On the Impacts of Greedy Thermal Management in Mobile Devices

Onur Sahin, *Student Member, IEEE,* Ayse K. Coskun, *Member, IEEE,*

***Abstract*—As mobile system-on-chips incorporate multicore processors with high power densities, high chip temperatures are becoming a rising concern in mobile processors. Modern smartphones are limited in their cooling capabilities and employ CPU throttling mechanisms to avoid thermal emergencies by sacrificing performance. Traditional throttling techniques aim at achieving maximum utilization of the available thermal headroom so as to minimize the performance penalty at a given time. This letter demonstrates that such greedy techniques lead to fast elevation of temperature on other system components and cause substantially sub-optimal performance over increased durations of phone activity. Through experiments on a commercial smartphone, we characterize the impact of application duration on throttling-induced performance loss and propose quality-of-service (QoS) tuning as an effective way of providing the mobile system user with *consistent performance levels over extended application durations*. The proposed QoS-aware frequency capping technique achieves up to 56% improvement in *performance sustainability*.**

***Index Terms*—Smartphones, thermal throttling, performance sustainability**

I. Introduction

DRIVEN by the increasing user demand and competitive mobile phone market, modern smartphones push the boundaries of power and performance envelope by incorporating high-performance multicore processors and multiple application-specific integrated circuits into a single chip. Due to increased power densities, however, maintaining safe chip temperatures is becoming increasingly challenging. Elevated chip temperatures lead to lower device reliability and increase the overall power consumption due to the exponential dependence of leakage power on temperature. Therefore, dynamic thermal management (DTM) techniques form an essential piece of the runtime management stack in mobile systems.

Smartphones present unique challenges in thermal management since multiple heat generating components such as battery, display, and CPU are incorporated into a small form factor package with limited cooling capabilities. State-of-the-art smartphones employ CPU throttling techniques which utilize dynamic voltage and frequency scaling (DVFS) to slow down the processor and/or turn off individual CPU cores in case of thermal emergencies. Existing DTM solutions, however, are greedy as they aim to maximize performance within processor thermal limits without considering the other components that share the same thermal budget. This letter demonstrates that such DTM on CPU results in faster elevation of temperature on other components (i.e., the battery) and quick exhaustion of available thermal budget, which, in turn, causes major performance loss over longer durations of phone use. This performance loss contrasts with the mobile system user's expectations, as the users often demand consistent performance standard for applications that run for minutes or longer (i.e., consistent frame rate in graphics/video streaming [7] or webpage rendering time in browsing [12]). Thus, we should rethink the thermal management techniques in mobile devices for providing *performance sustainability* rather than favoring short term performance without considering the thermal impacts of other components in the smartphone.

Manuscript received February 06, 2015; accepted March 25, 2015. Date of publication April 07, 2015. Recommended for publication by W. Zhao.

The authors are with Boston University, Boston, MA 02215 USA (email: sahin@bu.edu; acoskun@bu.edu)

Color versions of one or more of the figures in this letter are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/LES.2015.2420664

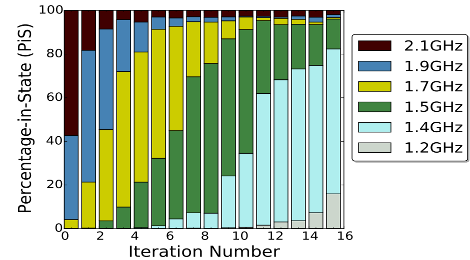
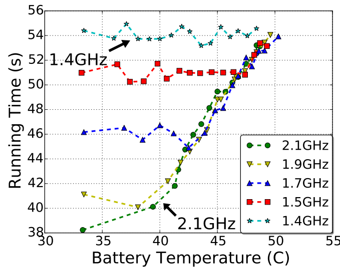


Fig. 1: Frequency residencies over time on MDP8974 smartphone during continuous use.

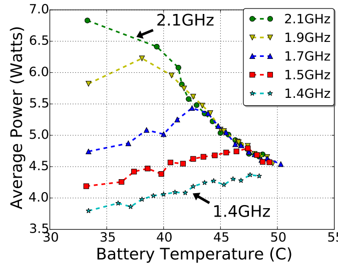
Consider the example given in Figure 1, which shows the frequency residencies when the FFT application from Scimark suite [8] is repeatedly run on a Qualcomm Snapdragon MDP8974 smartphone [5]. The default ondemand frequency governor [6] seeks to scale the CPU frequency to the highest level due to high CPU load. Initially, the baseline DTM policy allows the application to utilize the highest three frequencies to boost performance while meeting the thermal constraints. With increasing number of iterations, however, there is a clear shift towards using lower frequencies, which significantly reduces performance over time. For instance, in the last iteration, more than 80% of the running time is spent at 1.4GHz and 1.2GHz, while the application was well able to run without scaling down to those two frequencies initially. This example illustrates that greedy exhaustion of available thermal budget results in significantly reduced performance over extended durations of mobile phone use, degrading performance sustainability. We demonstrate that performance sustainability can be significantly improved by making the thermal policies adaptive to application QoS requirements.

