcal{J}'$. The job-set $\mathcal{J}$ comprises the following jobs - OpenCL tasks GESUMMV(4096) and 2DCONV(1024×1024) arriving with period values of 6 seconds and 15 seconds respectively. The platform was thermally stable for the schedule $\mathcal{S}$ corresponding to $\mathcal{J}$ in the previous epoch. In the current epoch, no job is removed from $\mathcal{J}$ and the new job set $\mathcal{J}'$, has three additional jobs from OpenCL task GEMM(1024×1024) arriving with task period of 10 seconds along with $\mathcal{J}$. As depicted in the top sub-figure of Figure 9, each job of GESUMMV and GEMM is mapped to CPU, while each job of 2DCONV is mapped to GPU. Each job of a task is represented by a coloured

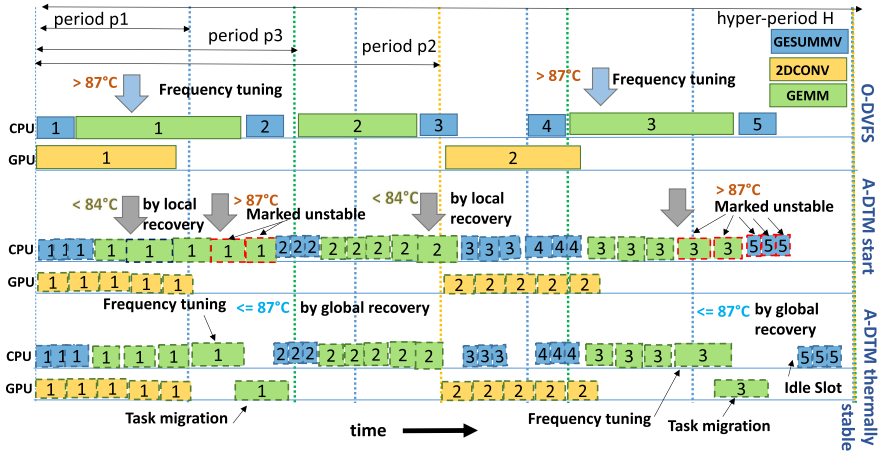

Fig. 10. O-DVFS vs A-DTM : for ADTM, the first hyper-period at start of the current epoch and an eventual thermally stable hyper-period are depicted.

rectangle. The corresponding period is depicted with the help of a vertical dotted line of the same colour. The coloured rectangle pertaining to the p^{th} job of a task is annotated with the number p.

We observe for BASE in Figure 9 that the default thermal throttling gets triggered when the peak temperature reaches 90°C leading to deadline violation of the third job of GESUMMV. In case of O-DVFS, the frequency is gradually reduced in a stepwise manner when the temperature reaches 88°C. When the temperature goes below 88°C, O-DVFS does not reduce the frequency any further.

We next describe how our proposed A-DTM scheme compares with respect to O-DVFS in Figure 10. The CPU-GPU schedule in the top row of the figure reproduces the O-DVFS output for comparison. The middle row provides a snapshot of the first hyper-period of the new epoch started with job injection under A-DTM scheme. The bottom row shows a snapshot of the eventual thermally stable hyper-period in the same epoch as achieved by A-DTM. Also, since A-DTM operates at the granularity of sub-tasks, we represent each constituent sub-task of every job by dotted rectangles. We observe that for A-DTM, in the first hyper-period of the scheduling epoch, both local and global recovery modes get activated. This is indicated by the grey arrows in Figure 10. In case of global recovery, the unstable sub-tasks are marked with red dotted lines and the task mappings are changed in next hyper-periods. We observe that different recovery actions in A-DTM scheme managed to reduce the peak temperature, leading to a thermally stable state.

For this experiment, Figure 11 illustrates the highest peak temperatures the platform reached per hyper-period over a scheduling epoch for each of the three schemes. In the BASE case, the highest temperature in all subsequent hyper-periods is above 90°C. This is because tasks are always mapped to devices at the highest frequency setting, and the default thermal throttling gets triggered when the peak temperature reaches 90°C. In O-DVFS, the frequency is gradually reduced whenever the temperature reaches 88°C. However, the O-DVFS scheme is unable to bring the temperature below 88°C in this case.

