Energy- and Temperature-aware Scheduling: From Theory to an Implementation on Intel Processor

Qaisar Bashir[†], Mohammad Pivezhandi[‡], Abusayeed Saifullah[‡]
[†]*AMD, Inc., California*
[‡]*Wayne State University, Michigan*

***Abstract*—Temperature, energy, and performance are the key considerations of real-time multicore systems. Thermal hotspots and high temperatures not only degrade reliability and performance but also increase cooling costs and leakage current. To address these issues, we propose an energy, performance, and temperature-aware semi-partitioned scheduling technique for sporadic real-time tasks. The proposed approach incorporates DVFS, Dynamic Power Management, and chip floor-plan in classical partitioned scheduling technique. To further reduce temperature, energy, hotspots, and gradients, the proposed partitioned scheduling also considers individual task behaviors in the form of hot and cold tasks and applies reactive thermal management techniques such as task migration (hence called *semi-partitioned scheduling*) for system safety. To our knowledge, this is the first paper that considers energy, performance, reliability, and temperature together in real-time multicore scheduling. Finally, we present the results through experiments on the Intel Xeon 2680 v3 multicore platform. The results show that the proposed approach on average saves 15% power, reduces $3^{\circ}c$ average temperature, improves task schedulability, and avoids 100% thermal emergencies compared to the state-of-the-art technique.**

I. Introduction

In multicore technology, thermal management plays a crucial role. Power density is doubling nearly every three years. During processing, heat is produced, and energy is consumed. To remove heat from the processor die, hardware-based solutions are typically used, such as fans. However, it increases cooling costs by almost $3 per watt of heat dissipation [27]. An overheated chip not only increases the cooling costs but also reduces reliability [30]. Moreover, the power consumption of a chip includes the power necessary for normal operation (dynamic power) and the power of invalid electric leakage (static power). The invalid electric leakages account for half of the total power consumption when the cheap temperature exceeds 86°C. A chip's temperature management is crucial for reducing electric leakage and increasing reliability.

An uncontrolled temperature can adversely deteriorate the performance and reliability of the system. Furthermore, spatial gradients (a situation where a core is operating at high temperature as compared to the others), temporal gradients (a situation where a core is overly used as compared to the others), and thermal cycles (a large temperature variation) bring challenges to leakage power, cooling cost, and performance. Moreover, the temperature of the multicore system on the chip heavily depends on energy or power density [12]. Aggressive frequency scaling and increasing system temperature may result in a significant increase in power consumption [28]. Today, real-time embedded systems are exploiting the ubiquitous multicore platforms. For multicore platforms, power-aware designs are important, but localized heating occurs at a much faster rate as compared to chip-wide heating. It increases thermal cycles, thermal hotspots, and temperature gradients that cause physical damage [30]. Therefore, it is important to control both the energy and temperature of multicore systems.

There are many reliability-aware techniques that maintain system reliability, including replication, frequency elevation, and rollback recovery [16], [33], [34]. But these techniques incur high energy consumption and degrade performance. It is important to consider energy consumption, temperature, and problems associated with peak temperatures, including hotspots, temporal and spatial gradients, and thermal cycles, when considering overall system performance. By considering the temporal and spatial gradients of the operating temperatures, power-aware techniques can provide a better performance, energy efficiency, and reliability. In a recent study, researchers identified that a 6°C of temperature reduction could boost chip reliability by two times [19].

Thermal and energy management for multicore systems can be introduced at both software and hardware levels. Scheduling is one of the ways to overcome these problems. So, it is important to introduce a temperature, performance, and energy-aware mechanism for real-time systems. Energy and temperature-aware scheduling techniques have been widely introduced previously but most of the techniques in literature either considered temperature or energy individually [18], [20].

This work addresses real-time scheduling of constrained deadline sporadic tasks on multicore that takes temperature, energy, reliability, and performance into account. A complex interdependence between temperature, performance, and energy makes temperature, performance, and energy-aware scheduling techniques challenging. Each core's temperature profile depends on its power profile, floor plan, neighboring cores' power profiles, and execution pattern of tasks.

In this paper, we focus on energy, temperature, and performance trade-off by exploiting the Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM) capabilities of multicore platforms in real-time scheduling. Our approach regulates the voltage and frequency of each core by taking into account individual task behavior,

thereby enabling the control of energy consumption as well as the temperature of each core. Specifically, we make the following new contributions.

- We observe that a fixed partitioned scheduling may not ensure system reliability under adverse heat effects on a multicore. Therefore, we propose a temperature- and energy-aware semi-partitioned scheduling for real-time tasks on a multicore platform which allows occasional task migration with negligible overhead. It takes into account the temperature profile of each core, floor-plan, neighboring cores' power profiles, the execution pattern of the tasks based on their characteristics, and the individual behaviors in the form of hot and cold tasks.
- We implement the proposed scheduling on the Intel Xeon 2680 v3 multicore platform [1] and evaluate through experiments on Intel Xeon 2680 multicore considering actual workload. To our knowledge, this is the first implementation and systematic experimentation of such real-time scheduling using actual processor workload on a commercial multicore platform. The results show that the proposed approach, on average, reduces power consumption by 15%, reduces $3^{\circ}C$ average temperature, improves task schedulability, and avoids 100% thermal emergencies compared to the state-of-the-art technique.

In the the rest of the paper, Section II reviews related work, Section III describes the system model and problem definition, Section IV discusses the proposed scheduling, and Section V describes our implementation and experimentation.

II. Related Work

While there are many works on semi-partitioned scheduling (see [10]), they do not consider temperature or energy-aware scheduling. Existing temperature-aware techniques are based on prediction [18], model-based stochastic [32], or floor-planning [35]. Their actions are performed only when the operating temperature goes beyond a certain threshold limit, and requests are issued by the scheduler to control frequencies. The work in [22] formulated a constructive speed-based scheduling algorithm and derived thermal feasibility checks for periodic tasks. A response time analysis was studied in [11] considering energy harvesting. It does not consider energy or performance while optimizing temperature.

The issues related to reactive thermal management techniques such as hotspots, thermal cycles, and thermal gradients also adversely affect the reliability of the system. They lower the lifespan of the system [29]. A few reliability-aware techniques were introduced in the past. These techniques are based on frequency elevation [34], rollback [33], and replication [16]. But none of these techniques considered energy as a constraint.

