Adaptive and Hierarchical Run-time Manager for Energy-Aware Thermal Management of Embedded Systems

Anup Das, Bashir M. Al-Hashimi and Geoff V. Merrett

Modern embedded systems execute applications, which interacts with the operating system and hardware differently depending on type of workload. These cross-layer interactions result in wide variations of chip-wide thermal profile. In this paper, a reinforcement learning-based run-time manager is proposed that guarantees application-specific performance requirements and controls the POSIX thread allocation and voltage/frequency scaling for energy-efficient thermal management. This controls three thermal aspects – peak temperature, average temperature and thermal cycling. Contrary to existing learning-based run-time approaches that optimize energy and temperature individually, the proposed run-time manager is the first approach to combine the two objectives, simultaneously addressing all three thermal aspects. However, determining thread allocation and core frequencies to optimize energy and temperature is an NP-hard problem. This leads to an exponential growth in the learning table (significant memory overhead) and a corresponding increase in the exploration time to learn the most appropriate thread allocation and core frequency for a particular application workload. To confine the learning space and to minimize the learning cost, the proposed run-time manager is implemented in a two-stage hierarchy: a heuristic-based thread allocation at a longer time interval to improve thermal cycling, followed by a learning-based hardware frequency selection at a much finer interval to improve average temperature, peak temperature and energy consumption. This enables finer control on temperature in an energy-efficient manner, while simultaneously addressing scalability, which is a crucial aspect for multi-/many-core embedded systems. The proposed hierarchical run-time manager is implemented for Linux running on nVidia's Tegra SoC, featuring four ARM Cortex-A15 cores. Experiments conducted with a range of embedded and cpu intensive applications demonstrate that the proposed run-time manager not only reduces energy consumption by an average 15% with respect to Linux, but also improves all the thermal aspects – average temperature by 14°C, peak temperature by 16°C and thermal cycling by 54%.

Categories and Subject Descriptors: D.4.7 [**Operating System**]: Organization and Design

General Terms: Run-time Manager, Reinforcement learning, Thermal management, Energy consumption

Additional Key Words and Phrases: Embedded systems, Linux operating system

1. INTRODUCTION

To accommodate the growing demand for performance, modern embedded systems integrate multiple general purpose cores on the same system-on-chip (SoC). Examples of these SoCs are Texas Instrument's OMAP, nVidia's Tegra and Samsung's Exynos. A major challenge of these multicore SoCs is decreasing lifetime reliability, threatened by high power densities and hence elevated operating temperatures. This leads to an acceleration of device wear-out, manifesting as hard logic and intermittent timing faults. Additionally, elevated temperature increases leakage current exponentially, resulting in a higher energy consumption, which is critical especially for battery-operated embedded systems. Energy-aware thermal management is therefore emerging as a pri-

This work is supported by the Engineering and Physical Sciences Research Council (EPSRC) Programme Grant, EP/K034448/1. See www.prime-project.org for more information about the PRiME programme.
Author's addresses: A. Das, B. M. Al-Hashimi and G. V. Merrett are with the School of Electronics and Computer Science, University of Southampton, United Kingdom SO17 1BJ.

DOI:http://dx.doi.org/10.1145/0000000.0000000

mary design objective for embedded SoCs. At the other end of the multiprocessor spectrum, the growing prevalence of multi-/many-core systems has resulted in widespread use of parallel programming models such as OpenMP [Dagum and Menon 1998], MapReduce [Dean and Ghemawat 2008] and POSIX [pos 1994]. Threads of these programming models have been used extensively in literature to design parallel applications: better utilizing the hardware resources and possibly shortening the execution time. Determining the appropriate thread allocation on the processing cores, to satisfy a given objective (performance, energy, or temperature), is an NP-hard problem. This is typically managed by the operating system or the software running on the embedded platform. However as we demonstrate, operating system controlled thread affinity results in a poor thermal behavior.

Real-time applications such as video play-back and image processing are characterized by dynamic workloads and thermal profiles that are difficult to foresee at compile-time. As such, static compile-time policies [Rai et al. 2011; Schor et al. 2013; Das et al. 2015a] (with limited knowledge of application-specific variations) are often outperformed, even by naive run-time managers, both in terms of thermal overhead and energy consumption – the two key design aspects of modern systems. This has motivated researchers in recent years to investigate run-time approaches, thriving the development of intelligent run-time systems for energy and thermal management [Cochran et al. 2011b; Javaid et al. 2011; Juan et al. 2013; Ye and Xu 2014; Srinivasan et al. 2004; Sharifi et al. 2013; Shi et al. 2013; Faruque et al. 2010; Ge and Qiu 2011; Coskun et al. 2009a; Mercati et al. 2013; 2014; Das et al. 2014; Coskun et al. 2009b; Ebi et al. 2009; Ebi et al. 2011; Shen et al. 2012].

One of the emerging thermal concerns for embedded systems is thermal cycling i.e., the wear-out induced by thermal stress due to a mismatched coefficient of thermal expansion of the adjacent material layers. Most of the earlier works on run-time thermal optimization have focused on minimizing the average and peak temperature, leading to reduction in temperature-related wear-outs, such as electromigration (EM), negative bias temperature instability (NBTI) and time-dependent dielectric breakdown (TDDB) [Srinivasan et al. 2004]. As we demonstrate in this work, the voltage-frequency control proposed in these approaches does not affect thermal cycling as significantly as it does to minimize the average and peak temperature, limiting the adaptability of these existing techniques to thermal cycling optimization.

Contributions: We propose a reinforcement-learning-based run-time manager for energy-efficient thermal management of embedded systems, simultaneously addressing the three thermal aspects – peak temperature, average temperature and thermal cycling. To provide a scalable solution to the NP-hard problem of finding the appropriate thread allocation and frequency selection for an application workload, and to minimize the learning overhead, the run-time manager is implemented as a two-stage hierarchy. The objective is to control thermal cycling using thread allocation (*thread affinity* – a Pthread feature) from the first stage (upper hierarchy). The chip-wide DVFS feature of the processing cores is explored at the next stage (lower hierarchy), corresponding to a given thread allocation to enable finer control over average temperature, peak temperature and energy consumption. Following are our key contributions:

— a low overhead hierarchical run-time manager for energy-aware thermal management, validated extensively with cpu intensive and embedded applications on an ARM-based embedded system;
— reinforcement learning-based adaptation to application specific workload and thermal variations; and
— an approach to optimize thermal cycling, simultaneously with average temperature, peak temperature and energy consumption.

The remainder of this paper is organized as follows. An overview of the related works is provided in Section 2. The problem formulation is discussed in Section 3 motivating the choice of reinforcement learning as the solution. The hierarchical run-time manager is discussed in Section 4 along with an overview of the fundamentals of reinforcement learning. Evaluation of the proposed run-time manager is presented in Section 5 and the paper is concluded in Section 6.

2. RELATED WORKS

Dynamic thermal management (DTM) through voltage and frequency control has received significant attention in recent years. A thermal prediction model is developed in [Sharifi et al. 2013] based on offline thermal and power characterization of an application. A dynamic thermal management approach is proposed in [Shi et al. 2013] based on a lumped thermal control model. Using this, an approach is proposed to optimize the performance of an application with soft thermal constraints. This approach also suffers from similar limitations as discussed before. A distributed agent-based approach is proposed in [Faruque et al. 2010] that uses fast context-aware task migration to minimize peak temperature-related hotspots. This approach does not minimize average temperature and thermal cycling. A reinforcement learning-based thermal management approach is proposed in [Ge and Qiu 2011] that uses feedback from hardware thermal sensors to adjust the voltage and frequency of processing cores. Although this approach is closest to the one proposed in this work, the approach does not optimize thermal cycling, nor does it adapt to workload-specific variations. Another learning-based approach is proposed in [Coskun et al. 2009a] to manage temperature of multiprocessor systems, and selects between a set of expert policies depending on workload characteristics. HotSpot is used for temperature modeling based on thermal characteristics of UltraSPARC. However, HotSpot has a known limitation on accuracy and simulation time [Das et al. 2015a], making this approach difficult to use for real-time applications. A control-theoretic approach is proposed in [Mercati et al. 2013] to optimize the lifetime reliability of a multiprocessor system. Task scheduling decisions are controlled at longer intervals and the voltage/frequency scaling is performed at a shorter interval. This approach is extended in [Mercati et al. 2014] as a governor for the Android Operating System. Another control approach is proposed in [Sironi et al. 2013] to manage the temperature of applications running on multiprocessor systems. A thermal-safe power budgeting is proposed in [Pagani et al. 2014] for dynamic thermal management of many-core system. A fast even-driven approach is proposed in [Cui and Maskell 2012] to estimate the temperature of a multiprocessor system. Based on this a thermal aware scheduling approach is proposed to reduce the temperature of the system at run-time. Apart from these works, there are other studies to reduce the power consumption of a multicore system by scaling the hardware frequency dynamically [Dhiman and Rosing 2009; Javaid et al. 2011; Ye and Xu 2014; Khan and Rinner 2014]. However, as shown in [Faruque et al. 2010], these approaches cannot guarantee to minimize a system's thermal overhead effectively for all applications. A cross-layer thermal optimization technique is proposed in [Das et al. 2014] to manage temperature-related emergencies. Although these studies have shown improvement in thermal profile leading to extended lifetime reliability using scaled voltage and frequency, thermal cycling and energy consumption are not jointly addressed.

The power consumption of a system scales linearly with frequency and quadratically with voltage. Therefore, the thermal management approaches are presumed to minimize energy consumption as well, which comes as a secondary benefit with thermal improvements. Two of the most widely accepted system-level design techniques for power optimization are dynamic voltage and frequency scaling (DVFS) [Simunic et al. 2001] and dynamic power management (DPM) [Benini et al. 1998]. In DVFS, the volt-

age and frequency are scaled down dynamically to reduce both the active and leakage power consumption, whereas in DPM, the processing cores are shut down (or put into sleep mode) to reduce leakage power. A continuous frequency adjustment technique is proposed in [Jung and Pedram 2008] based on predicted workload, which is formulated as an initial value problem (IVP). The technique in [Dhiman and Rosing 2007; Shen et al. 2012; Shen et al. 2013; Ye and Xu 2014] uses online learning to select the most appropriate frequency for the processing cores based on the workload characteristic of a given application. A workload characteristic aware thread scheduler is proposed in [Dhiman et al. 2010] based on dynamic workload characterization. In [Jung and Pedram 2010], a supervised learning in the form of a Bayesian classifier for energy management is proposed. This framework learns to predict the system performance from the occupancy state of the global service queue. The predicted performance is then used to select the frequency from a pre-computed policy table. In [Cochran et al. 2011b; 2011a], a multinominal logistic regression classifier is built using a large volume of performance counters by offline workload characterization. This classifier is queried at run-time for a given application to predict the workload, and select the frequency and thread packing such that performance is maximized under a power cap.