In contrast, for our proposed A-DTM approach, the highest temperature reduces per hyper-period after applying different recovery actions and the updated task mapping is propagated to the next hyper-period. We observe from Figure 11, that the maximum temperature recorded for A-DTM is 90°C in the N^{th} hyper-period (beginning of scheduling epoch for $\mathcal{J}'$). This is because no recovery actions are applied in the first hyper-period while accommodating the new jobs in

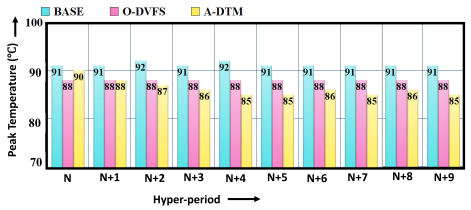

Fig. 11. Peak temperature per hyper-period.

$\mathcal{J}'$ using our greedy mapping policy discussed earlier. In the $(N+1)^{th}$ hyper-period, both global and local recovery actions are active, and the local recovery actions manage to reduce the peak temperature to 88°C($> T_{major}$). In the next hyper-period, the global recovery actions change the task mapping of the unstable sub-tasks. The updated task mappings are reflected in the $(N+2)^{th}$ hyper-period (see Figure 11). Eventually, as can be seen from further hyper-periods depicted in Figure 11, with A-DTM scheme the peak temperature drops to 85°C.

Performance comparison for individual control actions: Our next set of experiments characterize the relative merits and demerits of each recovery action in our proposed framework across a set of diverse task scenarios. This is done by running our framework with different combinations of control actions in the global controller. Given our choice of three possible control actions, we have a total of seven different combinations considering all possible subsets of three recovery action choices. Our experimental setup comprises five test cases each with a different initial job set $\mathcal{J}$ in a thermally stable state and a modified set $\mathcal{J}'$ in a thermal unstable state at the beginning of a new scheduling epoch. We have $\mathcal{J}' = \mathcal{J} \cup \mathcal{J}_{new}$ and $\mathcal{J}_{new}$ contains multiple (or single) instances of a single task. No existing job exits the system. Table 3 describes the details for each of the five test cases in each row. The second column of the table describes each task (SIMD kernel name, input sizes, task period and device mapping) present in the initial job set $\mathcal{J}$ whereas column 6 contains the same details of the task in $\mathcal{J}_{new}$ which is added to the modified job set. The peak temperature T_{peak} and initial CPU and GPU device utilization parameters ($U(H, \delta_C)$ and $U(H, \delta_G)$) for the job set $\mathcal{J}$ at the end of the previous scheduling epoch are represented in the third and fourth column of Table 3. Column 5 contains the hyper-period H length. For simplicity, we have considered that the hyper-period remains the same for both $\mathcal{J}$ and $\mathcal{J}'$. We consider the epoch length to be $10 \times H$. The peak temperature T'_{peak}, and (CPU, GPU) device utilization parameters ($U'(H, \delta_C)$, $U'(H, \delta_G)$) at the beginning of the current epoch for $\mathcal{J}'$ are represented in the seventh and eighth column of Table 3.

For each test case, we create 5 different scheduling scenarios, by modifying the deadline requirement of job instances of the new task added in J'. We model such relative deadline requirements as throughput constraints of individual jobs in $\mathcal{J}_{new}$. The throughput is calculated by the following expression of *throughput index* $TI = (\#FLOPs^{total}_{new}/d_{new})$ where $\#FLOPs^{total}_{new}$ and d_{new} are the total numbers of Floating Point Operations and the deadline of the new task respectively. Five discrete TI values ranging from 6 to 12 (in the order of $1e7$) are selected for our experiments. A lower TI value indicates that the deadline is relaxed, whereas a higher TI value indicates the deadline is stricter.