The energy benefits of multicore scheduling were studied in [4], [5], [38]–[40]. These techniques improve energy consumption under performance constraints but do not consider temperature. Some of these techniques are based on DPM and switch cores to low-energy modes to optimize energy consumption. A detailed review of these techniques is presented in [6]. While the DPM-based techniques perform well in thermal emergencies, they drastically reduce system performance.

A number of studies considered optimizing temperature and energy under real-time constraints [7], [9], [25]. The work in [25] is based on genetic algorithms that has high computational and runtime overhead, making it unsuitable for real-time systems in practice. The other approaches are based on core configurations [9] that only provide control on a group of cores. In contrast, our approach regulates frequencies and low energy states of individual cores, which is more effective in controlling temperature and energy consumption. In addition, our approach considers task behavior to ensure system reliability and adopts occasional task migration to control peak temperature and improve performance. To our knowledge, ours is the first work that considers temperature, energy, performance, reliability, and individual task behavior (hot and cold tasks) at once. Furthermore, we have implemented the proposed scheduling on the Intel Xeon platform considering actual workload.

TABLE I
Definition of notations

Symbol	Defintion
ζ_i	Power deviation factor of task τ_i
ρ_i	The time period of task τ_i
χ_i	Dropping priority level of task τ_i
C_i	Worst-case execution time of τ_i
u_i	Utilization of task τ_i
β	Static power
I_L	Leakage current
V_{op}	Operating voltage
T_{ambient}	Ambient temperature
I_0	Current in ambient temperature
J	Junction temperature
RC	Time Constant for damping voltage
V_{tt}	Transistor threshold temperature
T_j	Temperature of the j^{th} core
$P[t_2 - t_1]$	Power consumption in time period $[t_2 - t_1]$
R_{thermal}	Thermal Coefficient
p_{ij}	Impact of power consumption of j^{th} core on i^{th} core
$U(f^i)$	Total power consumption of the respective i^{th} core
C_k^l	Worst case execution time of τ_k at frequency level l
D_i	Deadline of τ_i

III. System Model

A. Task Model

We consider a task set τ consisting of n sporadic tasks with constrained deadlines: $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task $\tau_i, 1 \leq i \leq n$, is a sequence of instructions, and is characterized using five-tuple $(C_i, D_i, \rho_i, \zeta_i, \chi_i)$ representation. We consider a multicore system consisting of m cores where each core is DVFS-enabled. Apart from that, we have also assigned a power deviation factor ζ_i to each task τ_i to designate it as a hot or cold task. ζ is the fraction of core power consumed by a task instance execution. This value varies between 0.01 and 1.0 to indicate 1 to 100 percent deviation of power. A task with $\zeta > 0.5$ is considered as *hot task* and a task with $\zeta \leq 0.5$ is considered as *cold task*. The time period of a task τ_i is denoted by ρ_i. The time period is the minimum duration before a task repeats itself. The deadline of task τ_i is denoted

by D_i, and $D_i \leq \rho_i$. The dropping priority level of each task is denoted by χ_i , the lower this value, the higher the priority to be dropped. The worst-case execution time of task τ_i is represented via C_i. For an individual task τ_i, *utilization* $u_i = \frac{C_i}{\rho_i}$; for task set τ, *total utilization* $u_{\text{sum}} = \sum_{i=1}^{n} u_i$.

B. Power and Thermal Model

A widely used power model for processors [3] models power P as $P = \beta + \alpha s^{\gamma}$, where $\alpha > 0$ is the *switching capacitance*, and γ is a fixed-parameter dependent on the hardware, and $\beta > 0$ is the static power [31]. So the energy consumption of a core at speed s over a time duration $\frac{C}{s}$ at given workload C is modeled as $E(C, s) = (\beta + \alpha s^{\gamma})\frac{C}{s} + \alpha C s^{\gamma-1}$.

The leakage power is a major cause of overall energy consumption below the critical frequency [21]. In practice, this power model cannot be used exactly as it is, due to the temperature dependence of static power. Specifically, static power β is given by $\beta = I_L.V_{\text{op}}$, where I_L is the *leakage current* and V_{op} is the *operating voltage*. I_L can be expressed as shown below [7].

$$I_L = I_0 \cdot \frac{J^2}{T_{\text{ambient}}^2} \cdot e^{\alpha \cdot V_{op} \cdot \frac{J - T_{\text{ambient}}}{J \cdot T_{\text{ambient}}}} \tag{1}$$

In Equation (1), T_{ambient} is the *ambient temperature*, I_0 is the *leakage current* at the corresponding ambient temperature, J is the *junction temperature* (highest operating temperature), and V_{op} is the *operating voltage*. Equation (1) shows that an increase in junction temperature increases I_L. The value of β increases due to an increase in I_L, thereby increasing the system's overall power consumption [13].

For the temperature profile of each core, we first need to derive a relation between the operating voltage and the frequency of the core. The relation between V_{op} and processor frequency f is given by $f = RC.\frac{(V_{op} - V_{tt})^2}{V_{op}}$, where V_{tt} is the transistor threshold voltage or mobility parameter, and RC is the constant that relates the core frequency to the operating voltage level [26]. Based on the model described in [8], we can derive the temperature T_j of the j^{th} core against power consumption in the presence of fixed ambient temperature as $T_j = p_{[t_2 - t_1]} \cdot R_{\text{thermal}} + S_2 + T_{\text{ambient}}$, where $p_{[t_2-t_1]}$ is the power consumption during the time interval $[t_2 - t_1]$, T_{ambient} is the fixed ambient temperature, $T_j(0)$ is the reference temperature at time 0, R_{thermal} is the thermal coefficient, and S_2 is given by

$$S_2 = T_j(0) \cdot p_{[(t_2)-(t_1)]}.R_{\text{thermal}} - T_{\text{ambient}} \cdot e^{\frac{t_2 - t_1}{R_{\text{thermal}} \cdot C}}.$$

The above model is used to calculate the temperature of each core for any duration of time without considering the effect of neighboring cores. But this model is not the exact representation of each core temperature due to the significant impact of neighboring temperatures. Now, to take into account the effect of neighboring cores, we have adopted the model proposed in [29] and derived the constants based on hot and cold tasks upon incorporating ζ based on utilization as follows.

$$\begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ .... \\ .... \\ T_m \end{bmatrix} = \begin{bmatrix} T_{1(\text{initial})} \\ T_{2(\text{initial})} \\ T_{3(\text{initial})} \\ .... \\ .... \\ T_{m(\text{initial})} \end{bmatrix} + \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ .... \\ .... \\ p_m \end{bmatrix} \cdot [A] \tag{2}$$