However, the assumption of thermal improvement through power control levers has recently been challenged in [Coskun et al. 2009b], directing subsequent studies to focus on exploring the trade-off between chip temperature and energy consumption. A distributed agent-based power and thermal optimization technique is presented in [Ebi et al. 2009] based on control theoretic principles. Reinforcement learning is proposed in [Ebi et al. 2011] as an alternative. Another reinforcement learning-based approach is proposed for multi-core systems in [Shen et al. 2012] with DVFS to control the average temperature and energy consumption. None of these techniques optimize energy and temperature considering all the three thermal aspects (peak temperature, average temperature and thermal cycling) simultaneously.

3. PROBLEM FORMULATION

There are three dependencies that are relevant to this work – the sub-threshold leakage power of a system and its lifetime reliability are both dependent on temperature; the dynamic power consumption is dependent on the voltage and frequency of operation; and the temperature is dependent on the voltage, frequency and also on the application workload. In this section, we provide an overview of these dependencies and formulate the objective we optimize in this work.

3.1. Power Consumption

As discussed in [He et al. 2004], a typical processor can be considered as being in one of the three power states:

— Standby Mode: in this mode, a processor is idle. The total power consumption is comprised of the leakage component (P_s) only;
— Active Mode: in this mode, a processor is active and executes instructions; The total power is comprised of leakage (P_s) and dynamic (P_d) components; and
— Inactive Mode: in this mode, a processor is usually power gated. The total power is comprised of the reduced leakage component (P_r).

The dynamic power of a processor is directly proportional to the frequency (f) of operation and quadratically proportional to the voltage (V), i.e.,

$$P_d \propto f \times V^2 \tag{1}$$

Table I. MTTF considering different wear-out mechanisms.

Wear-out	MTTF	Comments
Electromigration (EM)	$\frac{A_{EM}}{J^n}\exp\left(\frac{E_{a_{EM}}}{KT}\right)$	A_{EM} is a material-dependent constant, J is the current density, n is empirically determined constant with a typical value of 2 for stress related failures, $E_{a_{EM}}$ is the activation energy of electromigration, K is the Boltzman's constant, and T is the temperature.
Negative Bias Temperature Instability (NBTI)	$\frac{A_{NBTI}}{(V_{GS})^{\gamma}}\exp\left(\frac{E_{a_{NBTI}}}{KT}\right)$	A_{NBTI} is a constant dependent on the fabrication process, γ is the voltage acceleration factor and $E_{a_{NBTI}}$ is the activation energy.
Hot Carrier Injection (HCI)	$A_{HCI}\exp\left(\frac{\theta}{V_{DS}}\right)$	A_{HCI} and θ are empirically determined constants and V_{DS} is the drain to source voltage.
Time Dependent Dielectric Breakdown (TDDB)	$A_{TDDB}\cdot A_G\cdot\left(\frac{1}{V_{GS}}\right)^{\alpha-\beta T}\exp\left(\frac{X}{T}+\frac{Y}{T^2}\right)$	V_{GS} is the gate voltage, T is the temperature, $\alpha,\ \beta,\ X$ and Y are fitting parameters, A_G is the surface area of the gate oxide and A_{TDDB} is an empirically determined constant.
Stress Migration (SM)	$A_{SM}\lvert T_0-T\rvert^{-n}\exp\left(\frac{E_{a_{SM}}}{KT}\right)$	A_{SM} is a material dependent constant, T_0 is the metal deposition temperature and $E_{a_{SM}}$ is the activation energy.

The standyby power (P_s) is given by [He et al. 2004]

$$P_s = V \times I_{leak} \tag{2}$$

where I_{leak} is the leakage current. Of the different leakage components, the sub-threshold leakage current is the dominant one, and is given by

$$I_{sub} = V \times I_o \times \left[AT^2 e^{\frac{\alpha V+\beta}{T}+Be^{\gamma V+\delta}}\right] \tag{3}$$

where T is the temperature, I_o is the leakage current at the reference temperature, and $A, B, \alpha, \beta, \gamma$ and δ are the technology dependent constants. Clearly, the sub-threshold leakage current is super-linearly dependent on the temperature. This work performs thermal management by down scaling the frequency and controlling the thread-to-core affinity. Temperature reduction results in reduction of leakage power and frequency scaling reduces the dynamic power.

3.2. Energy Consumption

The energy consumption of a system is given by the area under the power curve i.e.,

$$E = \int_0^{\tau} P(t)dt = \int_0^{\tau} (P_d + P_s)dt \tag{4}$$

where τ is the time duration of an application.

3.3. Lifetime Reliability

The lifetime reliability is defined as the long term reliability of a circuit and is measured in terms of the mean time to permanent failure (MTTF). The MTTF due to different wear-out mechanisms are highlighted in Table I [Srinivasan et al. 2004]. As can be seen, average and peak temperature play an important role in determining the MTTF. One of the emerging lifetime concern with scaled transistor geometry is thermal cycling, which is defined as wear-out caused by thermal stress due to a mismatched coefficient of thermal expansion of the adjacent material layers. Thermal cycling related MTTF is computed by.

1. Calculating the thermal cycles from a thermal profile using Downing's simple rainbow counting algorithm [Downing and Socie 1982].

2. Calculating, from each thermal cycle, the number of cycles to failure using Coffin-Manson's rule [Manson 1972; Coffin Jr 1973].

$$N_{TC}(i) = A_{TC}\left(\delta T_i - T_{Th}\right)^{-b} e^{\frac{Ea_{TC}}{KT_{max}(i)}} \tag{5}$$

where $N_{TC}(i)$ is the number of cycles to failure due to i^{th} thermal cycle, A_{TC} is an empirically determined constant, δT_i is the amplitude of the i^{th} thermal cycle, T_{Th} is the temperature at which elastic deformation begins, b is the Coffin-Manson exponent constant, $E_{a_{TC}}$ is the activation energy of thermal cycling and $T_{max}(i)$ is the maximum temperature in the i^{th} thermal cycle.

3. Calculating the MTTF using Miner's rule [Chaboche and Lesne 1988].

$$MTTF = \frac{N_{TC}\sum_{i=1}^{m} t_i}{m} \tag{6}$$

where t_i is the time for the i^{th} thermal cycle, m is the number of thermal cycles obtained in step 1 and N_{TC} is the effective cycles to failure determined using

$$N_{TC} = \frac{m}{\sum_{i=1}^{m}\frac{1}{N_{TC}(i)}} \tag{7}$$

Combining Equations 5-7,

$$MTTF = \frac{A_{TC}\sum_{i=1}^{m} t_i}{\textit{Thermal Stress}} \tag{8}$$

where *Thermal Stress* is an indication of the stress experienced due to the thermal cycling. This is obtained using the following equation.

$$\textit{Thermal Stress} = \sum_{i=1}^{m}(\delta T_i - T_{Th})^b \times e^{\frac{-Ea}{KT_{max}(i)}} \tag{9}$$

3.4. Optimization Objective

The objective of this paper is to perform energy-aware thermal management. It is important to note that the leakage power is dependent on the average and peak temperature, while the lifetime reliability is dependent on average temperature and thermal cycling. In order to optimize both the leakage power and lifetime reliability of a system, it is essential to optimize average temperature, peak temperature and thermal cycling. Additionally, by scaling down the voltage and frequency to control temperature (as detailed in Section 4), dynamic power is also minimized, achieving an overall reduction of energy consumption (Equation 4).

The primary goal of this work is therefore to optimize the three thermal parameters, which are combined into a single objective, thermal overhead (T_O). This is computed as follows. Let $T_1^i, T_2^i, \cdots, T_{N_l}^i$ denote the N_l thermal sensor readings obtained in the time interval t_i to t_{i+1}. The thermal overhead in this interval is given by

$$\begin{aligned} T_O(t_i \rightarrow t_{i+1}) \;=\; & \omega_1 \times \text{mean}(T_1^i, T_2^i, \cdots, T_{N_l}^i) + \\ & \omega_2 \times \max(T_1^i, T_2^i, \cdots, T_{N_l}^i) + \\ & \omega_3 \times \text{ThermalCycle}(T_1^i, T_2^i, \cdots, T_{N_l}^i) \end{aligned} \tag{10}$$

where $ThermalCycle$ computes the thermal cycle related damage obtained from the temperature time series, and ω_1, ω_2 and ω_3 are the weights assigned to the three thermal parameters. All results in this work are generated with equal weights assigned to these thermal parameters, i.e. $\omega_1 = \omega_2 = \omega_3 = \frac{1}{3}$. However, the proposed approach is orthogonal to the choice of these weights.

3.5. Motivation for Machine Learning

Thermal overhead of an embedded system during the time interval $t_i \rightarrow t_{i+1}$ can also be represented as

$$T_O(t_i \rightarrow t_{i+1}) = f(AppComp) + g(ArchComp) + h(EnvComp) \quad (11)$$

where $AppComp$ are the application-specific factors that contribute to temperature. As established in [Sharifi et al. 2013; Shi et al. 2013; Faruque et al. 2010; Ge and Qiu 2011; Coskun et al. 2009a], the following factors contribute to temperature

— application workload (instruction types present in the application);
— application performance constraints.

The relationship between temperature and these components is not known with certainty. Although application characterization technique proposed in [Sharifi et al. 2013] can be used to determine the function $f(AppComp)$, the accuracy of this approach is highly sensitive to the training set used for characterization.

$ArchComp$ are the architectural components that contribute to temperature. As discussed in [Skadron et al. 2004], the following architectural components impact temperature significantly.

— processor temperature as a function of power states;
— power consumed per instruction;
— power consumed in memory access;
— floorplan of the system-on-chip; and
— presence of heat-sink, fan, etc.

There are also environmental factors (such as ambient temperature) indicated in Equation 11 as $EnvComp$, which influences the temperature.

As shown in [Das et al. 2014], temperature of an embedded system can be controlled significantly by controlling the processor power states (i.e., their voltage and frequency) and the application thread allocation (that limits context switching). However, the amount of thermal control achieved using these control levers is dependent on the application, its cross-layer interaction with the system software and the hardware, and also on the working environment. Q-learning (a variant of reinforcement learning [Barto 1998]) provides a generic framework to identify these interactions and to delegate appropriate control in order to optimize the long-term thermal overhead of a system.

3.6. Choice of programming Model

Several parallel programming models have been proposed in literature. Examples include OpenMP [Dagum and Menon 1998], MapReduce [Dean and Ghemawat 2008] and POSIX [pos 1994]. These programming models implement independent control within a regular process that share global data but maintain their own private stack, local variables and program counters. In this paper we adopted the POSIX programming model. This choice is partially guided by the fact that the native GNU compiler supports the POSIX standard by default, while OpenMP or MapReduce requires programming model specific compiler support. However, the underlying approach proposed here is generic and can be used with other programming models. The thread extension to POSIX (referred to as Pthreads) describes the interface for lightweight threads on a shared memory architecture and have a smaller context size than application processes. Pthreads have been used extensively in literature to design parallel applications: better utilizing the hardware resources and possibly shortening the execution time. An application's POSIX threads are scheduled using operating system's `thread affinity` API.