In each experiment, i.e., for a test case and new task along with job level relative deadline/TI requirement, the global controller is executed in seven possible modes, each representing a combination of recovery actions to reduce the peak temperature over multiple hyper-periods in the epoch. The peak temperature obtained in the last hyper period of the epoch is recorded in

Table 3. Test Case Scenarios

Test Cases	Job set $\mathcal{J}$				Job set $\mathcal{J}'$		
	Kernel in $\mathcal{J}$ with mapped device and task period	T_{peak}	$U(H, \delta_C)$,$U(H, \delta_G$	H	**Kernel in $\mathcal{J}_{new}$ with mapped device and task period**	T'_{peak}	$U'(H, \delta_C)$, $U'(H, \delta_G$
1	2DCONV(512 × 512)->CPU (10s), GESUMMV(2048)->GPU(6s)	79°C	0.25,0.16	30s	GEMM(1024 × 1024)->CPU (30s)	91°C	0.65,0.16
2	GEMM(1024 × 1024)->CPU (20s), GESUMMV(4096)->CPU(8s)	83°C	0.8, 0.17	40s	GEMM(512 × 512)->CPU (20s)	91°C	0.92, 0.17
3	2DCONV(512 × 512)->CPU (10s), GEMM(128 × 128)->GPU (2s), GESUMMV(4096)->GPU(6s)	81°C	0.25, 0.45	30s	2MM(512 × 512)->CPU (10s)	88°C	0.5, 0.45
4	2DCONV(512 × 512)->CPU (5s), GEMM(256 × 256)->CPU (3s), GESUMMV(4096)->GPU(5s), 2MM(64 × 64)->CPU(3s), ATAX(64 × 64)->CPU(6s)	85°C	0.54, 0.5	30s	3MM(512 × 512)->CPU (10s)	91°C	0.81, 0.45
5	**LeNet(32 × 32)**->CPU (5s), 3MM(512 × 512)->CPU (10s), ATAX(256 × 256)->GPU(2s)	82°C	0.35, 0.15	10s	**LeNet(32 × 32)**->CPU (5s)	90°C	0.45, 0.15

each case as an indication of thermal performance achieved through the recovery scheme. We summarize our results in the subplots of Figure 12. Each subplot contains multiple line plots that depict the thermal performance variation of each test case with respect to different TI values. For example, in the top-left sub-plot, we provide peak temperature in last hyper-period (Y axis) as recorded for test case 1, with different recovery schemes and varying TI values for $\mathcal{J}_{new}$ (X axis).

In the **first test case**, the initial device utilization values for initial job set $\mathcal{J}$ depicted in Table 3 are low. When the job set is modified to $\mathcal{J}'$, major thermal violations are detected. For a TI value of 6, we observe that task migration alone was able to reduce the peak temperature from 91°C at the beginning of the epoch to 86°C at the end of the epoch. For the same scenario, idle slot insertion was able to reduce the temperature to only 89°C. Also, with frequency tuning as the sole recovery action, the peak temperature reduced to 83°C. We observe that the performance of idle slot insertion and frequency tuning degrades as TI value increases, i.e. the headspace available for a potentially thermally unstable task reduces. For high TI values, the scope for idle slot insertion becomes limited, and the frequency tuner always sets the frequency value to a high setting in order to meet the target throughput. In contrast, we observe that if the cooler device has low device utilization, task migration is able to reduce the peak temperature more than frequency tuning even at higher TI values. This is because the task (GEMM) in $\mathcal{J}_{new}$ executes faster in the GPU at the highest frequency setting than in the CPU at its highest frequency setting. Task migration only fails when the throughput of the new task is set so high that it is greater than the maximum throughput achievable in the highest frequency setting of the GPU. We observe that the combination of frequency tuning and migration performs better than the other schemes and almost as good as the scheme where all recovery actions are used together. For TI value of 6, we observe that applying all recovery actions together (i.e. our proposed scheme, marked by 'All' in figure legend) resulted in even lower peak temperatures.