Here, p_i and T_i are the individual power consumption and individual temperature of the i^{th} core, A is a $m \times m$ matrix given by

$$A = \begin{bmatrix} p_{11} & p_{12} & p_{13} & .... & ... & p_{1m} \\ p_{21} & p_{22} & p_{23} & .... & ... & p_{2m} \\ p_{21} & p_{32} & p_{33} & .... & ... & p_{3m} \\ .... & ... & .... & .... & ... & p_{3m} \\ .... & ... & .... & .... & ... & p_{3m} \\ p_{m1} & p_{m2} & p_{m3} & .... & ... & p_{mm} \end{bmatrix}$$

where p_{ij}, $1 \leq i, j \leq m$, indicates the impact of power consumption of the j^{th} core on the i^{th} core. To calculate these parameters, we have used a similar formula from [17] based on utilization and ζ as follows.

$$p_{ij} = \frac{f^i - f^j}{U(f^i) - U(f^j)}; \qquad U(f^i) = \sum_{k=1}^{n} b_{ki} \cdot C_k^{f^i} \cdot \zeta_k \cdot \alpha s^{\gamma}$$

Here i and j are the indices of individual cores and f^i and f^j are their respective frequencies; $U(f^i)$ indicates the total power consumption of core i; C_k^l is the worst-case execution of τ_k task at frequency level l; b_{ki} is a boolean variable which is 1 if task τ_k runs on core i, and 0 otherwise. The above relation can be used by passing an array of cores as a parameter to see the impact of multiple cores on a single core at once. This power model has considered the impact of neighboring cores to (almost) accurately measure the temperature of each core. Table I summarizes the notations used in this paper.

We consider a multi-core with m identical cores $\{M_1, M_2, ..., M_m\}$, with discrete frequency levels. The chip floor-plan is also an important factor for system reliability and the chip temperature model. Chip floor-plan considers the distance between cores, thickness, width, and many other parameters. Our system model also considers the low energy modes $\{sleep, deepsleep, standby\}$ of each core, and switching time between low energy modes and active or idle mode.

IV. Proposed Semi-Partitioned Scheduling

In a partitioned scheduling technique, each task is statically assigned to some core and not allowed to migrate. In our semi-partitioned scheduling, the currently active job of a task must complete its execution on the assigned core. After its completion, the next job of the same task may occasionally be assigned to another core.

To characterize workload, we need to define *request bound functions (rbf)*. The value of *rbf*(τ_i, L) is the total execution requirement of all jobs of τ_i during interval L (only the jobs of τ_i whose release times are in the interval are considered). An approximation of *rbf* is given in [14] as follows. The constant ξ is the value that relates frequency and worst-case

execution time of a task. Task utilization u_i based on frequency adjustment is: $u_i = \frac{\xi.C_i}{\rho_i}$.

$$rbf(\tau_i, L) = (\xi \cdot C_i) + \frac{\xi.C_i}{\rho_i} \cdot L \tag{3}$$

A. Initial Partitioning

In our approach, the task assignment on an individual core depends on the core frequency and scheduling algorithm. Deadline monotonic (DM) is an optimal algorithm for an individual core for sporadic tasks with constrained deadlines.Therefore, we have used DM for tasks assignment on an individual core.

Assume all tasks $(\tau_1, \tau_2, \cdots, \tau_n)$ are in decreasing priority based on DM and $(\tau_1, \tau_2, \tau_3, ..., \tau_{i-1})$ are already assigned to the cores and a new task τ_i requests core time. Task τ_i can be feasibly assigned to any core $M_k, 1 \leq k \leq m$, that satisfies the following condition (see [23] for proof), where b_{jk} is a boolean variable which is 1 if task τ_j is already assigned to core k (i.e., M_k), and 0 otherwise.

$$(D_i - \sum_{j=1}^{n} b_{jk} \cdot rbf(\tau_j, D_i)) \geq \xi \cdot C_i \cdot \tag{4}$$

We assign the task to the first available core that satisfies Condition (4). The algorithm continues to assign tasks to the available core as long as this condition is satisfied. We switch to the next frequency level and adjust the execution requirements according to the next frequency level if the above condition is not satisfied. We continue to increase frequency levels (up to maximum available frequency) and stop whenever the tasks are feasible. If no frequency level is found where the tasks are feasible (schedulable), Algorithm 3 returns partitioning as failed.

B. Energy Minimization

Our proposed approach computes *rbf* and evaluates Condition (4) at each core. Since increasing the frequency of the whole platform leads to an abrupt increase in power consumption, it selects the core with maximum remaining utilization and increases the frequency of only the individual core when Condition (4) is not satisfied.

Note that a frequency much lower or higher than critical frequency leads to leakage power that becomes the major source of total energy consumption. Hence, for energy efficiency, first we choose the range of frequencies that are closer to (but not lower than) critical frequency due to lower energy consumption in that range. This frequency range is different for each processing platform. We denote the lower and upper bounds of this frequency range by f_{min} and f_{max}, respectively. So our proposed algorithm will not choose a frequency lower than f_{min} in any case and will limit the frequency at f_{max}.

Most of the current techniques distribute workload across the multicore platform and continue to operate all cores. In our approach, the objective is to minimize the number of cores for task assignment and keep the unused cores in low energy modes. Hence, when we assign tasks to a core, we continue assigning workload as long as it is feasible up to the maximum frequency f_{max}. As shown in Table II the power values of Intel PXA270 processor at different frequency levels [2], placing a core in low energy mode consumes much less power compared to running/active or idle mode. When the task is not feasible on available running cores, we move a core from low energy mode to running mode and assign the task to it.

TABLE II
INTEL PXA270 POWER CONSUMPTION AT DIFFERENT FREQUENCY LEVELS

Levels	Frequency (MHz)	Active Power(W)	Idle Power(W)
Level 1	624	0.925	0.260
Level 2	520	0.747	0.222
Level 3	416	0.570	0.186
Level 4	312	0.390	0.154
Level 5	208	0.279	0.129
Level 6	104	0.116	0.064

C. Handling Temperature Threshold and Reliability

In Section III, we have shown how temperature increases due to the change in frequency and impact of power consumption on temperature. In Section III, Equation (2) shows the neighboring cores' effect on a core's individual temperature. As we know, every processor has a maximum temperature threshold value. A thermal hotspot appears when the temperature of any core $(T_1, T_2, ..., T_m)$ is greater than the maximum temperature threshold that may cause permanent failure or stall the system. We propose a task migration policy to handle temperature and associated problems. We have defined multiple threshold levels to safely operate below the maximum temperature threshold. $T_{\text{threshold}}$ represents the maximum threshold temperature allowed. We have defined the $T_{\text{threshold}(2)}$ and $T_{\text{threshold}(1)}$ just below the maximum threshold temperature to avoid thermal violations ($T_{\text{threshold}} > T_{\text{threshold}(1)} > T_{\text{threshold}(2)}$). Multiple threshold levels are also beneficial for task migration policy. Task migration policy is proposed to control peak temperature.