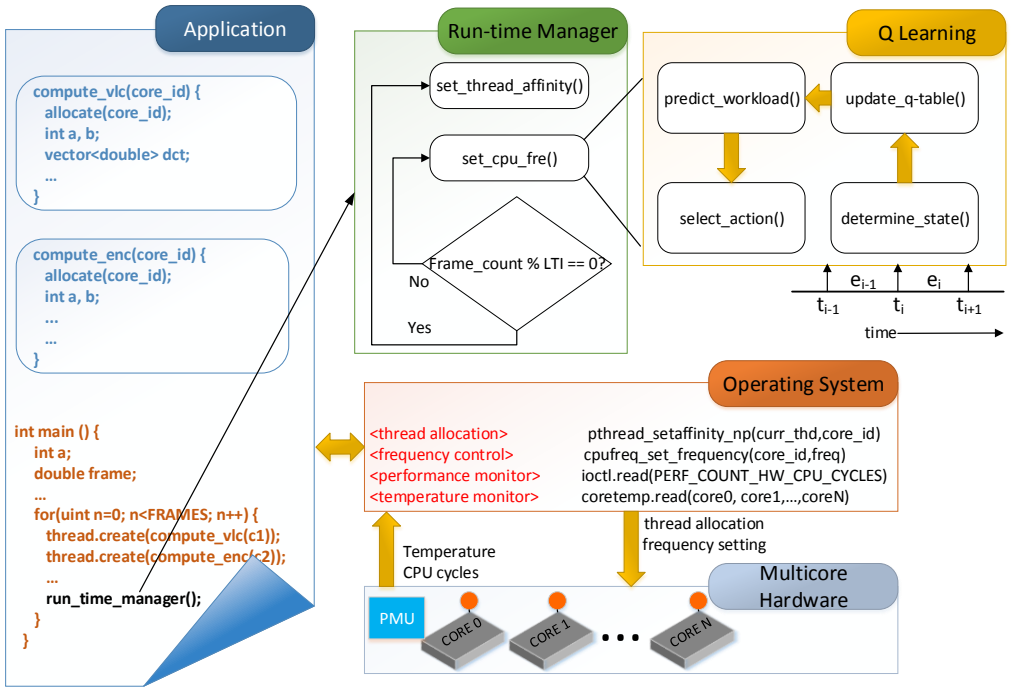

Fig. 1. Proposed hierarchical run-time manager

4. PROPOSED RUN-TIME MANAGER

As indicated in the IEEE POSIX standard [pos 1994], Pthread implementation can be carried out as a Kernel implementation (where functionalities are implemented as part of the operating system), as a library implementation (where functionalities are implemented as part of the user program), or a mix of both. Figure 1 describes the proposed hierarchical run-time manager implemented as a library implementation. The application code (typically written in a high level language such as C/C++/Java) implements the `run_time_manager()` function which is indicated by the solid arrow. The application interfaces with the operating system (OS) using drivers and application programming interfaces (APIs) as shown in the box titled *Operating System*. These drivers and APIs monitor/control the hardware, which consist of processing cores, thermal sensors and a performance monitoring unit (PMU). The monitoring parameters and control levers are indicated against the arrows between the *Operating System* and *Multicore Hardware*. Following are the details of the different components of the proposed framework, starting with an overview of the Q-learning algorithm.

4.1. Fundamentals of Q-learning

In a standard Q-learning framework, a learning agent (the run-time manager, RTM) repeatedly observes the current state of the system, and selects an action. The selected action changes the system state, which is used to determine the immediate numeric payoff. Positive payoffs are termed as profits and negative payoffs are termed as punishments. The RTM must learn to select actions in order to maximize the long-term sum or average of the future payoffs.

Initially, the RTM does not know what effects its actions have on the state of the system, nor what immediate payoffs its actions will produce. Rather, it tries out various actions in different states. This phase of the algorithm is known as the *exploration* phase. The payoffs received in this phase (also known as RTM's experience) are stored in a table (termed Q-table). To make this framework sensible, the RTM needs to further evaluate good decisions by selecting them and observing the state of the system. This phase of the algorithm is known as the *exploration-exploitation* phase. In this phase, the RTM uses a fraction (typically less than 100%) of the payoffs as experience to update the Q-table. Finally, at the end of this phase, the RTM is said to have fully learnt a system's thermal behavior. This phase is known as the *exploitation* phase. In this phase, the RTM always selects the best action (i.e., the action corresponding to the highest payoff) for a system state. The mapping of different components of Q-learning in the run-time management framework are discussed next.

4.1.1. Decision Epochs. The RTM works at the system time ticks (indicated in Figure 1). The interval between two consecutive ticks is referred to as the *decision epoch* (indicated by the letter e in the figure). The decision epoch e_i is the time interval between ticks t_i and t_{i+1}. The learning algorithms works as follows. At time instant t_i, the RTM performs the following steps in order

— computes payoff for the time interval $t_{i-1} \rightarrow t_i$;
— updates the Q-table entry corresponding to the state and action at time t_{i-1};
— predicts the system state for the interval $t_i \rightarrow t_{i+1}$;
— selects the action for the interval $t_i \rightarrow t_{i+1}$ based on the predicted state.

It is to be noted that, in this work we proactively manage the temperature; therefore, the next system state is predicted and appropriate actions are enforced, before the system reaching the state. In this way, the approach prevents thermal emergencies from occurring (proactive), rather than reacting when such emergencies occur (reactive).

4.1.2. Payoffs. As discussed before, payoff defines the optimization objective, which in our context is the thermal overhead (average temperature, peak temperature and thermal cycling, combined using Equation 10). Since we are concerned with constrained optimization problem, the performance constraint needs to be incorporated in the payoff, which is given by the following equation

$$R(t_i) = \begin{cases} w_t \times [T_O^{max} - T_O(t_{i-1} \rightarrow t_i)] & \text{if } L_i \geq L_c \\ w_s \times (L_i - L_c) & \text{otherwise} \end{cases} \quad (12)$$

where $R(t_i)$ is the payoff calculated at time instance t_i, L_i is the application performance during the interval t_{i-1} to t_i, $T_O(t_{i-1} \rightarrow t_i)$ is the thermal overhead of the system in this interval (given by Equation 10), L_c is the performance constraint, T_O^{max} is the thermal overhead at the highest voltage and frequency, and w_t and w_s are respectively the weights for the temperature and performance. These weights are calculated as $\frac{w_t}{w_s} = \frac{L_i^{max}}{T_O^{max}}$, where L_i^{max} is the performance obtained with the highest voltage and frequency applied on the processing cores[1]. The performance of the system is measured as the inverse of the timing requirement. The equation is interpreted as follows: if the performance obtained in the interval of interest is greater than the performance constraint, the thermal overhead is used to compute the payoff. On the other hand, if there is performance violation, the negative of the performance slack is used as the payoff.

[1]It is important to note that in the proposed approach, an application is executed for a few iterations (or frames) to determine these constants, before the reinforcement learning algorithm is initiated.

4.1.3. System State. The state of a system is usually represented using CPU cycle count, obtained by reading the performance monitoring unit. However, for some systems, especially for ARM-based SoCs, direct access to the performance registers are disabled in the user mode of operation. For such system, CPU utilization can be used as an alternative. Thus, the system state s_i at time t_i is

$$s_i = Statistics(t_{i-1} \rightarrow t_i) \tag{13}$$

where $Statistics(t_{i-1} \rightarrow t_i)$ is the performance-related statistics (CPU Cycles or utilization) in the interval $t_{i-1} \rightarrow t_i$. A point to note here is that, the states form the rows of the Q-table. Therefore to limit the size of the Q-table, the range of the performance-related statistics is discretized into N_s levels. The discretized value of the statistics s_i is represented as $\hat{s}_i$.

4.1.4. Action Space. The action space comprises of the thermal control levers for an embedded system. We use processor frequency and thread affinity as actions, similar to [Das et al. 2014]. Let the affinity be represented as a matrix

$$M_A(k) = \begin{pmatrix} c_1^k & c_2^k & \cdots c_{N_t}^k \end{pmatrix} \tag{14}$$

where c_j^k is the core where thread j is allocated in the k^{th} configuration and $c_j^k \in \{c_1, c_2, \cdots, c_{N_c}\}$ with N_c being the number of cores. Most embedded and high performance systems allow chip-wide DVFS i.e., all the processing cores have the same voltage-frequency value. Therefore, the k^{th} action can be represented as

$$a_k = \langle M_A(k) \mid\mid (V_k, f_k) \rangle \tag{15}$$

i.e., an action is composed of the thread affinity matrix and the voltage-frequency values for all the cores. Here, $(V_k, f_k) \in \{(V_1, f_1), (V_2, f_2), \cdots, (V_{N_f}, f_{N_f})\}$ can assume one of the N_f voltage-frequency values supported on the hardware. Usually, the operating system allows scaling the frequency only using the `cpufreq` API. The voltage is scaled accordingly. Therefore, Equation 15 can be simplified to

$$a_k = \langle M_A(k) \mid\mid f_k \rangle \tag{16}$$

The actions form the columns of the Q-table. The total number of actions of the Q-learning is given by

$$N_a = N_f \times N_c^{N_t} \tag{17}$$

Clearly, the number of actions grows exponentially with an increase in the number of threads and cores. In Section 4.2, we discuss the algorithmic modifications to limit the number of actions of the Q-learning algorithm.

4.1.5. Q-table Update. The Q-table is a two-dimensional table composed of system states as rows and actions as columns. The Q-table entries for the state and action at time t_{i-1} are updated at time t_i using the payoff as given below.

$$Q(\hat{s}_{i-1}, \hat{a}_{i-1}) = Q(\hat{s}_{i-1}, \hat{a}_{i-1}) + \alpha \times R(t_i) \tag{18}$$

where $\hat{a}_{i-1}$ is the action taken during time $t_{i-1} \rightarrow t_i$ and $\hat{a}_{i-1} \in \{a_1, a_2, \cdots a_{N_a}\}$. The learning rate α ($0 \leq \alpha \leq 1$) denote the fraction of the payoff used as learning experience for updating the Q-table entries. The learning rate can be expressed as

$$\alpha = \begin{cases} 1 & \text{for } 0 \leq N < N_{explore} \\ 2^{(N_{explore} - N)} & \text{for } N_{explore} \leq N < N_{exploit} \\ 0 & \text{for } N \geq N_{exploit} \end{cases} \tag{19}$$

where N is the number of visits, and $N_{explore}, N_{exploit}$ are the constants indicating states of the learning, i.e., exploration, exploration-exploitation and exploitation.

Table II. Memory and learning overhead executed on quad-core embedded system with five frequencies.

# Threads	Learning-table related memory overhead (KB)	Learning time (mins)	Memory overhead with modification (KB)	Learning time with modification (mins)
4	10	15	0.62	0.89
6	51	73	0.93	1.34
8	160	230	1.25	1.79
12	810	1,162	1.87	2.68
16	2,560	3,671	2.50	3.58

These parameters are selected considering the trade-off between Q-table convergence rate and exploration time. For a balance of these metrics, we have used $N_{explore} = 3$ and $N_{explot} = 8$ (similar to [Das et al. 2014]), with N being the number of visits to an entry of the Q-table.

