Thermal Load-aware Adaptive Scheduling for Heterogeneous Platforms

Srijeeta Maity, Anirban Ghose, Soumyajit Dey
Department of Computer Science and Engineering, IIT Kharagpur
srijeeta.maity@iitkgp.ac.in, {anirban.ghose,soumya}@cse.iitkgp.ernet.in

Swarnendu Biswas
Indian Institute of Technology Kanpur
swarnendu@cse.iitk.ac.in

***Abstract*—Modern-day heterogeneous embedded computing platforms integrate processing elements (PE) with varying compute capabilities on the same die. While such platforms help in delivering high-throughput computation with low power budgets, it also exposes the possibility of violating the thermal envelope. Sustained thermal envelope violation degrades the reliability of the PEs. This work presents an OpenCL run-time extension which adaptively tunes parameters of heterogeneous multicores in order to respect a given temperature constraint without violating real-time deadlines. The feedback-oriented iterative behavior of the proposed run-time extension helps in cancelling out core-level temperature constraint violations which may happen due to dynamic task injection in plug-n-play embedded computing platforms.**

***Index Terms*—thermal envelope, heterogeneous MPSoCs, OpenCL, DVFS, adaptive scheduling**

I. Introduction

Heterogeneous multiprocessor system-on-chip (MPSoC) is an attempt to address the tradeoffs in performance, power dissipation, and energy consumption. MPSoCs, such as Samsung Exynos 5422, provide processing elements (PEs) with different configurations, compute capability, and power dissipation tradeoffs. This can lead to thermal imbalance among the different cores in such heterogeneous MPSoCs. Maintaining temperature constraints for different PEs is important on MPSoCs, otherwise sustained violation of the thermal envelope will degrade the reliability of the PEs [13]. Maximizing performance while meeting the temperature constraint on the MPSoCs is a challenging problem.

Existing power capping techniques in the firmware primarily analyze hardware's current state and scale tunable architectural features, such as frequency and number of active cores, to adjust to the change in power dissipation, while trying to minimize the loss in performance [15]. However, the number of knobs to tune performance and power consumption on modern architectures are too many. For example, an application executing on a CPU-GPU system has the choice of tuning the thread distribution per core, dynamic voltage and frequency scaling (DVFS) setting on each core, and DRAM frequency. Furthermore, the impact of such architectural features on applications in terms of performance and power consumption and core temperature is highly nonlinear, and also depend on the phase of execution of the application [11].

This work considers the problem of software-level online thermal management of heterogeneous MPSoCs executing multiple real-time tasks. This work proposes methodologies that allow run-time schedulers to accommodate high-level requests, such as task injection and increasing the frequency of execution of existing tasks running on a heterogeneous platform, while ensuring that the core temperatures do not violate their respective thermal constraints. Support for such adaptive run-time schedulers is important in many scenarios. For example, there may be an over-the-air update of an automotive platform with a new task getting added. Similarly, given poor lighting conditions, vehicular cameras may have to increase the sampling rate adaptively in a future automobile with the platform task manager being asked to execute CNN-based inferencing of tasks for image recognition at a higher rate. Such high sampling rates may be schedulable as peak system load, but they may violate the thermal envelope leading to reliability issues in the long-term. Thus, the onus is on the platform scheduler and task manager to react to the changes and find a suitable task mapping.

Processor-specific and working-set-aware thermal models are hard to build for every new heterogeneous MPSoC. Hence, we opt for a control-theoretic approach which does not depend on availability of chip-level thermal models. Our proposed approach assumes temperature as an observable quantity with unmodeled dynamics, periodically samples temperature sensors for every core, and tunes architectural as well as application knobs, such as core frequency and thread mapping, which have nonlinear relationships with core temperatures. Our proposed adaptive run-time is thermal-model-agnostic, and is not constrained to a fixed heterogeneous platform.

There exist several approaches with similar goals (Section II). Given the intricate dependence of application-level power consumption and thermal characteristics on architectural parameters, there exists both machine learning (ML) as well as control-theoretic auto-tuning techniques for finding the optimal mapping of single application on architectures. Most existing work rely on multiple profiling runs to characterize application power and performance, and create a ML model with respect to architectural features like core frequency. These performance prediction models and control-theoretic dynamical equations for application speedup and power consumption are employed for tuning architectural parameters *online*. In this regard, several control schemes like Model Predictive Control (MPC) [1], Supervisory control [2], and state-space-based control [8] have been employed. Compared to existing approaches, we summarize our contributions as follows.