In the **second test case**, the CPU has high device utilization whereas GPU has low device utilization. Due to the difference in terms of utilization values between the devices, we observe that at higher TI values, combinations of recovery actions that have migration perform better than idle slot insertion and frequency tuning. Another observation in test case 2 is that temperature reduction by any of the recovery actions is comparatively lower than that of test case 1 for the same scheme. This is due to the higher CPU device utilization $U(H, \delta_C)$ and T'_{peak} values. In the **third test case**, the GPU has higher device utilization compared to CPU and thus has less resource bandwidth for accommodating new tasks. We observe that here frequency tuning outperforms both migration and idle slot insertion even at high TI values. Task migration performs poorly in

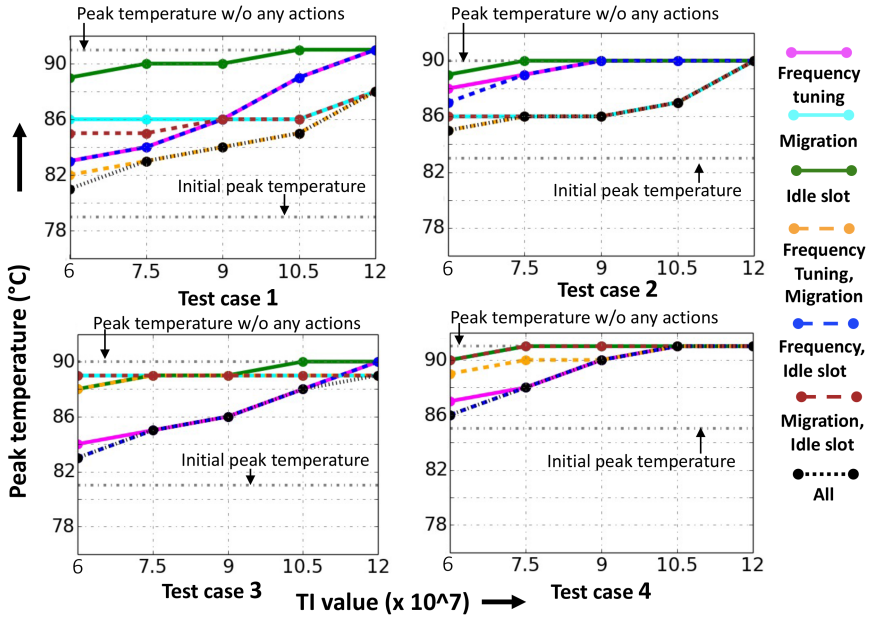

Fig. 12. Peak temperature at different choices of thermal recovery actions and TI value.

this case, as migration from GPU to CPU is not a thermal friendly move in the platform. This is because the CPU is typically always hotter than the GPU due to OS, temperature monitor thread and our scheduler thread continuously running in the little cores along with tasks in the big cores. Also, migration from GPU to CPU is not possible as GPU is already heavily utilized here. Hence, in this case, combinations of recovery actions that contain frequency tuning naturally perform well in reducing peak temperature, except for combination of migration and frequency tuning. In the **fourth test case**, the device utilization for both devices are comparatively higher. Here, we observe that the thermal performance of all the seven possible combinations of recovery actions is lower compared to that of the first three test cases. This is because due to high device utilization in both CPU and GPU ($U(H, \delta_C)$,$U(H, \delta_G)$) and high T'_{peak} values, the headspace to apply any recovery action is comparatively smaller. In this case, frequency tuning is the primary action responsible for thermal management. Our proposed scheme with all actions and the scheme with frequency tuning and idleness insertion performs marginally better than using only frequency tuning for TI value 6.

The primary objective of our framework is to reduce the peak temperature below the thermal threshold by taking suitable actions on unstable tasks, ensuring that deadline constraints are satisfied. It may happen the recovery actions chosen fail to reduce the peak temperature. For example, in test case 2 for TI value 12, we observe that the peak temperature could not be reduced below trip point even after applying all combination of recovery actions. In such cases, when the peak temperature goes beyond the trip point, the default OS governed thermal throttling kicks in and reduces the frequency drastically to 900MHz. Table 4 summarizes the maximum throughput achievable for O-DVFS and A-DTM schemes that manage to avoid invoking OS induced default thermal throttling scheme (BASE). We record the TI values for each test case at which A-DTM and O-DVFS fail to mitigate thermal violation and populate Table 4. We observe from the table that the performance of A-DTM can sustain more throughput index compared to O-DVFS for test case 1 and 2. Whereas, for test case 3 and 4, both O-DVFS and A-DTM can sustain same value of TI.