We allow migration of tasks between cores, if a core's operating temperature $T_i, 1 \leq i \leq m_1$ (where m_1 is the number of active cores, and $1, 2, \cdots, m_1$ are the indices of active cores) is close to $T_{\text{threshold}}$ or the core's workload is so small that it can be shifted to other cores, while ensuring that all tasks meet their deadlines and the cores operate in frequency range $[f_{\text{min}}, f_{\text{max}}]$ on targeted cores. The benefit of task migration is that either we can control the peak temperature by migrating tasks or we can move some cores in low energy mode by shifting their workload to the cores whose $T_i, 1 \leq i \leq m$ (m is total number of cores) is less than $T_{\text{threshold}(2)}$ to save overall power consumption. The proposed approach also considers the case when no such core exists that can accommodate the complete workload of the core (whose workload is selected for migration). Then our approach shifts a limited number of tasks by meeting feasibility while operating at a range of frequency $[f_{\text{min}}, f_{\text{max}}]$. The task that is selected for migration must complete its currently active job on the current core. The next job will migrate to a different core.

Let τ_d is the set of tasks of core M_i that needs to migrate
m_1 is the number of active cores that are operating in frequency range $[f_{\min}, f_{\max}]$ and operating temperature is $< T_{\text{threshold}(2)}$
Step 1: Sort active cores based on operating temperature
Step 2: for each task in τ_d check Condition (4) on m_1 cores
Step 3: **if** *conditions satisfied;* **then** Assign task to corresponding core and update corresponding *rbf*;
Step 4: **if** *conditions not satisfied and frequency is* $< f_{max}$ *of any core among* m_1*;* **then** update m_1 and increase frequency level by 1 step of m_1 cores and move to Step 2;
Step 5: **if** τ_d *empty;* **then** move core in low energy mode;
Step 6: **if** τ_d *not empty;* **then** continue to execute remaining tasks;
Step 7: Exit;

Algorithm 1: Migration of task (When a core has small workload)

Algorithm 1 shows the first scenario of task migration (when a core has a small workload). It first finds the list of active cores (m_1 cores) that are operating in the frequency range $[f_{\min}, f_{\max}]$ and whose operating temperatures T_i are below $T_{\text{threshold}(2)}$ except the core whose workload is selected for migration. Step 1 sorts the available cores in decreasing order of T_i. Step 2 selects each task one after another from the core and considers it as a new task and assigns the task to the first available core that satisfies Condition (4). Steps 3–6 shift the core (whose workload is selected for migration) in low energy mode on a feasible assignment of all tasks. Otherwise, the core continues to execute the remaining tasks.

Now we consider the second scenario of task migration when a core operating temperature approaches $T_{\text{threshold}}$). To control $T_{\text{threshold}}$, Algorithm 2 checks the temperature of each core ($T_i, 1 \leq i \leq m$) as shown in Equation 2, and if T_i is below $T_{\text{threshold}(2)}$, the core continues to operate at the same operating frequency. If the operating temperature of any core is between $T_{\text{threshold}(2)}$ and $T_{\text{threshold}(1)}$, Algorithm 2 finds the list of active cores that are operating in the frequency range $[f_{\min}, f_{\max}]$ with operating temperatures below $T_{\text{threshold}(2)}$.

Migration overhead. The proposed task migration does not impose special latency overhead since with task migration instructions will be redistributed through the change of the program counter (PC) in different threads. This reconfiguration of PC value takes a few clocks for flush and stall of the ongoing instructions. Since the clock is in term of gigahertz, the latency overhead is negligible.

D. Switching Mechanism and Incorporating Break Even Time

As we have seen, the proposed approach moves a core from low energy mode to running mode and assigns the task to the core. However, if a core is in low energy mode, it cannot start executing the workload right away. It needs a considerable amount of time before starting the execution of an assigned task based on low energy mode (*sleep*, *deepsleep*, *standby*).

Figure 1 shows the Intel Marvell's XScale PXA270 processor switching time between different states. Its mechanical and thermal specification is publicly available in [2], but we cannot extract the same information from the available

Let τ_d be the set of tasks of core M_i that needs to migrate
m_1 is the number of active cores that are operating in frequency range $[f_{\min}, f_{\max}]$ and operating temperature is $< T_{\text{threshold}(2)}$
Step1:**if** *(*T_i *and* $T_{threshold(2)}$ *and* T_i *and* $T_{threshold(1)}$ *and* $m_1 \neq 0$*)* **then** select a task from τ_d with maximum ζ_i;
Step 2: for each task in τ_d check Condition (4) on m_1 cores
Step 3: **if** *conditions satisfied;* **then**
  Assign task to corresponding core and repeat step 1 to 3 unless temperature is below $T_{\text{threshold}(2)}$
Step 4: **else if** *conditions not satisfied and frequency is* $< f_{max}$ *of any core among* m_1*;* **then**
  update m_1 and increase frequency level by 1 step of m_1 cores and move to Step 1
Step 5:**else if** m_1*=0 and* $T_i > T_{threshold(2)}$ **then**
  Select the task from τ_d with maximum χ_i and drop the current job of the task
  Continue to repeat Step 5 unless temperature is below $T_{\text{threshold}(2)}$;
Step 6: **else**
  go to step 7;
Step 7: Exit;

Algorithm 2: Migration of task (When Operating temperature is close to $T_{\text{threshold}}$)

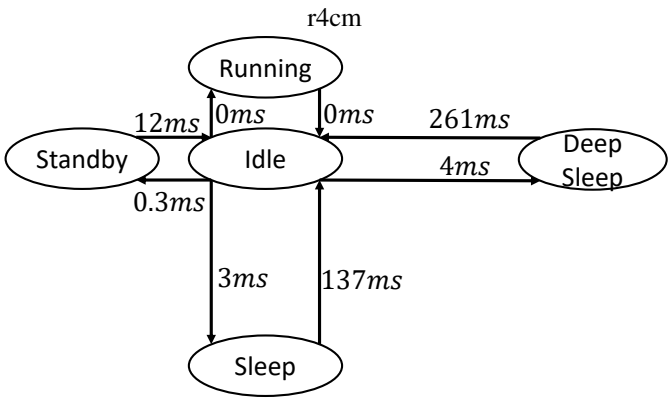

Fig. 1. PXA270 State Switching.