The authors acknowledge generous support from DST SERB grant no. ECR/2016/001235, and research grants from Intel, and Qualcomm.

2380-6923/20/$31.00 ©2020 IEEE
DOI 10.1109/VLSID49098.2020.00039

- We consider *multiple* soft real-time tasks executing on a heterogeneous MPSoC instead of a single task.
- We assume a steady state of operation of the system where the run-time scheduler has already found a suitable task-platform mapping with satisfactory thermal characteristics, before *external disturbances* are received. Examples of external disturbances are arrival of a new task and high-level system requests to increase the frequency of an existing task. In such scenarios, our intelligent platform manager controls core frequency and thread migration for temperature-aware real-time task scheduling. Our technique leverages existing methods of learning-based platform-specific task characteristic discovery.
- We build a thermal-load-aware control-theoretic online task scheduler on top of the OpenCL run-time system for easy platform adaptation and porting. Our tool is useful for any heterogeneous multicore platform with core-level temperature sensors and OpenCL support for vector instruction processing (see Section V).

II. Related Work

Of late, there has been significant interest and research in intelligent trade-offs of thermal and power dissipation while minimizing the detrimental impact on performance [3]–[7]. In the following, we focus our discussion on thermal and power management for heterogeneous platforms only. Much prior work [3], [4] focus on energy efficiency and ignore temperature control which can negatively impact performance, reliability and lifespan of the device. Prior work has used design-time thermal optimization using DVFS, but has ignored concurrent utilization of CPU and GPU devices [5]–[7]. Other techniques [9] have considered thermal and energy efficient mapping at run-time using OpenCL on heterogeneous SoC, but ignored dynamic changes in the system such as increase in temperature and injection of new tasks. Such dynamic change in behavior often leads to a considerable difference in temperature across cores.

Control-theoretic techniques provide mechanisms for maintaining desired behavior in dynamic systems with formal guarantees, and are now increasingly being used to develop adaptive controllers and self-tuning regulators for handling the trade-off between performance and power on heterogeneous SoC [8]. These techniques have generalized adaptive control design that exposes the model parameters to the user but do not consider the thermal condition of the platform which can accelerate chip failure and reduce reliability. Power, temperature, and reliability management control strategies for heterogeneous SoCs have been evaluated by [12] on a simulation platform, but not on real hardware.

Several other approaches have turned to Machine Learning (ML) for predicting an optimal energy- or power-efficient configuration within a complicated configuration space [10], [13], [16]. These techniques use ML models to find the optimal resource mapping and thread-partitioning of applications across available cores. These approaches do not consider multiple tasks running at different periods on different cores with their respective soft real-time deadlines, along with the facility of dynamic task injection.

III. Formal Problem Statement

We consider a heterogeneous integrated CPU-GPU processing platform $\mathcal{P}$ comprising of different classes of compute cores. All the compute cores on such integrated platforms reside on the same die and share a common Last-Level Cache (LLC). The platform is considered heterogeneous because each compute core may be of varying computational power in terms of SIMD support, cache size, and clock frequency. We assume that for the platform $\mathcal{P}$, there exists a set of K temperature bands $\mathcal{K} = \{[x_0, x_1], \cdots, [x_{K-1}, x_K]\} = \{B_1, B_2, \cdots, B_K\}$, $\forall i, x_{i-1} < x_i < x_{i+1}$.

Let $\mathcal{T} = \{T_1, T_2, \cdots T_n\}$ denote the set of periodic real-time data-parallel tasks executing with periods $\{p_1, p_2, \cdots p_n\}$. From a power consumption and reliability perspective, we consider that a thermal load specification for the mapping of task set $\mathcal{T}$ on platform $\mathcal{P}$ is expressed as follows. Over the hyper-period $H(p_1, \cdots, , p_n)$ (which is the l.c.m of the periods), the time spent by a core $C_j \in \mathcal{P}$ in the temperature bands of $\mathcal{K}$ is considered to be upper bounded by $\langle u_1 \cdot H, \cdots, u_K \cdot H \rangle$ where u_i represents the fraction of time spent by the compute cores of the platform in the temperature band B_i considering the total window of observation as H. If during a hyper-period H, the time spent in a temperature band B_i is greater than $u_i \cdot H$, we consider it as a thermal violation where our run-time controller needs to execute.