The major contributions of this paper are as follows:

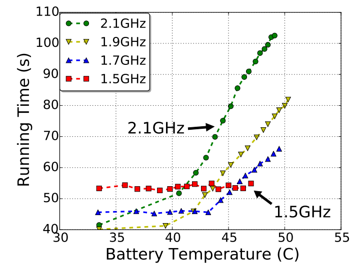
- To the best of our knowledge, our work is first to analyze the impact of battery temperature and duration of phone activity on performance on a working smartphone.
- We evaluate the impact of application duration and battery temperature on performance sustainability in the



(a) Performance vs. Battery Temperature with PID



(b) Power vs. Battery Temperature with PID



(c) Performance vs. Battery Temperature with Stop-Go

Fig. 2: Impact of Battery Temperature on CPU Power/Performance with Different CPU Frequency Limits

context of existing DTM techniques (Section III).

- We propose a *QoS-aware Frequency Capping* policy to tune application QoS level and show that QoS-aware throttling can improve performance sustainability by as much as 56% (Section IV).

II. Experimental Methodology

A. Target Platform & Applications

We perform experiments and test our policies on a state-of-the-art Qualcomm Snapdragon MSM8974 smartphone platform [5]. The chipset contains a Quad Core Krait 400 CPU (a common CPU in mainstream smartphones such as Nexus 5 and Samsung S4) along with an Adreno 330 GPU, 2GB LPDDR3 RAM and is powered by a 1,600mAh Li-ion battery. The phone runs Android Jelly Bean version 4.1.2 and Linux kernel 3.4.0. The Krait 400 CPU supports 12 operating frequencies ranging from 300MHz to 2.1GHz. The phone allows per-core temperature measurements via on-chip thermal sensors where one sensor is assigned to each core. Temperature sensors for the CPU cores and the battery can be read through the thermal virtual file system provided by the Linux kernel (i.e., /sys/class/thermal) with ±1°C accuracy. We use *perf_event* kernel API for accessing hardware performance counters. Our phone allows for measuring only the overall power consumption. Since our focus is on the CPU power consumption, we disable cellular activities and Wi-Fi by switching the phone into airplane mode and turning-off the LCD display throughout the measurements.

We run a set of applications from various domains. Three scientific computing kernels, FFT, LU and SOR, are selected from Scimark 2.0 [8], which is a benchmark suite for testing Java based platforms. Such kernels are commonly found in image/video processing and emerging mobile healthcare applications. A video encoding (H.264) and an artificial intelligence application (Sjeng) are chosen from the SPEC CPU2006 [4].

B. Implementation of Policies

DVFS-Based Feedback Controller. The default CPU frequency scaling policy in our phone and in most state-of-the-art mobile devices is the *ondemand governor* [6], which adjusts the CPU frequency based on CPU load. Thermal management in modern smartphone platforms is accomplished by a variant of *thermal-daemon (thermald)* [1] that uses a PID controller to keep the temperature below a thermal set-point by adjusting the maximum CPU frequency limits of the *ondemand governor*. The *ondemand* governor is forced to use the frequencies that are below this assigned limit. As thermald is not open source, we implement our own baseline DVFS-based PID controller as a user-level program that adjusts the frequency limits, and manually tune the controller's parameters to mimic *thermald*. Frequency limit assignments and sensor readings are performed every 100ms to enable non-intrusive (less than %1 execution overhead) thermal management while maintaining sufficient time granularity to avoid thermal emergencies.