Intel Xeon 2680 datasheets. Since the ratio of the PXA270 state transitions, e.g., the time to transfer from deep sleep to idle v.s. the time to transfer from standby to idle state, is similar to the Intel Xeon 2680 v3 platform, the simulation and behavior analysis is based on the constraints extracted from this platform. The same result can be extracted when scaling the timing according to the results extracted from experiments.

The processor must move from low energy mode to idle or active/running mode to accommodate that task. In Figure 1, we observe that time required to move a processor from low energy to idle or active mode is much larger than the time required to move a processor to low energy mode from the active mode. To accommodate the switching times between states, we have used the concept of *break-even time* (BET) defined as $BET = t_{\text{low}} + t_{\text{wake}}$, where t_{low} is the total time required to move to low energy mode, t_{wake} is the wake-time (time needed to move from low energy to idle/active mode).

The value of t_{low} varies depending on the low energy state. For example, the values of t_{low} for *standby*, *sleep*

and *deepsleep* for Intel processor PXA270 are 0.3 ms, 3 ms and 4 ms respectively. The values of t_{wake} are much higher compared to t_{low}. The corresponding values of t_{wake} against *standby*, *sleep* and *deepsleep* are 12 ms, 137ms and 261 ms respectively as shown in Figure 1. Figure 1 shows 0 ms is required to move a processor from idle mode to running/active mode. So we can conclude that a processor starts execution of a task right away if it is in running mode or idle mode. One greedy approach is that we move a core to low energy mode as soon as it is idle. This approach only works well when the system has very small break-even time. In contrast, the proposed algorithm moves a core (if available) from low energy mode to idle mode as soon as one of the running cores approaches the upper limit of frequency range or operating temperature of the core is close to threshold. So a newly arrived or migrated task can start execution on the available core instantly. The proposed technique moves the core in opposite direction only if the core workload can be feasibly assigned to other active cores with $T_i < T_{\text{threshold}(2)}$ within the defined frequency range.

The sporadic task model is denoted by $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$ m is the number of cores and n is the total number of tasks
PARTITION(τ, m)
for each task in τ
Step 1: m_1 := get list of active cores
Step 2: **while** $m_1 < m$ *and frequency of all active cores* $< f_{max}$ *and* $T_i < T_{threshold(2)}$ **do**
  Step 3: sort m_1 in increasing order based on $T_{i \in m_1}$
  Step 4: **if** τ_i *satisfies Condition (4)* **then**
    assign task τ_i to the core; update assigned task utilization on selected core;
    proceed to next task;
  Step 5: **if** τ *is empty* **then**
    go to step 13;
  Step 6: **if** *Condition (4) is not satisfied and frequency of any active core is* $< b$ **then**
    update m_1 and choose core with minimum T_i;
    update frequency of selected core and go to step 2;
  Step 7: **else**
    Move a core from low energy to active mode; Move to step 2;
Step 8: **while** *no of active cores=m and* $T_i < T_{threshold(1)}$ **do**
  Run task migration algorithm;
  Step 9: **if** τ_i *satisfies Condition (4)* **then**
    assign task τ_i to the core;
    update assigned task utilization on selected core; proceed to next task;
Step 10: **while** $T_{i \in m_1} > T_{threshold(1)}$ **do**
  select the core with maximum T_i; drop task with maximum χ_i value from selected core; Continue to drop tasks unless temperature is below $T_{\text{threshold}(1)}$ **if** $T_i < T_{threshold(1)}$ **then**
    Move to Step 8;
Step 11: **if** τ *is empty* **then**
  go to step 13;
Step 12: **else**
  return Partition Failed;
Step 13: return Partition Successful;

Algorithm 3: Semi-Partitioned Scheduling Technique

E. Handling Power Deviation and Extreme Thermal Condition

Our proposed approach also considers power deviation factor ζ to control peak temperature and temperature balance among the cores. ζ is used to differentiate between *hot* and *cold* tasks. This categorization represents the static power behavior of cores. For example, hot tasks indicate that the temperature is above some threshold in steady state. We keep tracking the power deviation factor of tasks assigned to each core. This information is used for task assignments and migrations. For example, if the core operating temperature is near the threshold level and one of the tasks needs to migrate from the core to reduce the temperature, the task with a maximum value of ζ is selected to migrate based on target core feasibility. Migrating a hot task from a core leads to a sharp decrease in corresponding core operating temperature. To avoid spatial gradients and temporal gradients, the proposed approach uses two counters for each core to keep track of how many times a core is selected and how long it remains active. We use these counters to ensure that all cores are equally used.

The proposed approach also considers extreme thermal emergencies. We used a task migration mechanism to control peak temperature, but a situation can arise in which no core is available to migrate tasks feasibly and all cores are operating near $T_{\text{threshold}}$. To handle this scenario, we assigned a task dropping priority χ_i to each task. The dropping priority varies between 1 to 6. The lower χ_i value shows the higher priority of the task. The proposed approach drops tasks, when $(T_i, 1 \leq i \leq m_1)$ of any core is greater than $T_{\text{threshold}(1)}$. The proposed approach dropped the task with the lowest dropping priority from the selected core. The purpose of dropping tasks with low priority is to avoid thermal violations that may lead to permanent failure of the device. The proposed approach continues to drop tasks unless the operating temperature of the core is below $T_{\text{threshold}(1)}$.

We choose dropping instead of halting to avoid complication that arises when skipping the jobs. For dropping the tasks, we get the Linux process ID of each process and assign the priorities on each ID manually according to the process functionality. However, in halting process, skipping/dropping only the affected jobs requires significant effort as we have to save the current state of the process and the associated variables. Besides, the Intel memory architecture does not provide the memory unlocking mechanism which makes halting without knowing the current state of the execution infeasible. This means the execution of the process should start from the beginnings for the user cores when dropping the jobs.