In the steady-state of system operations, we assume that the thermal specification is satisfied by each core until some new task T_{new} is introduced. T_{new} is mapped to P using the initial mapping which simply finds the "best-fit" of T_{new} on a subset of cores in $\mathcal{P}$ that satisfy the deadline constraints of $\mathcal{T} \cup \{T_{new}\}$. This is done by considering the available idle slots in cores or increasing the frequency of core(s) to allow the execution of the modified task set. However, such mapping decisions are thermal-agnostic leading to potential violation of $\mathcal{K}$. Considering this violation as an input disturbance to the system, we propose the design of an *online* controller that runs periodically and executes suitable thread partitioning and frequency tuning decisions such that the violation of the thermal specification is minimized.

IV. Methodology

Given a task-to-core allocation and mapping for the existing task set in a steady state, let us consider that task T_{new} has been injected for execution on the heterogeneous platform. As depicted in Fig. 1, the new task undergoes a profiling phase for certain runs until an initial mapping is determined which satisfies its deadline constraints. A temperature monitor continuously surveys the core temperatures of the platform by discretely sampling the temperature readings of different core sensors at different points of time during the hyper-period H.

If a violation in $\mathcal{K}$ is detected, our proposed methodology invokes a temperature band analysis routine which determines, for each core C_j, the time intervals when C_j operates in different thermal bands. Using this, a working set of intervals

$\mathcal{W}$ is identified which represents the operating region of the controller. The controller decides suitable control actions for tasks executing during the intervals identified in $\mathcal{W}$ and modifies the mapping within $\mathcal{W}$. The controller executes iteratively while consulting with the temperature monitor and attempts to minimize the violation scenario for the chosen working set. Once either a timeout is reached or the violation is resolved, the system resorts to steady state of operations. This entire process of interval identification followed by iterative control is instantiated every time a violation scenario is detected. Next, we explain each routine in detail.

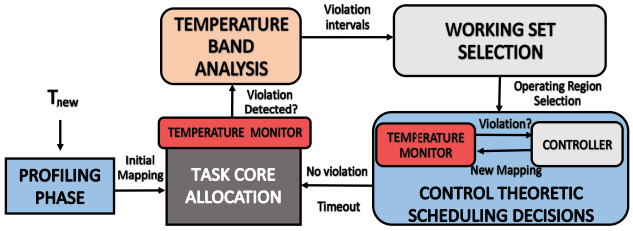

Fig. 1: Methodology Overview

Temperature Band Analysis: The temperature monitor returns a set of discrete temperature readings $t_{C_j} = \{t_1^j, t_2^j, \cdots, t_n^j\}$ for each core C_j over the course of an entire hyper-period H, where $n = \lfloor \frac{H}{h} \rfloor$ with h being the sampling period of the sensor. Each such set is used to construct a thermal band sequence for each core by labelling each discrete reading t_k^j with the corresponding thermal band B_i to which that reading belongs. This sequence of thermal bands is first processed to group contiguously occurring identical bands. From this sequence, we obtain a set of temperature band intervals $I_{C_j}^{B_i}$ for a compute core C_j (as shown in Fig. 2). From this global thermal band information, we greedily choose the highest non-empty band (B_N in Fig. 2) for constructing the operating region of the controller using the working set selection routine discussed next.