4.1.6. Action Selection. As discussed before, the RTM selects an action at time t_i for controlling the thermal overhead in the time interval $t_i \rightarrow t_{i+1}$ (proactive approach). So, the RTM first needs to predict the state of the system for the interval $t_i \rightarrow t_{i+1}$; subsequently, the RTM selects an action that has previously resulted in the least thermal overhead for that state. To effectively predict the system state, we use the exponential weighted moving average (EWMA) technique (similar to [Coskun et al. 2009a])[2]. In this technique, the predicted system state p_{i+1} during the time interval $t_i \rightarrow t_{i+1}$ is

$$p_{i+1} = \gamma \times s_i + (1 - \gamma) \times p_i \tag{20}$$

where γ is the smoothing factor. The equation is interpreted as follows. The predicted state in the interval $t_i \rightarrow t_{i+1}$ is determined from the predicted state during the interval $t_{i-1} \rightarrow t_i$ (p_i) and also, the actual state during that interval (s_i). It can be intuitively reasoned that the accuracy of the proactive thermal management approach is dependent on the accuracy of the prediction scheme. We provide its evaluation in Section 5.

The action selected in the interval $t_i \rightarrow t_{i+1}$ is

$$a_{i+1} = \texttt{argmax}\ \text{Q-table}(\hat{p}_{i+1}, :) \tag{21}$$

where Q-table$(\hat{p}_{i+1}, :)$ is the Q-table row corresponding to the predicted state p_{i+1} (discretized to $\hat{p}_{i+1}$) and the mathematical operation `argmax` returns the index of the highest argument.

4.2. Algorithmic Modifications

The size of the Q-table is a function of the number of states (N_s) and actions (N_a). This is reported in Table II. As can be seen, with an increase in the number of threads, there is an increase in the size of the learning table, resulting in a corresponding increase of the memory overhead (column 2). This is crucial, especially for embedded systems, where the on-chip memory is limited. The increase in size of the learning table also results in an increase in the time for the learning algorithm to converge to the best thread allocation and frequency setting for a particular application workload. This is reported in column 3 of the table.

To address this, we propose to separate the thread allocation from the frequency selection. Specifically, the thread allocation is changed at long term intervals (*LTIs*) composed of multiple decision epochs using a greedy heuristic. The frequency selection is performed at every decision epoch using the Q-learning. This decision is guided by

[2]It is to be noted that, although EWMA predicts workload to a reasonable accuracy for typical workloads on mobile platforms, our continuing work is to investigate more sophisticated prediction algorithms, such as those involving Kalman filtering and Probabilistic Clustering [Das et al. 2015b].

ALGORITHM 1: $run\_time\_manager()$: Proposed hierarchical run-time manager

Input: N_l temperature samples $\langle T_1^{i-1}, T_2^{i-1}, \cdots, T_{N_l}^{i-1} \rangle$
Output: Thread allocation and frequency selection
1 **if** $count == 0$ **then**
2 | Initialize $M_A(i) = 0$, $MinR = \infty$, $thd = core = 0$, $QTable[N_s][N_a] = 0$ and $t\_enable = 1$;
3 **end**
4 $count$ ++;
5 Calculate Payoff (Equation 12);
6 Update Q-table entry (Equation 18);
7 Predict Next State (Equation 20);
8 Select Action (Equation 21);
9 Map action to core selection and hardware frequency;
10 **if** t_enable && $(count \% LTI == 0)$ **then**
11 | Reset Q-table;
12 | $R = T_O(t_{i-1} \rightarrow t_i)$;
13 | **if** $R < MinR$ **then** $MinR = R$ **else** $M_A(i)[thd] = core_o$;
14 | **if** $core == N_c$ **then** $core = 0$ and $thd++$ **else** $core++$;
15 | $core_o = M_A(i)[thd]$ and $M_A(i)[thd] = core$;
16 | **if** converge **then** $t\_enable = 0$;
17 **end**

experiments on a real system showing that too frequent changes in thread allocation within a program's execution leads to degradation in performance. The hierarchical nature of the proposed run-time manager is shown in Figure 1 in the box titled *Run-time Manager* and as pseudo-code in Algorithm 1. At the start of every LTI, the run-time manager changes thread allocation and resets the learning table; subsequently, the Q-learning algorithm is triggered at every decision epoch to select the frequency.

The Q-learning algorithm (lines 4 - 9) is invoked at every decision epoch. A count is incremented to keep track of the number of times the algorithm is invoked. At every LTI decision epochs, the thread allocation is changed. The parameters needed for the thread allocation are initialized at the first time this algorithm is invoked i.e., corresponding to the count value of 0 (line 2). The essence of the thread allocation algorithm is a greedy heuristic that allocates every thread to every core in order to determine if the thermal overhead is reduced; if so, the thread allocation is retained, else the thread allocation is returned to the previous allocation. This is performed at every LTI at lines 10 - 17 of the algorithm. The thermal overhead is computed (line 12). If this is lower than the minimum overhead obtained thus far ($MinR$), the minimum overhead is updated; else the thread allocation is changed to the previous value (line 13). Thread and core selection are performed in line 14 of the algorithm. Assuming that the thread allocation takes η times to converge, the time complexity of the thread allocation heuristic is $O(\eta \cdot N_t \cdot N_c \cdot LTI)$, where N_t is the number of threads and N_c is the number of cores. The memory overhead and the learning time using the modified algorithm is indicated in columns 4 and 5 of Table II for LTI = 100. The significant reduction of memory overhead and learning time addresses the scalability of the approach for multi-/many-core embedded systems.

4.3. Q-learning Algorithm Demonstration

To demonstrate the working of the Q-learning algorithm, an experiment is conducted on a quad-core platform running Linux. Figure 2 plots the frequency selection of the proposed run-time manager executing an MPEG4 decoding application for 1000 frames of a reference 1080p video. This is shown in the figure using the solid red line. For this experiment, the LTI is selected as 100 frames. As discussed earlier in this

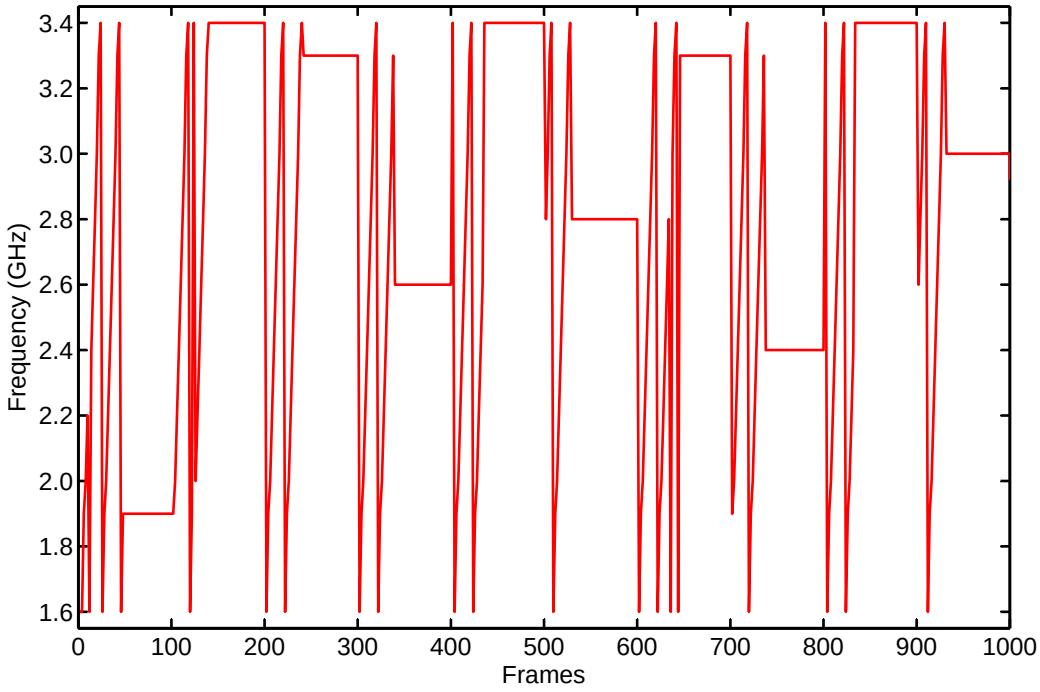

Fig. 2. Frequency selection of the run-time manager.

section, the thread affinity is changed at the start of this interval followed by a reset of the learning table. This is confirmed from the frequency selection results obtained using the proposed run-time manager. The first 40 frames (on average) of every LTI are spent learning the most effective voltage-frequency setting corresponding to the frame workload and the selected thread affinity. For the remaining 60 frames, the algorithm stays in the exploitation phase, thereby always selecting the best frequency for the corresponding workload. This trend is the same for all applications considered in this work. Two points to be noted from these results – the selection of the long-term interval (100 frames) is based on energy-reliability trade-off as discussed in Section 5; and although the learning table is reset at the beginning of every LTI, transfer of knowledge from one LTI to another is left as future work.

5. RESULTS

Experiments are conducted on nvidia's Tegra K1 SoC (Jetson) with quad-core ARM Cortex-A15 running Ubuntu Linux kernel 3.10.24 and supporting chip-wide DVFS. A set of multi-threaded benchmarks are considered from MiBench [Guthaus et al. 2001], PARSEC and the SPLASH2 [Bienia et al. 2008] suites. Each application is transformed to a periodic structure, where the application is executed for several iterations; each iteration is accompanied by a deadline, which serves as the performance requirement. At each iteration multiple threads are spawned, with each thread performing some task on the input data. These iterations are referred to as frames throughout the remainder of this work. It is important to note that video applications (ffmpeg, openCV.sobel etc.) automatically align to this general structure with a frame representing a video picture or a group of pictures (GoPs).

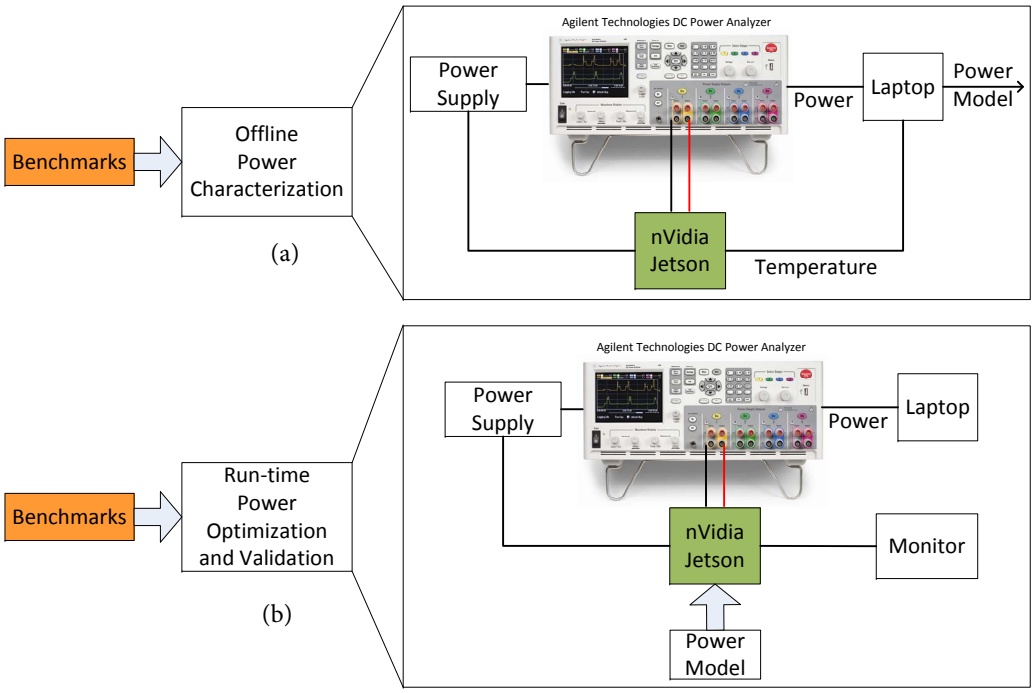

Fig. 3. Framework for (a) generating power model and (b) using it for power management.