Task assignment under temperature and energy constraints according to the proposed semi-partitioned scheduling technique is shown in Algorithm 3. Step 1 FINDS the list of the active number of cores. Steps (2-4) show the process of task assignment when the active cores are less than the total available cores and operating in the frequency range. Frequency scaling on the individual core is shown in step 6. Step 7 moves a core from low energy mode to active mode if all active cores are operating at the upper limit of the frequency range if we still have some cores in low energy mode. Steps 8 and 9 show the task migration mechanism. Steps 10–13 show the task dropping process.

TABLE III
INTEL XEON E5 2680 AVAILABLE FREQUENCIES (GHz)

Core Frequency	AP uncore frequency	PP uncore frequency
2.5	2.2	2.1
2.4	2.1	2.0
2.3	2.0	1.9
2.2	1.9	1.8
2.1	1.8	1.7
2.0	1.75	1.65
1.9	1.65	1.55
1.8	1.6	1.5
1.7	1.5	1.4
1.6	1.4	1.2
1.5	1.3	1.2
1.4	1.2	1.2
1.3	1.2	1.2
1.2	1.2	1.2

TABLE IV
INTEL XEON E5 2680 BASIC CC-STATES

ID	Name	Functional Description
C0	Active/Running	Core is executing instructions
C1	Halt/Idle	No execution, but return to C0 instantaneously
C1E	Auto Halt	Halt+DVFS(with lowest available frequency)
C2	Stop Clock	Same as C1, but requires longer time to C0
C3	Sleep	Require considerable long time to move to C0
C6	Power Gating	BCLK is off and remove core voltage

V. EXPERIMENT ON INTEL XEON 2680 MULTICORE

This section provides details of the implementation and systematic experimentation of real-time scheduling using actual processor workload on an Intel Xeon 2680 v3 platform [1]. By evaluating our technique on a 12-core Intel Xeon 2680 v3 processor, we were able to overcome the shortcomings of single-core simulations. Intel Xeon E5 2680 v3 supports per-core P-states. P-states allow changing the frequency and voltage level of the core [15]. Intel Xeon E5 2680 v3 supports multiple sets of P-states that correspond to different operating points (frequency-voltage combinations). P-states can be either hardware-controlled (Intel speed shift technology) or OS-controlled (Intel speed step technology). P-states work based on frequency, where the corresponding voltage level is automatically selected [24].

```
void task1(void *)
{ while(1)
    { clock_t wakeup = clock() + 50;
//elapsed time(50)
        while(clock() < wakeup)
        Sleep(50); } }
int main(int, char**)
{ int ThreadNr;  int process=30;
    for(int i=0; i < process; i++)
        _beginthread(task1,0,&ThreadNr);
    (void) getchar();
    return 0; }
```

Algorithm 4: Workload Generation Program

Another unique feature of Intel Xeon E5 2680 v3 is uncore frequency scaling (UFS). UFS is independent of the core frequencies. UFS enables the processor to control the frequency of uncore components (e.g., RAM, caches) without the interference of core frequencies. UFS can be specified by using the Model Specific Register (MSR) [36], [37]. The work in [15] derived lower bounds of active processor (AP) uncore frequencies and passive processor (PP) uncore frequencies, and showed that uncore frequencies can be used to limit power consumption. We have used uncore frequencies to differentiate between hot and cold tasks. A hot task will use a higher UFS value than a cold task. The available frequency levels are shown in Table III.

Intel Xeon E5 2680 v3 also supports per core C-states. C-states represent the power-saving states. They are used to power down a subsystem that is not executing any workload. C-states are further divided into package C-states (PC-states) and core C-states (CC-states) [37]. We cannot control PC-states as we cannot interfere the packages. We only control CC-states due to interaction with cores. Many other C-states involve hyper-threading, but we focus on basic CC-states [36]. The basic CC-states are shown in Table IV.

A. Workload Generation

System workload is an average core usage over a certain time duration. At any given time, a core either executes a task or remains idle. Generating an exact workload is difficult as the system threads never guarantee time at millisecond-level granularity. Most of the tools for workload generation use one process per-core to stabilize utilization. Algorithm 4 shows our dynamic program to generate a system workload with multiple processes per-core. We used the $getchar()$ function to block the program at any time. This function does not consume any time and allows to end the program by pressing any key. The program allows changing the number of processes and elapsed time (milliseconds) to adjust system utilization. The program generates stable utilization with +/- 2%.

B. Analysis of Power Consumption

We compared our proposed technique with TBP in terms of time spent in lower energy states. The technique that keeps cores in higher energy states consumes more power. Table V shows time spent by each core in CC-states at different utilization factors. At 25% utilization factor, the proposed technique on average spent 19% more time in lower CC-state as compared to TBP. Our technique spent more time in lower CC-states because it controls cores individually and only moves a core in C0 or C1 when the already running cores are out of good frequency range or due to high core temperature as defined in IV-D. At 50% and 75% utilization factors, the proposed technique on average spent 17% and 1.8% more time in lower energy states, respectively, as compared to TBP.

Figure 2 shows the power consumption results of Intel Xeon 2680 v3 platform. The technique that spent more time in lower CC-states saves more power. Our technique on average saves