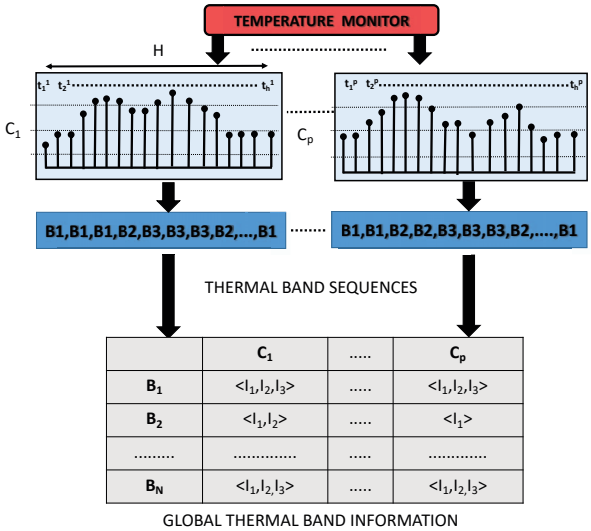

Fig. 2: Thermal Band Generation

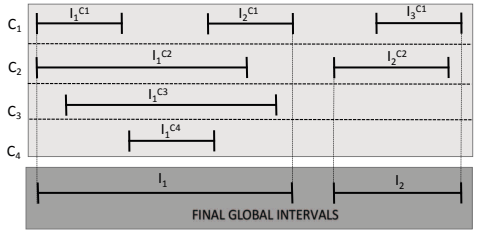

Fig. 3: Working Set Selection

Working Set Selection: This routine creates a set of global intervals spanning across all cores in which the controller will take scheduling decisions. Thermal band intervals generated across all cores that correspond to the highest non-empty temperature band B_N obtained in the aforementioned routine are typically merged in order to reduce the total number of operating regions of the controller. Intervals belonging to each set $I_{C_j}^{B_N}$ for cores $C_j \in \mathcal{P}$ are selected and a union operation is performed following the method discussed as follows. In every iteration, the largest interval I_{max} is first selected from all the sets. This is followed by selecting intervals which overlap with I_{max}. An union operation is applied to these intervals to generate one global interval that will belong to the working set. From the remaining set of intervals, again the largest interval and corresponding overlapping intervals are selected to generate another global interval. This process of selection and union is carried on until no more global intervals can be created. We present an illustrative example in this regard in Fig. 3 where we consider a heterogeneous platform comprising four cores C_1, C_2, C_3, and C_4.

Controller: The controller routine takes control-theoretic scheduling decisions in intervals belonging to the working set $\mathcal{W}$ and accordingly i) reconfigures the starting times of tasks, ii) changes the frequency of cores, and iii) migrates threads of tasks across cores in the heterogeneous platform. In every case, the controller considers task sets whose execution span overlaps with the global thermal band intervals obtained in the working set $\mathcal{W}$. Let $\mathcal{T}^{C_j} = \{T_1^{C_j}, \cdots, T_m^{C_j}\}$ denote the set of all such task sets where each task set $T_i^{C_j}$ comprises those tasks executing on core C_j whose execution span overlaps with some global thermal band interval $I_i \in \mathcal{W}$. The set of all tasks on C_j during the entire hyper-period is denoted by T^{C_j}. The controller functions in three distinct modes each carrying out one of the three possible scheduling decisions discussed above. The decision of mode selection by the controller is carried out by inspecting the number of tasks executing on a core in a global interval as well as the percentage of time a core is idle in the hyper-period. The number of tasks executing during $I_i \in \mathcal{W}$ on core C_j is given by $|T_i^C|$. The idleness of a core during H is given by the idle slots between every pair of consecutive tasks belonging to T^{C_j}. Let us denote the starting time and finishing time of a task $T_r \in T^{C_j}$ as $s(T_r)$ and $f(T_r)$ respectively. The idle slot between a pair of consecutive tasks T_r and T_{r+1} is denoted by $idle(T_r, T_{r+1}) = s(T_{r+1}) - f(T_r)$. The idleness of a core C_j is therefore $idleness(T^{C_j}) = \sum_{T_r, T_{r+1} \in T^{C_j}} idle(T_r, T_{r+1})$. When there is a thermal violation, the controller will appropriately start

execution in one of the three modes.

The primary working principle of the controller is depicted in Fig. 4. The controller is activated the moment a violation in $\mathcal{K}$ is detected followed by $\mathcal{W}$ selection. Note that the control strategy is greedy in the sense that the controller attempts to minimize the usage of the maximum temperature band which should potentially lead to removal of violation occurring in the highest band. In terms of specification, we assume careful and logical choice of $u_1, \cdots, u_k$ from the user in this work, i.e., the higher the temperature band, the lower has to be the value of u. In that case, the greedy control strategy is expected to mitigate potential band violations while prioritizing the higher bands. For an interval $I_i \in \mathcal{W}$, the number of tasks and core idleness is analyzed to determine in which mode the controller should start execution. The controller continues to execute in a particular mode until the violation is mitigated or the controller exceeds the permissible time limit for executing in that mode. The conditions that determine which mode the controller should be running in are discussed next with the help of Table I.