The Tegra K1 SoC supports 22 frequency levels and a thermal sensor for temperature measurement. Figure 3 shows the framework for building a power model and using it as part of the reinforcement learning algorithm for power and thermal management. In Figure 3(a), hardware performance counters are recorded for a set of benchmarks from the three benchmark suites. These counter readings are used together with voltage, frequency and temperature to correlate (using a nonlinear fit) to the actual power consumption recorded from a DC power analyzer from Agilent Technologies (N6705B). Readers can refer to [Walker et al. 2015] for details on the validation and accuracy of this power modeling approach. The nonlinear power model is then used at run-time as part of the run-time management for power and thermal optimization. Benchmarks used for building the power model are different to those used for validating the proposed reinforcement learning-based approach. The data logged using this framework is used to compute the energy consumption (Equation 4).

5.1. Workload Prediction Results

The smoothing factor γ (Equation 20) defines the relative importance of the predicted workload as compared to the actual workload of the prior frames. Figure 4 plots the effect of varying the smoothing factor γ on the number of deadline misses (expressed as a percentage of the total frames) and the power consumption (in watts) for the ffmpeg application used to play a 1080p video. As γ increases, the number of workload miss-predictions (over/under) decreases until $\gamma = 0.6$-0.7, beyond which the miss-prediction again increases. Workload under-predictions (actual workload higher than the pre-

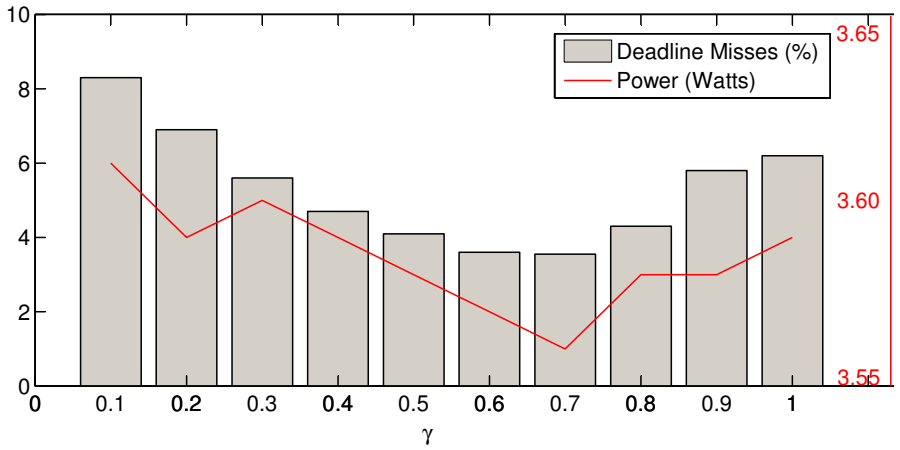

Fig. 4. Effect of workload under-prediction.

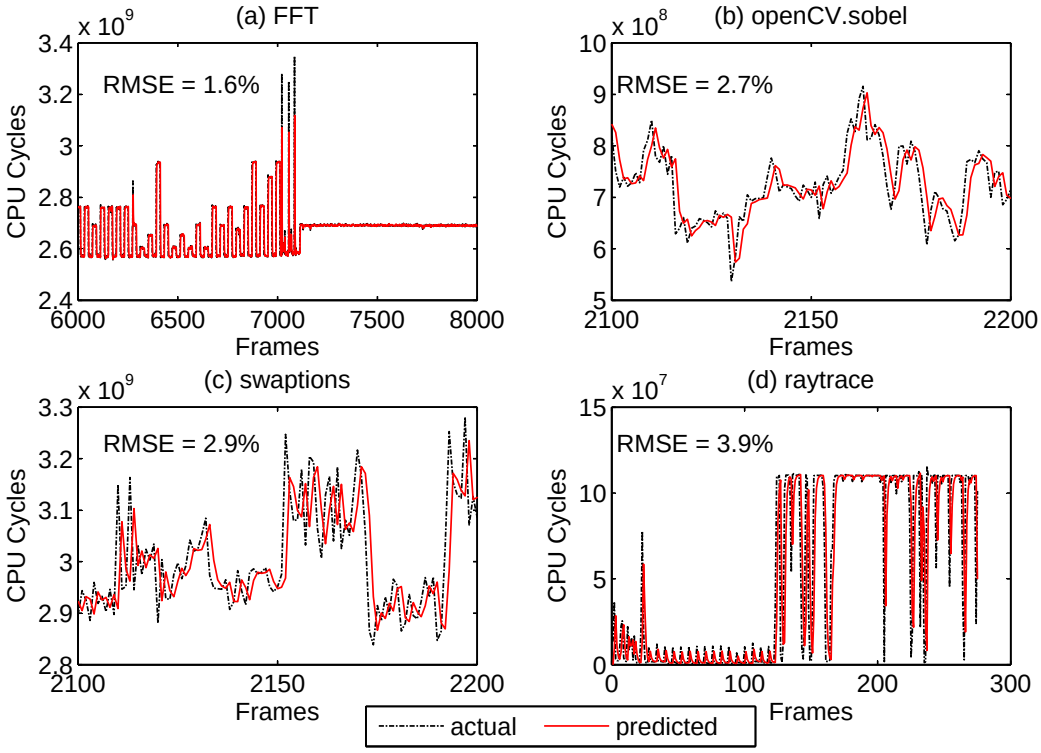

Fig. 5. Workload prediction using EWMA.

dicted workload) result in frames missing the deadline[3]. It is to be noted that in most video decoders, frames missing deadline are usually dropped. This results in a glitch

[3]Typically, the display subsystem has a buffer of one frame. Thus, the deadline for a frame is equal to 42 ms for a 24 fps video.

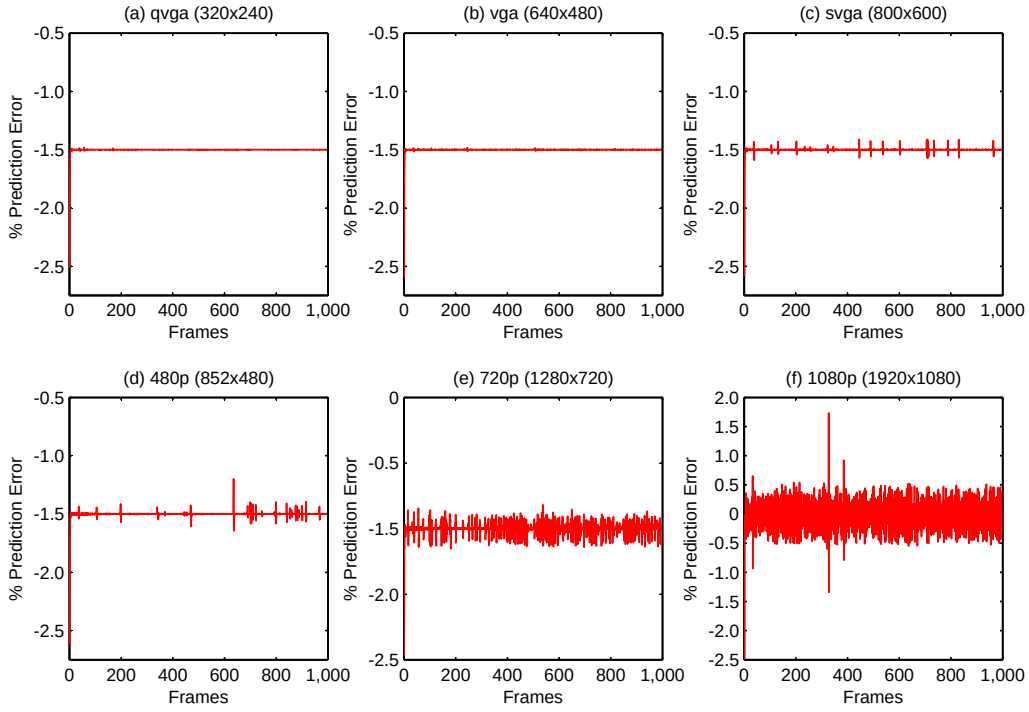

Fig. 6. Prediction error using EWMA for six resolutions of the same video.

in the output video and therefore, degrades quality of user experience. Similarly, workload over-predictions (actual workload lower than the predicted workload) results in higher power consumption. As seen from the figure, a γ value of 0.6-0.7 yields the best result in terms of the number of deadline misses and power consumption. A similar trend is observed for all other applications. We have therefore selected $\gamma = 0.6$ for all our experiments.

Figure 5 plots the difference between the actual and the predicted workload for four different applications. The applications FFT is from MiBench benchmark suite; *sobel* is an OpenCV application commonly used in mobile domain for edge detection; and swaptions and raytrace are from the PARSEC benchmark suite. In this figure, the actual workload is plotted in black dashed line and the predicted one in red solid line for the four applications. The root mean square (RMS) error in workload prediction is also reported (as percentage) for each of these applications.