TABLE V
TIME SPENT IN CC-STATES PROPOSED TECHNIQUE VS TBP

Utilization	25%						50%						75%					
Algorithm	Proposed Technique			TBP			Proposed Technique			TBP			Proposed Technique			TBP		
States	C0%	C1%	C2-C3%	C0%	C1%	C2-C3%	C0%	C1%	C2-C3%	C0%	C1%	C2-C3%	C0%	C1%	C2-C3%	C0%	C1%	C2-C3%
Core 01	25.6	10.3	64.1	28.4	34.6	37.0	52.4	13.2	34.4	54.4	27.3	18.3	76.4	21.6	2.0	81.2	18.7	0.1
Core 02	25.4	11.4	63.2	22.4	33.8	43.8	53.2	10.9	35.9	51.6	26.6	21.8	77.7	21.4	0.9	77.8	21.6	0.6
Core 03	30.4	13.6	56.0	21.6	30.6	47.8	51.6	13.4	35.0	53.5	30.7	15.8	81.3	18.2	0.5	76.5	23.4	0.1
Core 04	29.3	11.9	58.8	27.3	31.4	32.2	47.8	11.6	40.6	52.4	31.9	15.7	80.9	17.6	1.5	80.4	19.4	0.2
Core 05	26.6	13.7	59.7	27.2	32.4	40.4	56.4	12.9	30.7	51.7	34.0	14.3	82.3	15.4	2.3	79.8	19.8	0.4
Core 06	34.6	14.3	51.1	29.7	29.6	40.7	53.9	13.7	32.4	47.9	32.7	19.4	81.7	16.3	2.0	83.6	16.1	0.3
Core 07	28.7	13.3	58.0	26.6	33.4	40.0	51.6	14.4	34.0	51.4	30.3	18.3	83.2	14.2	2.6	82.2	17.4	0.4
Core 08	29.6	11.3	59.1	27.4	37.4	35.2	53.7	12.3	34.0	53.4	33.4	13.2	84.1	13.7	2.2	85.4	14.5	0.1
Core 09	32.8	13.2	54.0	28.2	26.6	45.2	52.8	10.6	36.6	57.4	31.6	11.0	83.1	13.4	3.5	79.3	20.1	0.6
Core 10	28.4	12.4	59.2	26.4	32.7	40.9	58.4	11.7	29.9	55.2	30.5	14.3	83.2	13.9	2.9	82.7	16.6	0.7
Core 11	23.6	13.1	63.3	25.4	40.4	34.2	59.2	9.6	31.2	53.3	29.6	17.1	81.2	16.4	2.4	81.7	18.1	0.2
Core 12	31.4	10.4	58.2	25.3	33.4	41.3	51.3	13.4	35.3	51.4	29.3	19.3	80.3	17.2	2.5	83.3	16.3	0.4
Average	28.9	12.4	58.7	26.3	33.0	39.9	53.5	12.3	34.1	52.8	30.6	16.5	81.3	16.6	2.1	81.1	18.5	0.3

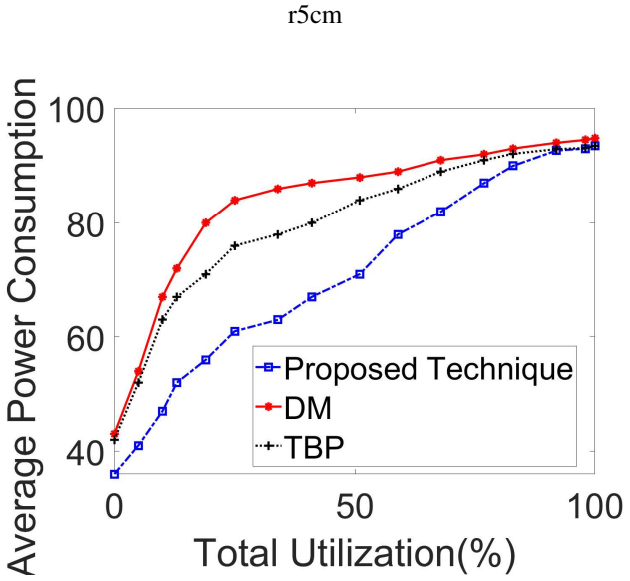

Fig. 2. Power Consumption under Varying Utilization.

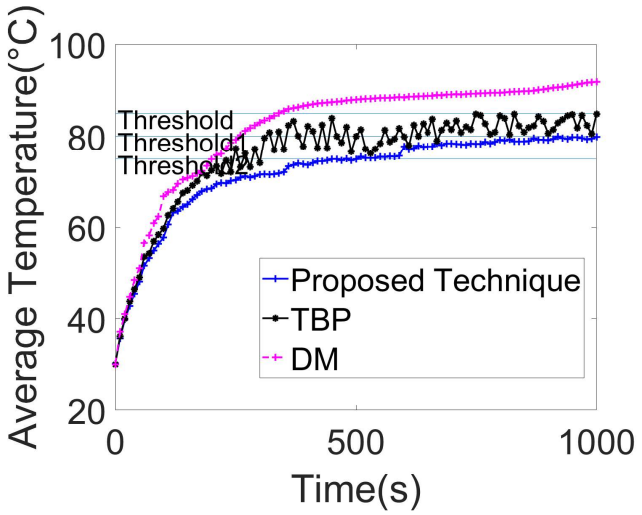

Fig. 3. Average operating temperature of the system at 60% utilization under different utilization policies on Intel Xeon 2680.

15% and 20% more power as compared to TBP and DM (with threshold), respectively.

C. Analysis of Temperature

Our approach not only reduces the overall power consumption of the system but also reduces the average temperature of the system. Figure 3 shows the average operating temperature of the system at 60% utilization. We utilized multiple temperature threshold levels as defined in IV-C. The proposed approach maintains a lower average temperature by operating all the cores at a good range of frequency. DM continues to execute workload without using low energy modes. Although TBP also supports low energy modes, it mostly operates at a maximum frequency that leads to a sharp increase in temperature. Table V shows that our technique spent more time in low energy modes as compared to TBP that further reduces the overall temperature of the system.

Figure 3 also shows that the average temperature of the system under proposed technique remains below the maximum threshold temperature. While some existing techniques drastically reduce the temperature when it is close to the maximum threshold, a sudden decrease in temperature produces large thermal cycles (temperature variation) and also affects the performance of the system. The large thermal cycles directly reduce the reliability of the system and may cause permanent damage. Our approach shows smooth behavior by operating at a good range of frequency. TBP executes workload at a higher frequency that creates large variations in operating temperature. Figure 3 also shows that the operating temperature of DM (without threshold) continues to increase and may cause permanent damage to the system.

Operating temperature of individual cores under our proposed technique is shown in Figure 4. Our approach not only keeps average temperature of the system under maximum threshold but also keeps the operating temperature of individual cores below the maximum threshold ($T_{\text{threshold}}$).

VI. CONCLUSION

We have proposed a semi-partitioned scheduling technique for real-time tasks under temperature, energy, and performance constraints on multicore. We have also implemented and evaluated the approach on the Intel Xeon platform.

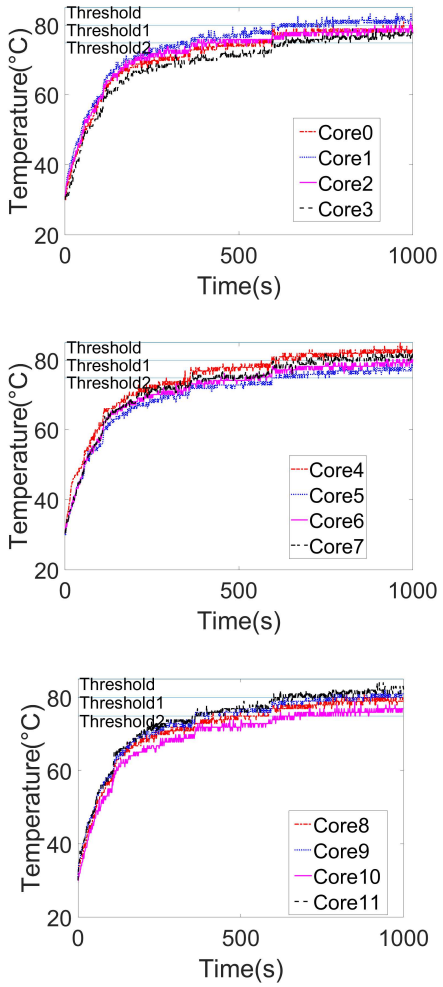

Fig. 4. Temperature Behavior of Each Core under Proposed Technique at 60% Utilization

ACKNOWLEDGMENT

This work was supported by NSF through grants CNS-2301757, CAREER- 2211523, CCF-2118202, and CNS-2211642.