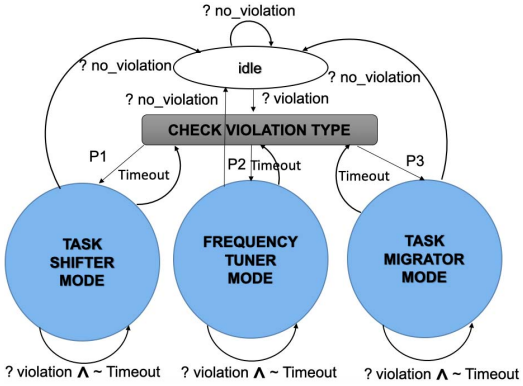

Fig. 4: Controller Automata

We consider a target heterogeneous MPSoC with two distinct compute cores C_1 and C_2, each corresponding to a CPU device and a GPU device respectively. The idea is easy to generalize for more number of heterogeneous devices. Given an interval I_i, the number of tasks in $T_i^{C_j}$ is considered high (H) if $|T_i^{C_j}| \geq x$ and low (L) when $|T_i^{C_j}| < x$. Similarly, the idleness of a core C_j is considered high (H) when $idleness(T^{C_j}) \geq y\%$ of H and low (L) vice-versa; x, y being threshold parameters. Different possible Boolean combinations of number of tasks being H/L and idleness being H/L is denoted by predicates P_1, P_2, P_3 in Table I.

Intuition for Control Action: If it is observed that the core idleness for both the cores in the heterogeneous platform is low, the controller aborts, since the system is completely loaded and it is unable to modify task core mappings without violating deadline constraints. The controller enters into the *Task Shifter* mode if it observes that core idleness is high for both the cores but the number of tasks is small for both the cores in a given interval I_i (i.e., predicate P_1 is True as shown in Table I). This may be attributed to the fact that since the number of tasks is low, shifting one or all of the tasks without violating each of their deadline constraints could potentially increase core idleness leading to a decrease in temperature. The controller enters into the *Frequency Tuner* mode if it observes that both the number of tasks as well as the core idleness for both the cores are high in a given interval I_i (i.e., predicate P_2 is True as shown in Table I). Since the number of tasks is greater here, shifting while maintaining deadline constraints for each and every task becomes cumbersome. Rather it makes sense to reduce core frequency for increasing execution time per task in order to leverage the core idleness during that interval. This reduction in core frequency may potentially lead to a decrease in core temperature thereby removing the thermal violation caused in I_i. Finally, the controller enters into the *Task Migrator* mode if it observes that there exists an imbalance with respect to the number of tasks and idleness between the cores of the heterogeneous platform (i.e. predicate P_3 is True as shown in Table I). The predicate P_3 becomes true for a total of four different combinations of boolean values of core idleness and number of tasks. Given that core idleness is high for both the cores, if it is observed that the number of tasks is low for one and high for the other (the first two combinations for predicate P_3 in Table I), the controller tries to remove this imbalance by migrating some of the threads of tasks executing on the core with more number of tasks to the other core. Again, if it is observed that the core idleness is high for one core and low for the other core, then irrespective of the number of tasks executing in both the cores (the last two combinations for predicate P_3 in Table I), the controller tries to migrate some threads until the violation is mitigated. In all the cases, the final objective is to make the thread distribution of tasks uniform across all the cores of heterogeneous platform. We next discuss the algorithms used by the controller once it enters in each of the three modes.

TABLE I: Predicates and Controller Modes

Predicate	C1 $\lvert T_i^{C_1}\rvert$, $idleness(T_i^{C_1})$	C2 $\lvert T_i^{C_2}\rvert$, $idleness(T_i^{C_2})$	Controller Mode
–	_, L	_, L	–
P_1:	L, H	L, H	Task Shifter
P_2:	H, H	H, H	Frequency Tuner
P_3:	H, H	L, H	Task Migrator
	L, H	H, H	
	_, L	_, H	
	_, H	_, L	

A. Task Shifter Mode: In this mode, the algorithm (Algorithm 1) used by the controller considers for each compute core C_j, every pair of consecutive tasks (T_r, T_{r+1}) belonging to each task set $T_i^{C_j}$ for every interval $I_i \in \mathcal{W}$ and tries to increase the idle slot size between them.