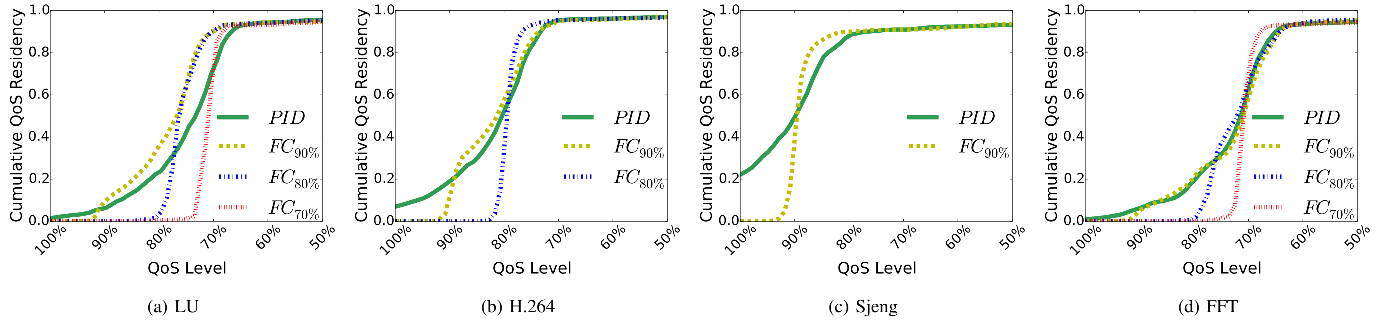
Stop-Go Policy. Some DTM policies change the number of active cores to meet the thermal constraints. Similarly, our Stop-Go mechanism is based on *offlining*, which allows the OS to power down individual cores and schedule threads onto remaining online cores. Whenever peak CPU temperature reaches 93°C, our Stop-Go controller offlines 3 cores until the temperature reduces to 87°C. The three cores either operate at the maximum frequency limit (*go phase*) that we assign or stay in offline state (*stop phase*). Due to OS restrictions, we are not able to power-down the first core. We disable the default *mpdecision* tool that manages the number of active cores to prevent interference with our Stop-Go policy. We set the polling interval for thermal sensors to 50ms for this policy.

III. Analysis of Current Throttling Strategies

In this section, we analyze the impacts of application duration and elevated battery temperature on performance with two common DTM techniques (Feedback controller and Stop-Go) and motivate the idea of sustained performance. We present our analyses for the FFT benchmark and leave the discussion on other applications to Section 4.

Battery Temperature and Performance Interaction. In order to evaluate the PID controller for different application durations, we design an experiment in which we run the application repeatedly under 5 different maximum frequency limits and record the battery temperature, power, CPU frequency, and running time.

Figure 2a shows how the running time changes as the battery temperature increases over time for the PID controller. Each data point represents the initial battery temperature of that iteration and the resulting running time. Let us assume close to 46 seconds (80% of maximum performance) is an acceptable running time for the user. 2.1GHz maximum frequency limit, which is the default maximum limit for the baseline policy, greedily aims to provide maximum performance at every run but causes a quick increase in battery temperature which reaches 42.5°C at the 5th iteration at *200 seconds*. Increased battery temperature induces more aggressive CPU throttling and running time increases sharply. When the maximum



(a) LU (b) H.264 (c) Sjeng (d) FFT

Fig. 3: Fraction of running time spent above a QoS level via baseline and QoS tuning ("$FC_{X\%}$" represents the proposed policy with X% target QoS).

frequency is limited to 1.7GHz, however, the application can run for a longer time before reaching 42.5°C, sustaining a running time of 46 seconds for 6 iterations and *270 seconds*. After the battery heats to 42.5°C, both settings achieve similar performance. Even though two traces achieve same the performance after a certain battery temperature point, the time it takes to reach this point differs upon power and battery temperature history. Note that small variations in running time occur as the phone is active with default background processes during the experiment.

The best running time at any battery temperature is achieved at the highest maximum frequency limit with PID controller. For the Stop-Go policy, however, utilizing lower maximum frequency limits during *go* phases at higher battery temperature levels significantly improves performance as shown in Figure 2c. For instance, limiting the maximum frequency to 1.5GHz provides 50% lower running time compared to default 2.1GHz when the battery temperature is 46°C. These experiments show that heat dissipation from the processor is adversely affected by the increasing battery temperature and we observed that utilizing lower frequencies at higher battery temperature allows for longer *go* phases, thereby, improves performance.

Battery Temperature and Power Interaction. As a result of the thermal coupling between the battery and the processor, the maximum power of the processor is also limited by the battery temperature. This case is illustrated in Figure 2b for the PID controller. The curve corresponding to 2.1GHz forms an upper bound on the maximum power that could be consumed, which reduces by 32% as the battery temperature increases from 33°C to 49°C. With the increasing battery temperature and over the iterations, the power dissipation with the lower 3 maximum frequency limits rises due to the increased leakage power. This leads to diminishing performance where the maximum achievable power is significantly reduced by the battery temperature and the share of dynamic power is shrinked due to increased leakage power in a small form factor mobile device.