Table 4. Trip Point Limit

Test Cases	*TI* leading to trip point	
	O-DVFS	A-DTM
1	12	13.5
2	9	15
3	12	13.5
4	10.5	10.5

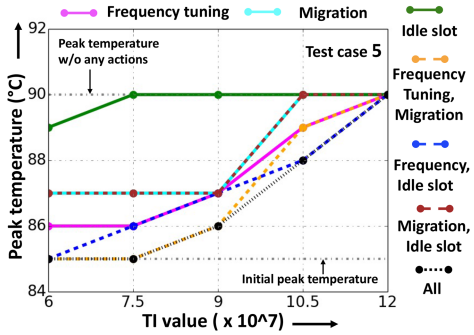

Fig. 13. Peak temperature w.r.t. TI value.

Observing all the four test scenarios each with multiple job insertion constraints in Figure 12 and Table 4, we can conclude that our framework with the combination of three recovery actions perform better than the widely used standalone frequency tuning scheme in terms of reducing peak temperature when GPU device utilization is low. In case of high GPU device utilization, the improvement is naturally lesser.

Insight with Real application: For the fifth test case, we have used *LeNet*, a real-life Convolution Neural Network (CNN) inference pipeline based on the popular LeNet architecture [41]. Each job of the inference task processes an input image of size 32×32. As depicted in Table 3, the initial job set $\mathcal{J}$ comprises jobs of LeNet pipeline along with other standard OpenCL kernels selected from Polybench. The job set $\mathcal{J}$ is modified to $\mathcal{J}'$ by adding new job instances of the LeNet pipeline due to halving of period. In Table 3, this is modelled as addition of another periodic LeNet instance with older period. The results as depicted in Figure 13 clearly establish that the proposed scheme with all three control actions is able to sustain different TI requirements for this job injection scenario with comparatively smaller peak temperature w.r.t. the other combinations having one or two control actions.

Sensitivity with respect to hyper-period length: The performance of our proposed framework is sensitive to the length of hyper-period. In our framework, when a major thermal violation is detected in a hyper-period, the relevant thermal recovery actions are applied in the next hyper-period. The violation mitigation can consume multiple hyper-periods. For large hyper-periods, such actions may thus be slow, thus increasing the time the platform sustains higher temperatures. For same task set, if the hyper-period length is reduced too much due to smaller periodicity, the device utilizations increase. This may reduce the performance of the framework as headspace to apply any recovery action decreases. As a quantitative experiment, we modify test case 1 from Table 3 with changed values of periods of the tasks in job set $\mathcal{J}$ and $\mathcal{J}_{new}$ such that the

hyper-period length is 15 seconds instead of 30 seconds. The initial device utilisation for CPU at the start of the new epoch becomes 0.96 and the peak temperature overshoots platform trip point 90°C after new job injection, thus inducing OS level default thermal throttling over multiple hyper-periods.

8 RELATED WORK

Thermal-aware Scheduling without GPU:. There exist a myriad of thermal-aware scheduling techniques for single-core and multi-core CPU systems that primarily use frequency scaling to reduce power consumption and temperature. Prior research in the field of scheduling real-time tasks focused on DVFS tuning for thermal management while meeting timing constraints for uniprocessor [30] and multiprocessor platforms [5, 28]. Apart from DVFS, mechanisms like thermal aware core mapping [40] and inserting idle periods [5] have also gained traction for dynamically regulating runtime chip temperatures. However, these solutions did not consider the presence of GPUs or their thermal effect on CPUs in integrated heterogeneous platforms.

Thermal Aware Scheduling in CPU-GPU platform: In the recent past, several GPU thermal management techniques have emerged, but mostly for non-real-time systems. Dev et al. [10] highlighted challenges and opportunities in scheduling OpenCL kernels on CPU-GPU platforms by characterizing the thermal coupling effect between CPU and GPU devices. A proactive Dynamic Thermal Management (DTM) policy that uses a thermal model for mitigating thermal violations on a CPU/GPU mobile platform has been reported in [43]. The framework in [46] can automatically select the partitioning point and the operating frequencies for optimal CPU-GPU co-execution under thermal constraints for a given kernel. Prakash et al. [38] proposed CPU-GPU cooperative frequency scaling for mobile gaming to meet thermal constraints with minimum loss in performance. The work in [25] performs online thermal management in CPU-GPU based mobile MPSoCs by partitioning the workload across available devices and choosing suitable frequency settings. However, such work do not consider deadline constraints for real-time systems while applying core-level DVFS.