1: **for** each core $C_j \in \mathcal{P}$ **do**
2: **for** $T_i^{C_j} \in \mathcal{T}^{C_j}$ **do**
3: **for** a pair of consecutive tasks $(T_r, T_{r+1}) \in T_i^{C_j}$ **do**
4: $slack(T_r) \leftarrow deadline(T_r) - f(T_r)$
5: $shift(T_r) \leftarrow min(idle(T_r, T_{r+1}), slack(T_r))$
6: $s(T_r) \leftarrow s(T_r) + shift(T_r)$

Algorithm 1: Task Shifter Mode

The slack time available for a task T_r is first calculated by considering its deadline and finishing time (line 4). The

variable $shift(T_r)$ represents the allowable shift for a task T_r and is the minimum of $slack(T_r)$ and $idle(T_r, T_{r+1})$ (line 5). The task is finally shifted by offsetting the starting timestamp of the task by $shift(T_r)$ (line 6). This process of shifting is repeated for each pair of tasks in $T_i^{C_j}$ whenever possible for every interval I_i in the hyper-period H.

B. Frequency Tuner Mode: The algorithm used (Algorithm 2) in this mode considers for each compute core C_j, every pair of tasks (T_r, T_{r+1}) belonging to each task set $T_i^{C_j}$ for every interval $I_i \in \mathcal{W}$ and alters the core frequency for the duration of task T_r.

```
1: for each core C_j ∈ P do
2:     for T_i^{C_j} ∈ T^{C_j} do
3:         for a pair of consecutive tasks (T_r, T_{r+1}) ∈ T_i^{C_j} do
4:             sp' ← get_speedup(s(T_r), f(T_r), idle(T_r, T_{r+1}))
5:             freq ← get_frequency(sp', T_r)
6:             if freq! = NULL then
7:                 Set frequency of core C_j to freq for task T_r
```

Algorithm 2: Frequency Tuner Mode

The algorithm in this mode leverages the current state of the art pole-based self-tuning control techniques [8], [14], which dynamically model speedup of a task as a function of core clock frequency values. Lookup tables that store speedup values of a task T corresponding to clock frequencies of a core C_j are constructed during the profiling phase for each task T_r executing on some core C_j. Given such lookup tables, the control techniques leverage speedup equations of the form $sp(k) = sp(k-1) + (1-\rho)/b \cdot e(k)$, where the speedup $sp(k)$ of a task at time instant k is a function of $sp(k-1)$, the pole of the controller ρ, the base speed of the task b, and error observed $e(k)$. Base speed represents the execution time for a task on a core C operating in minimum clock frequency. The observed error $e(k)$ is basically the difference between desired target speedup and measured target speedup at the k-th time instant. The controller iteratively tries to minimize $e(k)$ by selecting appropriate core clock frequency values in each iteration by referencing the corresponding lookup tables.

In Algorithm 2, for every task T_r, the required target speedup is calculated using the $get\_frequency$ function which considers how much the execution time of task T_r can be increased by considering the idle slot $idle(T_r, T_{r+1})$ (line 4). Given this target speedup value sp', the controller uses the $get\_frequency$ function to access the lookup table and the speedup equation pertaining to task T_r and core C_j and determines a frequency value $freq$. The controller sets the frequency of core C_j to $freq$ for the execution of task T_r in the next hyper-period. This process of core frequency selection is executed for each task in $T_i^{C_j}$ for each interval $I_i \in \mathcal{W}$.

C. Task Migrator Mode: The algorithm for the controller in this mode (Algorithm 3) ascertains from the predicate P_3, which core has a higher utilization i.e. which core has either a higher number of tasks and/or lesser idleness. Let us denote this core as C_{max}. The controller considers migrating partially or completely the threads of tasks belonging to $T_i^{C_{max}}$ during $I_i \in \mathcal{W}$ to the other compute core. For each task $T_r \in T_i^{C_{max}}$ during interval I_i, the controller determines the amount of free slots $avail$ available in the other core during the execution span of task T_r in interval I_i. Given the available bandwidth $avail$, the controller uses the $get\_partition$ function to determine the tuple $th = \langle th_1, th_2 \rangle$ where th_j specifies what fraction of the total number of threads of task T_r should be mapped to C_j.

```
1: for each interval I_i ∈ W do
2:     C_max ← Core with higher utilization as per P_3
3:     for T_r ∈ T_i^{C_max} do
4:         avail ← free slots in the other core during I_i
5:         th ← get_partition(T_r, avail)
6:         if th! = NULL then
7:             Set thread allocation of task T_r to th
```

Algorithm 3: Task Migrator Mode

During the profiling phase for a task, lookup tables are constructed which store the different execution times taken to run different fractions of the total number of threads required for task T_r executing on some core C_j. Using this, the algorithm determines what fraction of the total number of threads can be migrated to fit in the available bandwidth of the other core. This process is repeated for each task executing in $T_i^{C_{max}}$ for each interval in $I_i \in \mathcal{W}$.