The workload prediction error is dependent on the application and the input data used for experiment; the results shown above are for one input set of data from multiple data sets considered. For video decoding applications (such as an *MPEG4 decoder*), the prediction error is dependent on the resolution of the video being decoded. To establish this, Figure 6 plots the % prediction error for six different resolutions of the same video decoded using the MPEG4 codec of the *ffmpeg* application, which is the most commonly used video play-back application for embedded systems. Results are reported for the first 1000 frames for three standard definition resolutions (identified in the figure as *qvga* (320×240), *vga* (640×480) and *svga* (800×600)), and three high definition resolutions (identified in the figure as *480p* (852×480), *720p* (1280×720) and *1080p* (1920×1080)). As can be seen from the figures, the prediction error increases

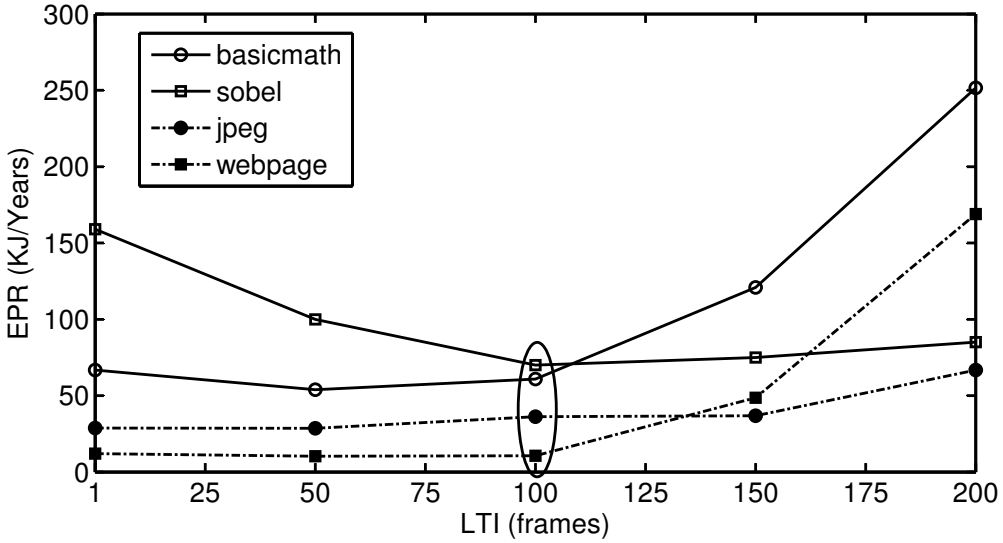

Fig. 7. Selection of long term interval.

with an increase in the resolution. This is expected because, with an increase in the resolution, the workload difference between consecutive frames increases. In future, other sophisticated prediction algorithms will be investigated.

5.2. Selection of Long Term Interval

To illustrate the impact of changing the LTI on energy and reliability, a joint metric – energy per unit reliability (EPR) is introduced ($EPR = Energy/Reliability$). Figure 7 plots the EPR obtained using the proposed run-time manager for the same four applications, with LTI varying from 1 to 200. When LTI is small, there is frequent switching of application threads. This increases the timing overhead and degrades performance. To meet the performance requirement, the proposed run-time manager raises the hardware frequency and therefore, the energy consumption is higher for lower LTI. On the other hand, lower LTI results in finer control on thermal cycling and therefore, the reliability values are also higher. When the LTI is increased, the energy overhead decreases and so does the reliability. When the two objectives are combined (as EPR), the value of EPR first decreases and then starts increasing. This is because, initially the decrease in energy dominates over the decrease in reliability causing a fall in ERP. However, beyond a point, the decrease in reliability starts dominating over that in energy consumption causing the overall EPR to increase. This is the general trend observed for most applications. The value of LTI corresponding to the minimum EPR is to be selected. We have used LTI = 100 as this results in best energy-reliability trade-off for most applications, including those not shown explicitly in the figure.

5.3. Energy Efficient Thermal Management: The Sobel Filter Case Study

Figure 8 plots the performance and power results using the proposed run-time manager in comparison with three popular Linux governors – ondemand, powersave and performance, and with that obtained using the learning-based technique of [Juan et al. 2013], which uses DVFS only. These results are obtained by executing *sobel* filtering on a 1080p video. The sobel filtering application is selected because, this application

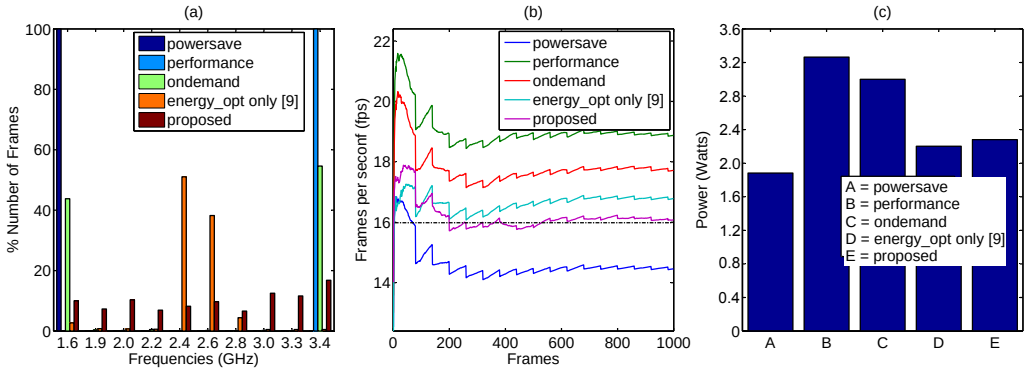

Fig. 8. Energy efficient thermal management: sobel filter case study.

Table III. Thermal improvement for sobel filter.

Techniques	Average Temperature	Peak Temperature	Thermal Cycling Related Damage
Linux ondemand	56.6°C	61°C	3.1
Energy opt. only [Juan et al. 2013]	49.9°C	59°C	2.7x
Proposed	44.1°C	50°C	1.3x

requires a lower frames per second (fps) as compared to video decoding, and the objective is to establish the fact that the existing power governors of Linux are performance agnostic, resulting in under or over performance.

Figure 8(a) reports the frequency selection results. As can be seen, the powersave governor always selects the lowest frequency of 1.6GHz for all the video frames. This results in the lowest power consumption of 1.9W (Figure 8(c)). The performance using this approach is also the least, achieving 14.5 fps (Figure 8(b)). The fps requirement of the application is 16 and is denoted by the dotted line. The powersave governor therefore results in under-performance. On the other hand, the performance governor always selects the highest frequency of 3.4GHz, achieving 19 fps and resulting in the highest power consumption of 3.3W. This governor therefore results in over-performance. The ondemand governor performs better than the existing governors. This governor switches between 1.6GHz and 3.4GHz. The performance achieved is 17.8 fps and results in a power consumption of 3.0W. Although the power consumption of the ondemand governor is lower than that obtained using the performance governor, there is still a performance slack (17.8 fps as compared to the required 16 fps), that can be exploited to save energy.

The learning-based approach of [Juan et al. 2013] achieves the best power results by selecting primarily between 2.4GHz and 2.6GHz, achieving 16.7 fps. The power consumption using this approach is 2.2W, which is 16% higher as compared to the powersave governor and 33.3% lower as compared to the performance governor. This technique also achieves 26.7% lower power consumption compared to ondemand governor. Finally, the proposed run-time manager has even distribution of the frequency as seen in Figure 8(a), achieving 16 fps. The power consumption is 2.4W, 20% lower as compared to the Linux's default ondemand governor. It is to be noted from Figure 8(b), that the proposed run-time manager results in a performance violation untill around 450 frames. This causes the run-time manager to scale up the voltage and frequency in order to satisfy the performance requirement of 16 fps; This results in an increase of power consumption (9% higher than [Juan et al. 2013]). Although not shown explicitly

Table IV. Average temperature, peak temperature and thermal cycling related damage index for different applications.

Applications	(Average Temperature, Peak Temperature, Thermal Cycles)			
	Linux / EOpt [Juan et al. 2013]	**JOpt [Shen et al. 2012]**	**MOpt [Das et al. 2014]**	**Proposed**
CPU Intensive Applications				
[redacted]	80.7, 85, 3.1	77.7, 83, 3.6	68.9, 75, 2.5	61.0, 64, 1.8
parsec.fluidanimate	78.8, 83, 3.3	78.0, 83, 3.1	71.0, 73, 2.7	66.8, 70, 1.7
splash2.raytrace	81.2, 85, 3.7	77.8, 82, 3.2	70.4, 79, 2.1	66.3, 71, 2.1
splash2.radix	75.3, 81, 2.7	72.1, 76, 3.3	69.9, 76, 2.2	60.6, 65, 1.5
Embedded Applications				
[redacted]	51.8, 62, 7.1	50.6, 65, 6.2	47.1, 57, 2.1	40.1, 50, 2.1
mibench.fft	59.9, 69, 10	55.0, 64, 10	51.5, 60, 7.1	49.8, 53, 3.5
mibench.x264	72.6, 76, 8.7	68.9, 70, 8.3	65.6, 70, 5.3	58.1, 60, 3.1

Summary: Average Improvement of the Proposed Approach				
[redacted]	**Linux / EOpt [Juan et al. 2013]**	**JOpt [Shen et al. 2012]**	**MOpt [Das et al. 2014]**	**Static [Das et al. 2015a]**
Avg. Temperature	14°C	11°C	6°C	20°C
Peak Temperature	16°C	13°C	8°C	16°C
Thermal Cycling	54%	54%	27%	407%

in this figure, the proposed approach is within 5% of the energy consumption of [Juan et al. 2013].

Finally, Table III reports the average temperature, maximum temperature, and thermal cycling related damage index for this case study, comparing the proposed run-time manager with [Juan et al. 2013] and Linux's default ondemand governor. As can be seen from this table, the proposed run-time manager reduces average temperature by 5.8°C , the peak temperature by 9°C and thermal cycling-related damage by 2x compared to the learning-based energy minimization approach [Juan et al. 2013]. In comparison to the Linux default governor, the proposed approach reduces the average temperature by 12.5°C , the peak temperature by 11 °C and thermal cycling related damage by 2.4x. To summarize the results for this case study: *the proposed run-time manager reduces power consumption by 20% with respect to default Linux. This reduction comes with additional benefit of thermal improvement of 12.5°C in terms of average temperature, 11°C in terms of peak temperature and 2.4*x *in terms of thermal cycling related damage, while delivering the required performance.* Performance, energy and temperature results are further analyzed in the subsequent sections.

5.4. Thermal Improvement

Table IV reports different thermal aspects obtained using the proposed approach for four CPU intensive applications and three embedded applications. The CPU intensive applications are obtained from the PARSEC and t heSPLASH2 benchmark suites and the embedded applications from the MiBench benchmark suite. The results obtained from the proposed run-time manager is compared with three state-of-the-art approaches – Linux's default ondemand governor, the joint optimization approach of [Shen et al. 2012], and the MTTF maximization technique of [Das et al. 2014].

A general trend that can be followed from this table is that for CPU intensive applications, the average and peak temperatures are higher than that for embedded applications. The exception for x264 application (row 11) is due to the 1080p resolution input video used for this application, which requires more CPU cycles to decode. On

Table V. Energy (KJ) and performance (fps) results for different applications.

Applications	Performance Requirement	Energy (Performance)				
		Linux	EOpt [Juan et al. 2013]	JOpt [Shen et al. 2012]	MOpt [Das et al. 2014]	Proposed
CPU Intensive Applications						
[redacted]	12.5 fps	13.7 (17.10)	11.1 (12.58)	13.3 (12.63)	14.1 (12.67)	12.5 (12.47)
parsec.fluidanimate	10.0 fps	23.3 (13.4)	17.7 (9.3)	22.8 (9.7)	23.2 (13.5)	18.4 (10.3)
splash2.raytrace	5.0 fps	16.4 (12.3)	10.1 (8.8)	13.9 (11.2)	21.8 (12.9)	10.8 (9.7)
splash2.radix	2.5 fps	8.2 (3.73)	6.5 (2.60)	8.9 (2.78)	9.1 (2.92)	7.9 (2.80)
Embedded Applications						
[redacted]	15.4 fps	1.0 (17.54)	0.8 (15.74)	0.9 (15.80)	1.1 (17.71)	0.9 (16.80)
mibench.fft	3.3 fps	5.0 (4.09)	4.1 (3.29)	4.9 (3.31)	5.6 (4.32)	4.8 (3.30)
mibench.x264	24 fps	1.4 (32.1)	1.25 (28.9)	1.5 (31.1)	1.6 (33.4)	1.28 (29.3)

the other end, the thermal cycling (represented as thermal damage) for embedded applications are higher than that for CPU intensive applications.