REFERENCES

[1] http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680_v3-30M-Cache-2-50-GHz.
[2] https://docs.toradex.com/100210-colibri-arm-som-pxa270.pdf.
[3] Hakan Aydin and Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. In *ISPDP '03*, 2003.
[4] Ashikahmed Bhuiyan, Zhishan Guo, Abusayeed Saifullah, Nan Guan, and Haoyi Xiong. Energy-efficient real-time scheduling of dag tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(5):1–25, 2018.
[5] Ashikahmed Bhuiyan, Di Liu, Aamir Khan, Abusayeed Saifullah, Nan Guan, and Zhishan Guo. Energy-efficient parallel real-time scheduling on clustered multi-core. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2097–2111, 2020.
[6] Attia et al. Dynamic power management techniques in multi-core architectures: A survey study. *J. Ain Shams Eng*, 2017.
[7] Baati et al. Temperature-aware dvfs-dpm for real-time applications under variable ambient temperature. In *Symp SIES '13*.
[8] Bampis et al. On multiprocessor temperature-aware scheduling problems. *J. of Scheduling*, 2013.
[9] Bashir et al. An online temperature-aware scheduling technique to avoid thermal emergencies in multiprocessor systems. *J Comp Elec Eng*, 2018.
[10] Casini et al. Task splitting and load balancing of dynamic real-time workloads for semi-partitioned edf. *IEEE Trans on Comp*, 2020.
[11] Chandarli et al. Response time analysis for thermal-aware real-time systems under fixed-priority scheduling. In *ISORC '21*.
[12] Chantem et al. Temperature-aware scheduling and assignment for hard real-time applications on mpsocs. *IEEE Trans on VLSI Sys*, 2010.
[13] Fallah et al. Standby and active leakage current control and minimization in cmos vlsi circuits. *IEICE trans elect*, 2005.
[14] Fisher et al. A fully polynomial-time approximation scheme for feasibility analysis in static-priority systems with arbitrary relative deadlines. In *ECRTS '05*, 2005.
[15] Hackenberg et al. An energy efficiency feature survey of the intel haswell processor. In *IPDPS '15*.
[16] Haque et al. On reliability management of energy-aware real-time systems through task replication. *IEEE TPDS*, 2016.
[17] Huang et al. Qt-adaptation engine: Adaptive qos-aware scheduling and governing in thermally constrained mobile devices. *IEEE Trans on Comp-Aided Des of Integ Circt and Sys*, 2019.
[18] Hwang et al. A predictive system shutdown method for energy saving of event-driven computation. *TODAES '00*.
[19] Lee et al. Thermal-aware scheduling for integrated cpus–gpu platforms. *TECS '19*.
[20] Liu et al. Overhead-aware system-level joint energy and performance optimization for streaming applications on multiprocessor systems-on-chip. In *Euromicro Conf Real-Time Sys*, 2008.
[21] Pagani et al. Energy efficient task partitioning based on the single frequency approximation scheme. RTSS '13.
[22] Quan et al. Leakage aware feasibility analysis for temperature-constrained hard real-time periodic tasks. In *ECRTS'09*.
[23] Sanjoy et al. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *RTSS '90*.
[24] Schöne et al. Energy efficiency features of the intel skylake-sp processor and their impact on performance. *arXiv*, 2019.
[25] Sheikh et al. Simultaneous optimization of performance, energy and temperature for dag scheduling in multi-core processors. In *IGCC '12*.
[26] Sheikh et al. Fast algorithms for thermal constrained performance optimization in dag scheduling on multi-core processors. In *Int Green Comp Conf and Workshp*, 2011.
[27] Skadron et al. Temperature-aware microarchitecture. *ACM SIGARCH*, 2003.
[28] Srinivasan et al. Exploiting structural duplication for lifetime reliability enhancement. In *ISCA '05*.
[29] Wang et al. A simple thermal model for multi-core processors and its application to slack allocation. In *IPDPS '10*.
[30] Wang et al. Adaptive routing algorithms for lifetime reliability optimization in network-on-chip. *IEEE Trans Comp*, 2015.
[31] Xu et al. Energy-efficient policies for embedded clusters. *ACM SIGPLAN Not.*, 2005.
[32] Zanini et al. A control theory approach for thermal balancing of mpsoc. In *Proc Asia and South Pacific Desgn Automtn*, 2009.
[33] Zhang et al. A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems. *IEEE Trans on Comp-Aid Des Int Circt and Sys*, 2005.
[34] Zhou et al. Balancing lifetime and soft-error reliability to improve system availability. In *ASP-DAC '16*.
[35] Zhou et al. Thermal-aware task scheduling for 3d multicore processors. *IEEE Trans on Para and Dist Sys*, 2009.
[36] Agner Fog. The microarchitecture of intel, amd and via cpus. *opt guide for ass programmrs*, 2011.
[37] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Vol 3B: Sys prog Guid*, 2011.
[38] Zhishan Guo, Ashikahmed Bhuiyan, Di Liu, Aamir Khan, Abusayeed Saifullah, and Nan Guan. Energy-efficient real-time scheduling of dags on clustered multi-core platforms. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 156–168. IEEE, 2019.
[39] Zhishan Guo, Ashikahmed Bhuiyan, Abusayeed Saifullah, Nan Guan, and Haoyi Xiong. Energy-efficient multi-core scheduling for real-time dag tasks. In *29th Euromicro conference on real-time systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
[40] Abusayeed Saifullah, Sezana Fahmida, Venkata P Modekurthy, Nathan Fisher, and Zhishan Guo. Cpu energy-aware parallel real-time scheduling. *Leibniz international proceedings in informatics*, 165, 2020.