V. Experimental Results

We have implemented our run-time extension on the ODROID XU4 embedded heterogeneous platform executing on an Ubuntu 18.04 LTS operating system. The ODROID platform comprises three distinct types of compute cores: (i) quad-core ARM Cortex-A7 (little) CPU, (ii) quad-core ARM Cortex-A15 (big) CPU, and (iii) ARM Mali-T628 GPU. The platform supports a range of operating frequencies and temperature sensors for each of these three devices. For the execution of our control based scheduler, we map the OS to three cores of the quad-core ARM little CPU device, with our scheduler executing on the fourth core. Additionally, we have a temperature monitor executing on the second core of the little CPU which discretely samples the temperature readings from the sensors with a sampling period of 50 ms. For each device, DVFS commands set same frequency for all constituent cores together. The scheduler 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 times of the tasks, and (ii) ascertain device availability through asynchronous callback functions which are instantiated when a device becomes free. Once a violation in $\mathcal{K}$ is detected, the interval analysis and working set selection routines execute and provide operating regions for the control scheme. Next the controller executes and provides task-core settings for the next hyper-period, H_i say. The temperature readings gathered in H_{i+1} are fed back to the controller based on which control decisions are again effected in H_{i+2}. The controller update rate is effectively $2H$ here. The iterations continue until the violation is mitigated or a pre-specified timeout is reached.

For our experimental results, we use instances of a standard tiled single-precision general matrix multiply program (SGEMM) with different square matrix input sizes $N \in$

$\{128, 256, 512, 1024\}$ for generating tasks mimicking multiple heterogeneous workload scenerios. We create different tasksets by choosing among these task instances and assigning them periods in the range [500 ms, 10 s]. Given space constraints, we report here only three specific thermal violation scenarios which establish the usefulness of our mode based control technique. We first consider a scenario where the *Task Migrator* mode executes and plot the respective temperature profiles after the controller starts execution for both the big CPU and GPU devices (shown in Fig. 5). For both the subplots, the x-axis denotes the time elapsed (in seconds) and the y-axis denotes the temperature of the core (in °C). The light blue lines in both the subplots denote the core peak temperatures of the respective devices.

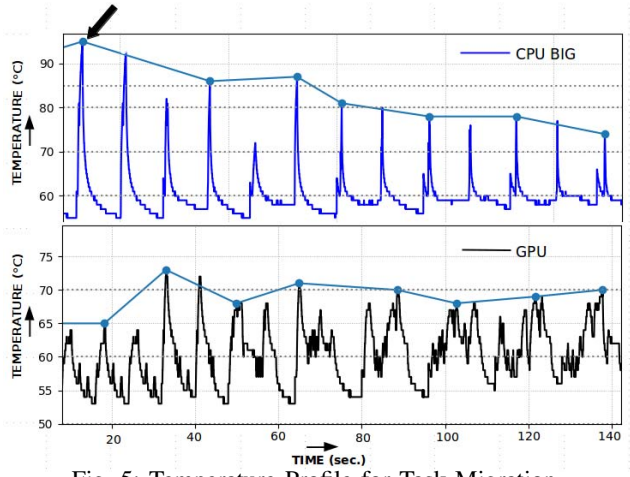

Fig. 5: Temperature Profile for Task Migration

A significantly large SGEMM task processing a large matrix ($N = 1024$) is mapped to the CPU thereby causing the core temperature for the CPU to reach 90°C whereas two SGEMM tasks processing smaller matrices of size $N = 256$ and $N = 512$ are mapped to the GPU device raising the core temperature to around 65°C. The controller moves into the *Task Migrator* mode at time 10 s (shown by the arrow in Fig. 5) and migrates threads of tasks executing on the CPU to the GPU successfully reducing the temperature of the CPU to around 70°C, while the increase in the temperature of the GPU is also limited to 70°C.