Thermal Aware real-time Scheduling in CPU-GPU platform: There exist few work that explore thermal aware real-time task scheduling in CPU-GPU integrated systems. These are mostly for static scheduling and do not take care of dynamic scenarios. In [31], the authors proposed a thermal aware CPU-GPU co-scheduling policy that i) performs thermally-balanced task-to-core assignment in CPU cores, ii) prevents power dissipation bursts arising due to thermal coupling between CPU and GPU and iii) maintains timing constraints of tasks under a given chip temperature limit. Isuwa et al. [25] proposed a thermal and energy management mechanism which achieves reduction in thermal gradient through energy efficient resource mapping by thread partitioning of tasks and online frequency tuning in heterogeneous MPSoCs.

Dynamic Scheduling: Given the intricate dependence of application level power consumption and latency performance on architectural parameters, there exist different optimization techniques, heuristics, machine learning (ML), and control theoretic techniques for finding the valid mapping of applications to architectures at runtime. Model-based [9] and rule-based heuristics [12] have been explored for architecture tuning in CPU systems. Such heuristic approaches are usually fast and lightweight, but are not robust in nature. Other approaches use ML techniques for predicting an optimal energy or power efficient configuration within a complicated configuration space [11, 25, 42]. Such ML based approaches handle complex modern processors, modeling an application's latency and power consumption as a function of resource configurations. Based on such models, different scheduling schemes [3, 8] are designed to meet different performance goals. However,

adding a feedback loop to factor in errors due to dynamic changes in the system is difficult in such models. Reinforcement Learning (RL) approaches can learn dynamically and perform adaptive scheduling but are computationally very expensive and not suitable for real-time systems [16, 24].

Control theoretic approaches use feedback to learn system dynamics automatically at run time and are able to adjust resource usage based on the error, i.e., the difference between measured and expected behaviour of system resource usage statistics [22, 23, 33]. Self-adaptive systems based on control theoretic foundations can also provide the user with quantitative guarantees on the time to convergence and robustness in the face of errors and noise. Such approaches have been used to develop adaptive controllers and self-tuning regulators for handling the trade-off between performance and power consumption on heterogeneous SoCs [1, 23, 32]. In this regard, several well known control schemes like Model Predictive Control (MPC) [34], Supervisory control [39] and simple state-space based control [23, 35] have been employed. However, existing approaches are oblivious to the thermal behaviour of the platform.

9 CONCLUSION

We believe the proposed work is the first comprehensive analysis of thermal aware scheduling on heterogeneous CPU/GPU embedded architectures where tasks considered are data parallel in nature. While existing work mostly consider DVFS as the only thermal-aware control action, we extend the OpenCL runtime framework with the flexibility of performing DVFS as well as task migration and task shifting as platform-level control actions. Given the increased complexity of the problem with respect to control actions as well as support for heterogeneity, designing a sound scheduling framework with performance guarantees risks the possibility of increasing the scheduling overhead itself. We thus envision this lightweight heuristic guided approach for generating real-time scheduling decisions. Future work includes generalizing lookup table based methods with a combination of control theoretic and online learning techniques proposed in recent work for heterogeneous scheduling [35]. Integrating other low level platform power domain control methods under this scheduling umbrella is also a promising avenue.

REFERENCES

[1] Tarek F. Abdelzaher, John A. Stankovic, Chenyang Lu, Ronghua Zhang, and Ying Lu. 2003. Feedback performance control in software services. *IEEE Control Systems Magazine* 23, 3 (2003), 74–90.

[2] Hussam Amrouch, Seyed Borna Ehsani, Andreas Gerstlauer, and Jörg Henkel. 2019. On the efficiency of voltage overscaling under temperature and aging effects. *IEEE Trans. Comput.* 68, 11 (2019), 1647–1662.

[3] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*. IEEE, 318–329.

[4] David Brooks and Margaret Martonosi. 2001. Dynamic thermal management for high-performance microprocessors. In *HPCA*. IEEE, 171–182.

[5] 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), 1884–1897.