As seen for the x264 application, DVFS based control adopted in [Shen et al. 2012] improves the average temperature by 3.7°C and peak temperature by 6°C (row 11 column 3 vs column 2) with respect to Linux's ondemand governor. However, thermal cycling related damage is not improved significantly (less than 5%). *Thus, DVFS in isolation is not sufficient to alleviate all temperature aspects.* The MTTF maximum technique of [Das et al. 2014] uses thread affinity and DVFS combination to optimize average temperature and thermal cycling; reducing average temperature by 7°C, this approach is able to reduce thermal cycling related damage by 40% compared to Linux's default governor (row 11 column 4 vs column 2). *This improvement signifies the importance of the two OS levers (DVFS and thread affinity) in controlling the different thermal aspects.* In comparison to all these approaches, the proposed run-time manager (row 11, column 5) is able to achieve best results – improving the average temperature by 14.5°C, peak temperature by 16°C and thermal cycling related damage by 2.8x with respect to Linux. The improvement with respect to the MTTF optimization approach of [Das et al. 2014] are 7.5°C, 10°C and 41.5%, respectively. *This improvement signifies the importance of the hierarchical nature of the proposed run-time manager to delegate a finer control on the three temperature-related aspects*. Similarly, results for the other applications can be analyzed. The average improvements of the proposed run-time manager compared to these state-of-the-art approaches (including the design-time-based static approach of [Das et al. 2015a][4]) are summarized in rows 13-16 of the table. *As can be seen, the proposed run-time manager outperforms all the existing approaches either in terms of average temperature, peak temperature or thermal cycling related damage.*

5.5. Energy Improvement and Performance Deviation

Table V reports the energy consumption (in KJoules) and the performance (in frames per second) of the proposed run-time manager for the same set of applications in comparison with the state-of-the-art approaches. The energy optimization technique of [Juan et al. 2013] is included for comparison to determine the distance from energy optimality of the proposed run-time manager. As seen from this table, the ondemand governor is pessimistic for most applications, selecting higher voltage-frequency values to always meet the performance requirement. This is due to the performance agnostic nature of this governor. The technique of [Juan et al. 2013] achieves the lowest energy. This is because this technique determines the minimum voltage-frequency setting needed to execute a given application at its desired performance requirement.

[4]The significant improvement with respect to the design-time-based approach justifies the central theme of this work i.e., run-time thermal management of applications with dynamic workloads.

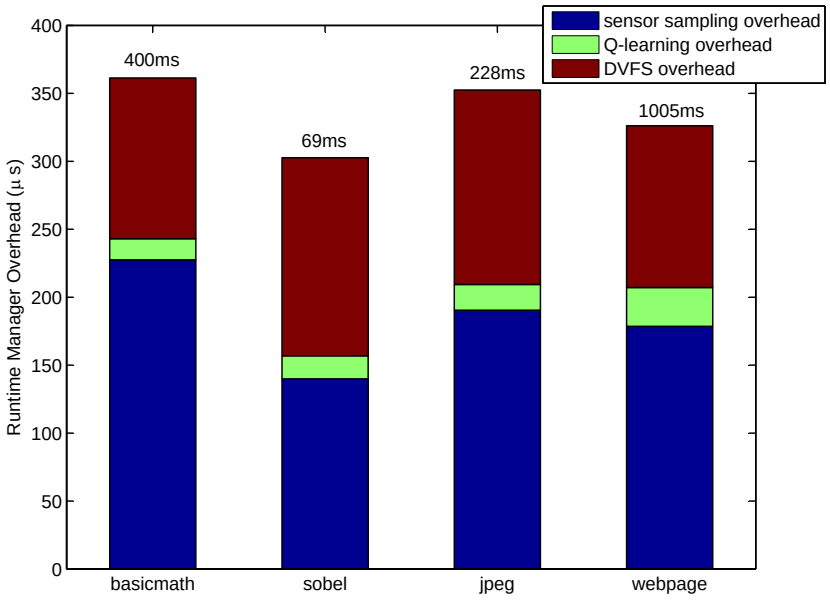

Fig. 9. Time overhead of the proposed run-time manager.

However, all the thermal aspects are not considered in this approach. The joint optimization technique [Shen et al. 2012] achieves a better result, sacrificing 25% in energy (on average) with respect to [Juan et al. 2013] to incorporate temperature in the optimization objective. However, as discussed in previous sections, thermal cycling is not considered in this approach. The MTTF optimization approach [Das et al. 2014] does not consider energy consumption and therefore, this approach leads to a significant energy overhead (116% for raytrace, average 45% for all applications) with respect to [Juan et al. 2013]. Finally, the proposed run-time manager satisfies the performance requirement for most applications and results in an energy reduction of average 15% as compared to Linux, 11% as compared to [Shen et al. 2012], and 21% as compared to [Das et al. 2014]. It is to be noted that, although thread affinity-based control (used in [Das et al. 2014] and proposed) alleviates thermal cycling-related reliability significantly, the execution time is extended. This results in the performance dropping below the frames per second requirement. To overcome this, the run-time manager raises the operating frequency slightly, resulting in a 9% increase in the energy consumption with respect to the energy minimum result of [Juan et al. 2013], which does not perform thermal management. To summarize, the proposed run-time manager outperforms any of the existing approaches in terms of energy-efficient thermal management.

5.6. Overheads of the Run-time Manager

Figure 9 plots the overhead of the proposed run-time manager for four applications. A stacked bar is plotted for each application, representing three key components – overhead of sampling the thermal sensors (identified in the figure as *sensor sampling overhead* and represented as the bottom stack), overhead of the Q-learning algorithm (identified in the figure as *Q-learning overhead* and represented as the middle stack), and the overhead for switching the voltage and frequency (identified in the figure as *DVFS overhead* and represented as the top stack). Finally, the number on each stacked

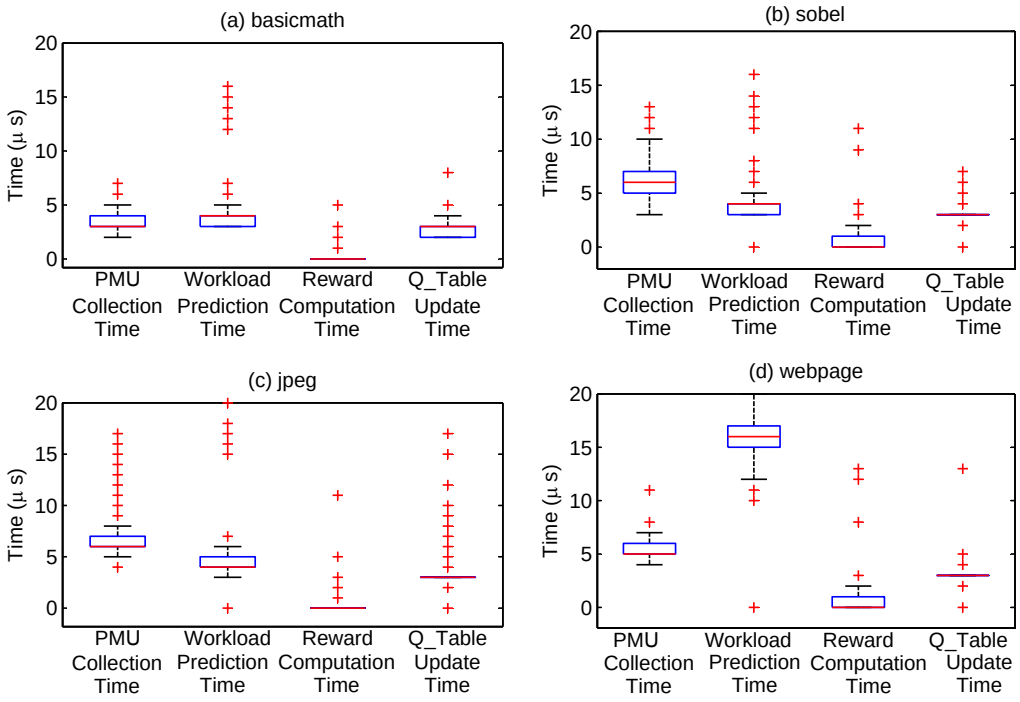

Fig. 10. Time overhead of the Q-learning algorithm.

bar represents the total time for processing a frame of the corresponding application. Other components constitute less than $5\mu s$ in total and are not included in the plot.

As can be seen from the figure, the overhead of the run-time manager constitutes between 0.003% to 0.43% only. A point to note from the figure is that, the Q-learning algorithm takes on average $20\mu s$ to execute. For some application such as webpage, the overhead is higher as compared to other applications. It is to be note that in addition to the above overheads, the thread allocation overhead is on average 300 μs and is incurred once every 100 frames (i.e., LTI) when the *thread affinity* is altered.

To give further insight into the overhead of the Q-learning algorithm itself, Figure 10 plots the different components of the Q-learning algorithm as box plot showing the variation as well as the median value. Results in the figure are shown for the same four applications with 20 different executions. As can be seen from the figure, the time to collect the performance statistics (indicated in the figure as *PMU collection time*) for *basicmath* application varies from $2.5\mu s$ to $8\mu s$. The top and the bottom edges of the blue bounding box represents respectively, the 25th and the 75th percentile of the *PMU collection time*, and the horizontal line (corresponding to $3\mu s$) marks the median value. The outliers are plotted as crosses.

6. CONCLUSION

A low overhead hierarchical run-time manager is proposed for multi-/many-core embedded systems for energy-efficient thermal management. The run-time manager employs Q-learning to select the minimum frequency of the hardware for a given thread allocation, determined using a greedy heuristic. The run-time manager is implemented on a real system supporting the POSIX programming model. Experiments conducted with CPU intensive and embedded benchmarks demonstrate that the proposed ap-

proach reduces energy consumption by 15% with respect to Linux, simultaneously improving the lifetime reliability by 1.5x. Our continuing work includes Kernel implementation of the run-time manager, extending the approach to include heterogeneous components such as GPUs and FPGAs.

Acknowledgment

This work was supported in parts by the EPSRC Grant EP/L000563/1 and the PRiME Programme Grant EP/K034448/1 (www.prime-project.org). Experimental data used in this paper can be found at DOI:10.5258/SOTON/382855 (http://dx.doi.org/10.5258/SOTON/382855).

REFERENCES

1994. IEEE Standard for Information Technology - Portable Operating System Interfaces (POSIX(R)) - Part 1: System Application Program Interface (API) - Amendment 1: Realtime Extension (C language). *IEEE Std 1003.1b-1993* (1994), 0–3. DOI:http://dx.doi.org/10.1109/IEEESTD.1994.121455

Andrew G Barto. 1998. *Reinforcement learning: An introduction*. MIT press.

Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. 1998. Dynamic Power Management of Electronic Systems. In *Proceedings on the International Conference on Computer Aided Design (ICCAD)*.

C. Bienia, S. Kumar, and K. Li. 2008. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *IEEE Symposium on Workload Characterization*. 47–56. DOI:http://dx.doi.org/10.1109/IISWC.2008.4636090

JL Chaboche and PM Lesne. 1988. A Non-Linear Continuous Fatigue Damage Model. *Fatigue & fracture of engineering materials & structures* 11, 1 (1988), 1–17.

R. Cochran, C. Hankendi, A. Coskun, and S. Reda. 2011a. Identifying the Optimal Energy-Efficient Operating Points of Parallel Workloads. In *Proceedings on the International Conference on Computer Aided Design (ICCAD)*.

Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. 2011b. Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. ACM, 175–185. DOI:http://dx.doi.org/10.1145/2155620.2155641

LF Coffin Jr. 1973. Fatigue at High Temperature. *ASTM STP* 520 (1973), 5–34.

A. K. Coskun, T. S. Rosing, and K. C. Gross. 2009a. Utilizing Predictors for Efficient Thermal Management in Multiprocessor SoCs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 10 (2009), 1503–1516. DOI:http://dx.doi.org/10.1109/TCAD.2009.2026357

Ayse K. Coskun, Richard Strong, Dean M. Tullsen, and Tajana Simunic Rosing. 2009b. Evaluating the Impact of Job Scheduling and Power Management on Processor Lifetime for Chip Multiprocessors. In *Proceedings of the Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. ACM, 169–180. DOI:http://dx.doi.org/10.1145/1555349.1555369

Jin Cui and D.L. Maskell. 2012. A Fast High-Level Event-Driven Thermal Estimator for Dynamic Thermal Aware Scheduling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 6 (2012), 904–917. DOI:http://dx.doi.org/10.1109/TCAD.2012.2183371

L. Dagum and R. Menon. 1998. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science Engineering* 5, 1 (1998), 46–55. DOI:http://dx.doi.org/10.1109/99.660313

A. Das, A. Kumar, and B. Veeravalli. 2015a. Reliability and Energy-Aware Mapping and Scheduling of Multimedia Applications on Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems* (2015). DOI:http://dx.doi.org/10.1109/TPDS.2015.2412137

Anup Das, Akash Kumar, Bharadwaj Veeravalli, Rishad Shafik, Geoff Merrett, and Bashir Al-Hashimi. 2015b. Workload Uncertainty Characterization and Adaptive Frequency Scaling for Energy Minimization of Embedded Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. San Jose, CA, USA.

Anup Das, Rishad A. Shafik, Geoff V. Merrett, Bashir M. Al-Hashimi, Akash Kumar, and Bharadwaj Veeravalli. 2014. Reinforcement Learning-Based Inter- and Intra-Application Thermal Optimization for Lifetime Improvement of Multicore Systems. In *Proceedings of the Design Automation Conference (DAC)*. ACM, Article 170, 6 pages. DOI:http://dx.doi.org/10.1145/2593069.2593199

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Communication of the ACM* 51, 1 (2008), 107–113. DOI:http://dx.doi.org/10.1145/1327452.1327492

Gaurav Dhiman, V. Kontorinis, Dean Tullsen, T. Rosing, Eric Saxe, and Jonathan Chew. 2010. Dynamic Workload Characterization for Power Efficient Scheduling on CMP Systems. In *International Symposium on Low Power Electronics and Design (ISLPED)*.

G. Dhiman and T.S. Rosing. 2009. System-Level Power Management Using Online Learning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 5 (2009), 676–689. DOI:http://dx.doi.org/10.1109/TCAD.2009.2015740

Gaurav Dhiman and Tajana Simunic Rosing. 2007. Dynamic Voltage Frequency Scaling for Multi-tasking Systems Using Online Learning. In *International Symposium on Low Power Electronics and Design (ISLPED)*.

S.D. Downing and D.F. Socie. 1982. Simple rainflow counting algorithms. *International Journal of Fatigue* 4, 1 (1982), 31 – 40.

Thomas Ebi, Mohammad Abdullah Al Faruque, and Jörg Henkel. 2009. TAPE: Thermal-aware Agent-based Power Economy for Multi/Many-core Architectures. In *Proceedings on the International Conference on Computer Aided Design (ICCAD)*. ACM, 302–309. DOI:http://dx.doi.org/10.1145/1687399.1687457

Thomas Ebi, David Kramer, Wolfgang Karl, and Jörg Henkel. 2011. Economic Learning for Thermal-aware Power Budgeting in Many-core Architectures. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM, 189–196. DOI:http://dx.doi.org/10.1145/2039370.2039401

Mohammad Al Faruque, Janmartin Jahn, and Joerg Henkel. 2010. Runtime Thermal Management Using Software Agents for Multi- and Many-Core Architectures. *IEEE Design & Test of Computers* 27, 6 (2010), 58–68.

Yang Ge and Qinru Qiu. 2011. Dynamic Thermal Management for Multimedia Applications Using Machine Learning. In *Proceedings of the Design Automation Conference (DAC)*. ACM, 95–100. DOI:http://dx.doi.org/10.1145/2024724.2024746

M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *IEEE Workshop on Workload Characterization*. 3–14. DOI:http://dx.doi.org/10.1109/WWC.2001.990739

Lei He, Weiping Liao, and Mircea R. Stan. 2004. System Level Leakage Reduction Considering the Interdependence of Temperature and Leakage. In *Proceedings of the Design Automation Conference (DAC)*. ACM, 12–17. DOI:http://dx.doi.org/10.1145/996566.996572

Haris Javaid, Muhammad Shafique, Jörg Henkel, and Sri Parameswaran. 2011. System-level Application-aware Dynamic Power Management in Adaptive Pipelined MPSoCs for Multimedia. In *Proceedings on the International Conference on Computer Aided Design (ICCAD)*. IEEE, 616–623.

Da-Cheng Juan, Siddharth Garg, Jinpyo Park, and Diana Marculescu. 2013. Learning the Optimal Operating Point for Many-core Systems with Extended Range Voltage/Frequency Scaling. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. IEEE, Article 8, 10 pages.

Hwisung Jung and M. Pedram. 2008. Continuous Frequency Adjustment Technique Based on Dynamic Workload Prediction. In *International Conference on VLSI Design*.

Hwisung Jung and M. Pedram. 2010. Supervised Learning Based Power Management for Multicore Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29, 9 (2010), 1395–1408.

Umair Ali Khan and Bernhard Rinner. 2014. Online Learning of Timeout Policies for Dynamic Power Management. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4, Article 96 (2014), 96:1–96:25 pages. DOI:http://dx.doi.org/10.1145/2529992

SS Manson. 1972. *The Challenge to Unify Treatment of High-temperature Fatigue: A Partisan Proposal Based on Strainrange Partitioning*. National Aeronautics and Space Administration.

Pietro Mercati, Andrea Bartolini, Francesco Paterna, Tajana Simunic Rosing, and Luca Benini. 2013. Workload and User Experience-aware Dynamic Reliability Management in Multicore Processors. In *Proceedings of the Design Automation Conference (DAC)*. ACM, Article 2, 6 pages. DOI:http://dx.doi.org/10.1145/2463209.2488735

Pietro Mercati, Andrea Bartolini, Francesco Paterna, Tajana Simunic Rosing, and Luca Benini. 2014. A Linux-governor Based Dynamic Reliability Manager for Android Mobile Devices. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. European Design and Automation Association, Article 104, 4 pages.

Santiago Pagani, Heba Khdr, Waqaas Munawar, Jian-Jia Chen, Muhammad Shafique, Minming Li, and Jörg Henkel. 2014. TSP: Thermal Safe Power: Efficient Power Budgeting for Many-core Systems in Dark Silicon. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

D. Rai, Hoeseok Yang, I Bacivarov, Jian-Jia Chen, and L. Thiele. 2011. Worst-case Temperature Analysis for Real-Time Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. 1–6. DOI:http://dx.doi.org/10.1109/DATE.2011.5763104

Lars Schor, Iuliana Bacivarov, Hoeseok Yang, and Lothar Thiele. 2013. Efficient Worst-Case Temperature Evaluation for Thermal-Aware Assignment of Real-Time Applications on MPSoCs. *Journal of Electronic Testing* 29, 4 (2013), 521–535. DOI:http://dx.doi.org/10.1007/s10836-013-5397-5

S. Sharifi, D. Krishnaswamy, and T.S. Rosing. 2013. PROMETHEUS: A Proactive Method for Thermal Management of Heterogeneous MPSoCs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 7 (2013), 1110–1123. DOI:http://dx.doi.org/10.1109/TCAD.2013.2247656

Hao Shen, Jun Lu, and Qinru Qiu. 2012. Learning based DVFS for Simultaneous Temperature, Performance and Energy Management. In *Proceedings of the International Symposium on Quality Electronic Design (ISQED)*. 747–754. DOI:http://dx.doi.org/10.1109/ISQED.2012.6187575

Hao Shen, Ying Tan, Jun Lu, Qing Wu, and Qinru Qiu. 2013. Achieving Autonomous Power Management Using Reinforcement Learning. *ACM TODAES* (2013).

Bing Shi, Yufu Zhang, and A. Srivastava. 2013. Dynamic Thermal Management Under Soft Thermal Constraints. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)* 21, 11 (2013), 2045–2054. DOI:http://dx.doi.org/10.1109/TVLSI.2012.2227854

Tajana Simunic, Luca Benini, Andrea Acquaviva, Peter Glynn, and Giovanni De Micheli. 2001. Dynamic Voltage Scaling and Power Management for Portable Systems. In *Proceedings of the Design Automation Conference (DAC)*.

F. Sironi, M. Maggio, R. Cattaneo, G.F. Del Nero, D. Sciuto, and M.D. Santambrogio. 2013. ThermOS: System Support for Dynamic Thermal Management of Chip Multi-processors. In *International Conference on Parallel Architectures and Compilation Techniques*. DOI:http://dx.doi.org/10.1109/PACT.2013.6618802

Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. 2004. Temperature-Aware Microarchitecture: Modeling and Implementation. *ACM Transactions on Architecture and Code Optimization (TACO)* 1, 1 (2004), 94–125. DOI:http://dx.doi.org/10.1145/980152.980157

Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. 2004. The Case for Lifetime Reliability-Aware Microprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. IEEE, 276–287.

Matthew J Walker, Anup Das, Geoff V Merrett, and BM Hashimi. 2015. Run-time power estimation for mobile ad embedded asymmetric multi-core CPUs. *HiPEAC Workshop on Energy Efficiency with Heterogenous Computing* (2015).

Rong Ye and Qiang Xu. 2014. Learning-Based Power Management for Multicore Processors via Idle Period Manipulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 7 (2014), 1043–1055. DOI:http://dx.doi.org/10.1109/TCAD.2014.2305838