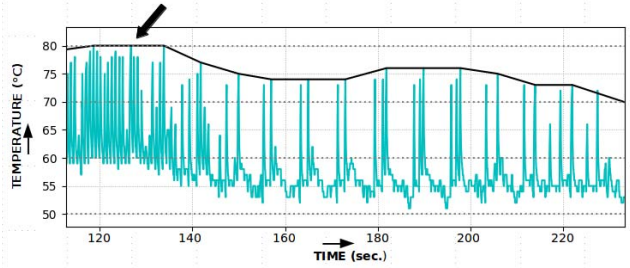

Fig. 6: Temperature Profile for Frequency Tuning

For scenarios where the *Frequency Tuner* and *Task Shifter* modes of the controller execute, we plot the temperature profiles in Figs. 6 and 7 respectively. The plots render temperature profiles for the big CPU device only from the time points when the violations occur. In both scenarios, core idleness is high, while the number of tasks is high for Fig. 6 and low for Fig. 7 resulting in respective modes getting activated. It can be observed for both the cases that the core peak temperatures decrease as the controller starts execution.

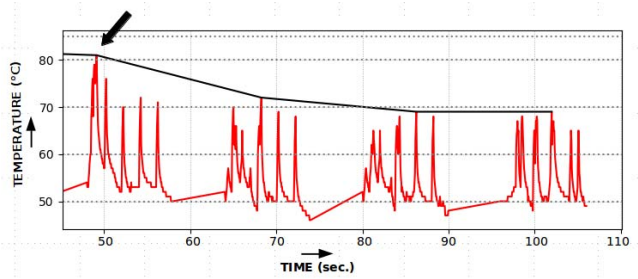

Fig. 7: Temperature Profile for Task Shifting

VI. Conclusion

We provide a control-theoretic methodology for thermal load-aware task management on heterogeneous platforms. Our proposed methodology explores multiple control modes amid the presence of a set of periodic tasks and disturbances that can potentially violate the thermal envelope. We plan to generalize our proposed control scheme by augmenting online learning techniques and control-theoretic decisions [14] and exploring the memory aware thermal management [17], similar to related work on heterogeneous scheduling.

References

[1] A. Majumdar et al. "Dynamic GPGPU Power Management Using Adaptive Model Predictive Control", *HPCA 2017*.
[2] A. Rahmani et al. "SPECTR: Formal Supervisory Control and Coordination for Many-core Systems Resource Management", *ASPLOS 2018*.
[3] A. Prakash et al. "Energy-efficient Execution of Data-parallel Applications on Heterogeneous Mobile Platforms", *ICCD 2015*.
[4] A. Singh et al. "Energy-Efficient Run-time Mapping and Thread Partitioning of Concurrent OpenCL applications on CPU-GPU MPSoCs", *TECS 2017*.
[5] Y. Liu et al. "Thermal vs Energy Optimization for DVFS-enabled Processors in Embedded Systems", *ISQED 2007*.
[6] S. Saha et al. "Thermal-constrained Energy-aware Partitioning for Heterogeneous Multi-core Multiprocessor Real-time Systems", *RTCSA 2012*.
[7] J. Zhou et al. "Thermal-Aware Correlated Two-level Scheduling of Real-time tasks with Reduced Processor Energy on Heterogeneous MPSoCs", *JSA 2018*.
[8] C. Imes et al. "POET: a portable approach to minimizing energy under soft real-time constraints", *RTAS 2015*.
[9] E. Wachter et al. "Reliable Mapping and Partitioning of performance-constrained OpenCL applications on CPU-GPU MPSoCs", *ESRTM 2017*.
[10] S. Isuwa et al. "TEEM: Online Thermal-and Energy-Efficiency Management on CPU-GPU MPSoCs", *DATE 2019*.
[11] S. Mitra et al. "Phase-Aware Optimization in Approximate Computing", *CGO 2017*.
[12] A. Bartolini et al. "A Virtual Platform Environment for Exploring Power, Thermal and Reliability Management Control Strategies in High-performance Multicores", *GLSVLSI 2010*.
[13] S. Dey et al. "EdgeCoolingMode: An Agent based Thermal Management Mechanism for DVFS enabled Heterogeneous MPSoCs", *VLSID 2019*.
[14] N. Mishra et al. "CALOREE: Learning Control for Predictable Latency and Low Energy", *ASPLOS 2018*.
[15] E. Rotem et al. "Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge", *IEEE Micro 2012*.
[16] S. Wang et al. "OPTiC: Optimizing Collaborative CPU–GPU Computing on Mobile Devices With Thermal Constraints", *CADICS 2018*.
[17] M. Rapp et al. "Pareto-Optimal Power-and Cache-Aware Task Mapping for Many-Cores with Distributed Shared Last-Level Cache", *ISLPED 2018*.