[6] Ting-Hsuan Chien and Rong-Guey Chang. 2016. A thermal-aware scheduling for multicore architectures. *Journal of Systems Architecture* 62 (2016), 54–62.

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

[8] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *IEEE Transactions on Computer Systems* 48, 4 (2013), 77–88.

[9] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. 2012. Coscale: Coordinating cpu and memory system dvfs in server systems. In *MICRO*. IEEE, 143–154.

[10] Kapil Dev and Sherief Reda. 2016. Scheduling challenges and opportunities in integrated CPU+GPU processors. In *ESTIMedia*. IEEE, 78–83.

[11] Somdip Dey, Enrique Zaragoza Guajardo, Karunakar Reddy Basireddy, Xiaohang Wang, Amit Kumar Singh, and Klaus McDonald-Maier. 2019. EdgeCoolingMode: An Agent based Thermal Management Mechanism for DVFS enabled Heterogeneous MPSoCs. In *VLSID*. 19–24.

[12] Ashutosh S. Dhodapkar and James E. Smith. 2002. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA*. IEEE, 233–244.
[13] Yang Ge, Parth Malani, and Qinru Qiu. 2010. Distributed task migration for thermal management in many-core systems. In *DAC*. IEEE, 579–584.
[14] Anirban Ghose, Soumyajit Dey, Pabitra Mitra, and Mainak Chaudhuri. 2016. Divergence aware automated partitioning of OpenCL workloads. In *ISEC*. 131–135.
[15] Anirban Ghose, Lokesh Dokara, Soumyajit Dey, and Pabitra Mitra. 2017. A framework for OpenCL task scheduling on heterogeneous multicores. *Parallel Processing Letters* 27, 03n04 (2017), 1750008.
[16] Anirban Ghose, Srijeeta Maity, Arijit Kar, and Soumyajit Dey. 2021. Orchestration of perception systems for reliable performance in heterogeneous platforms. In *DATE*. IEEE.
[17] Jairo Giraldo, David Urbina, Alvaro Cardenas, et al. 2018. A survey of physics-based attack detection in cyber-physical systems. *Comput. Surveys* 51, 4 (2018), 1–36.
[18] Dip Goswami, Alejandro Masrur, Reinhard Schneider, Chun Jason Xue, and Samarjit Chakraborty. 2013. Multirate controller design for resource-and schedule-constrained automotive ECUs. In *DATE*. 1123–1126.
[19] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *InPar*. IEEE, 1–10.
[20] Dominik Grewe and Michael F. P. O'Boyle. 2011. A static task partitioning approach for heterogeneous systems using OpenCL. In *CC*. Springer, 286–305.
[21] Heather Hanson, Stephen W. Keckler, Soraya Ghiasi, Karthick Rajamani, Freeman Rawson, and Juan Rubio. 2007. Thermal response to DVFS: Analysis with an Intel Pentium M. In *ISLPED*. IEEE, 219–224.
[22] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. 2004. *Feedback control of computing systems*. John Wiley & Sons.
[23] Connor Imes, David H. K. Kim, Martina Maggio, and Henry Hoffmann. 2015. POET: A portable approach to minimizing energy under soft real-time constraints. In *RTAS*. IEEE, 75–86.
[24] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*. IEEE, 39–50.
[25] Samuel Isuwa, Somdip Dey, Amit Kumar Singh, and Klaus McDonald-Maier. 2019. TEEM: Online thermal-and energy-efficiency management on CPU-GPU MPSoCs. In *DATE*. IEEE, 438–443.
[26] Young Geun Kim, Minyong Kim, Jae Min Kim, and Sung Woo Chung. 2015. M-DTM: Migration-based dynamic thermal management for heterogeneous mobile multi-core processors. In *DATE*. IEEE, 1533–1538.
[27] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. 2013. An automatic input-sensitive approach for heterogeneous task partitioning. In *ICS*. 149–160.
[28] Kai Lampka and Björn Forsberg. 2016. Keep it slow and in time: Online DVFS with hard real-time workloads. In *DATE*. IEEE, 385–390.
[29] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2013. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *PACT*. IEEE, 245–255.
[30] Youngmoon Lee, Hoon Sung Chwa, Kang G. Shin, and Shige Wang. 2018. Thermal-aware resource management for embedded real-time systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2857–2868.
[31] Youngmoon Lee, Kang G. Shin, and Hoon Sung Chwa. 2019. Thermal-aware scheduling for integrated CPUs-GPU platforms. *ACM Transactions on Embedded Computing Systems* 18, 5s (2019), 78–83.
[32] Chenyang Lu, John A. Stankovic, Sang H. Son, and Gang Tao. 2002. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems* 23, 1–2 (2002), 85–126.
[33] Srijeeta Maity, Anirban Ghose, Soumyajit Dey, and Swarnendu Biswas. 2020. Thermal load-aware adaptive scheduling for heterogeneous platforms. In *VLSID*. IEEE, 125–130.
[34] Abhinandan Majumdar, Leonardo Piga, Indrani Paul, Joseph L. Greathouse, Wei Huang, and David H. Albonesi. 2017. Dynamic GPGPU power management using adaptive model predictive control. In *HPCA*. IEEE, 613–624.
[35] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. 2018. CALOREE: Learning control for predictable latency and low energy. In *ASPLOS*. ACM, 184–198.
[36] Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy K. Baruah, and John A. Scoredos. 2010. Mixed-criticality real-time scheduling for multicore systems. In *CIT*. IEEE, 1864–1871.
[37] Aaftab Munshi. 2009. The opencl specification. In *HCS*. IEEE, 1–314.
[38] Alok Prakash, Hussam Amrouch, Muhammad Shafique, Tulika Mitra, and Jörg Henkel. 2016. Improving mobile gaming performance through cooperative CPU-GPU thermal management. In *DAC*. IEEE, 1–6.
[39] Amir M. Rahmani, Bryan Donyanavard, Tiago Mück, et al. 2018. SPECTR: Formal supervisory control and coordination for many-core systems resource management. In *ASPLOS*. ACM, 169–183.

[40] Andrea Rudi, Andrea Bartolini, Andrea Lodi, and Luca Benini. 2014. Optimum: Thermal-aware task allocation for heterogeneous many-core devices. In *HPCS*. IEEE, 82–87.
[41] Pierre Sermanet and Yann LeCun. 2011. Traffic sign recognition with multi-scale Convolutional Networks. In *IJCNN*. 2809–2813.
[42] Gaurav Singla, Gurinderjit Kaur, Ali K. Unver, and Umit Y. Ogras. 2015. Predictive dynamic thermal and power management for heterogeneous mobile platforms. In *DATE*. IEEE, 960–965.
[43] Gaurav Singla, Gurinderjit Kaur, Ali K. Unver, and Umit Y. Ogras. 2015. Predictive dynamic thermal and power management for heterogeneous mobile platforms. In *DATE*. IEEE, 960–965.
[44] Kevin Skadron. 2004. Hybrid architectural dynamic thermal management. In *DATE*. IEEE, 10–15.
[45] Khalid Tahboub, David Güera, Amy R. Reibman, and Edward J. Delp. 2017. Quality-adaptive deep learning for pedestrian detection. In *ICIP*. IEEE, 4187–4191.
[46] Siqi Wang, Gayathri Ananthanarayanan, and Tulika Mitra. 2018. OPTiC: Optimizing collaborative CPU–GPU computing on mobile devices with thermal constraints. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 38, 3 (2018), 393–406.
[47] Chi Xu, Xi Chen, Robert P. Dick, and Zhuoqing Morley Mao. 2010. Cache contention and application performance prediction for multi-core systems. In *ISPASS*. IEEE, 76–86.
[48] Junjie Yan, Xucong Zhang, Zhen Lei, Shengcai Liao, and Stan Z. Li. 2013. Robust multi-resolution pedestrian detection in traffic scenes. In *CVPR*. IEEE, 3033–3040.
[49] Huazhe Zhang and Henry Hoffmann. 2016. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *ASPLOS*. ACM, 545–559.
[50] Qi Zhu, Bo Wu, Xipeng Shen, Li Shen, and Zhiying Wang. 2017. Co-run scheduling with power cap on integrated CPU-GPU systems. In *IPDPS*. IEEE, 967–977.

Received April 2021; revised June 2021; accepted July 2021