IV. QoS Tuning for Performance Sustainability

The previous section shows that greedy thermal management approaches that aim to provide maximum performance at any battery temperature level degrade performance sustainability as performance drops significantly with increased application durations and battery temperatures. In this section, we show that performance sustainability could be significantly improved in a mobile system by making thermal policies aware of application requirements rather than letting the CPU consume available thermal headroom quickly to provide instant maximum performance. To this end, we propose a *QoS-aware frequency capping policy* for maximizing sustained performance and compare against the baseline PID controller.

Algorithm 1 QoS-aware Frequency Capping Policy

Every 100ms

1: Read Instructions Per Second (IPS) & Max. Chip Temperature
2: **if** $currIPS > targetIPS$ **then**
3: *FrequencyCap++*
4: **else if** $currIPS < targetIPS$ **then**
5: *FrequencyCap--*
6: **end if**
7: Clip FrequencyCap to range 300MHz-2.1GHz
8: PidController(Max. Temperature, FrequencyCap)

A. QoS-Aware Frequency Capping

Our policy sets maximum limits to CPU frequency as a control knob for power and performance. Since our focus is on CPU applications, we use throughput as an indicator of application's QoS level, measured in *millions of instructions per seconds* (MIPS). We provide our QoS-aware frequency capping policy in Algorithm 1. The desired QoS level, which could be given by the user or provided upon battery-level, duration of use, etc., is provided as input to our implementation. The policy calculates the throughput by reading the instruction counts from hardware performance counters. If the policy detects that the current QoS level is larger than the given threshold, it will limit the CPU from exhausting the thermal budget by restricting the use of higher frequencies. The current frequency cap and maximum temperature are also fed to the PID controller to assign the new maximum frequency limits for the *ondemand* CPU governor.

B. Results

We use the QoS-aware frequency capping policy presented in the previous section to show the benefits of QoS tuning on performance sustainability. We explore three QoS levels which are set to 90%, 80% and 70% of the average throughput achieved when the application is run only once on an initially cold system. All the experiments are initialized at the same battery temperature level. To emulate longer application durations, we run the applications repeatedly and, for each application, we set the number of iterations based on a maximum battery temperature limit of 55°C.

Figure 3 shows the cumulative QoS residencies (i.e., a data point indicates the fraction of running time spent above the corresponding QoS level) for the four applications. In an ideal

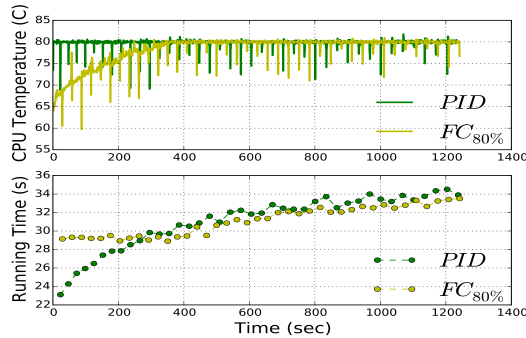


Fig. 4: Running time (bottom) and CPU temperature (top) over time for the iterative run of SOR application.

scenario, when the requested QoS is 80%, the curve would take a sharp increase to 1 at 80%, indicating that 100% of the running time is spent above the 80% QoS level.

For the power hungry Java application, LU, baseline PID controller can sustain the QoS above 80% for only 32% of the running time, as the CPU quickly consumes the thermal budget and more aggressive thermal throttling is induced over time. Tuning QoS to 80% via frequency capping, however, sustains the QoS above this constraint for 50% of the running time, providing 56% longer sustainability over the baseline. For the video encoding benchmark, H.264, tuning the QoS to 90% and 80% provides 22% and 33% longer sustainability, respectively, compared to the baseline. From the user's perspective, this would correspond to significantly longer time a certain video quality could be sustained without introducing performance drops. Tuning the QoS to 90% extends the duration of time spent above this QoS level from 57% to 72% for the Sjeng artificial intelligence application. Lower two QoS levels are omitted for clarity, as the QoS does not drop below 82%. Tuning the QoS to 90% and 80% for the FFT does not significantly improve sustainability as the CPU still quickly reaches the thermal limits due to high application power.

Figure 4 shows the CPU temperature and running time per iteration for the SOR application with 80% QoS constraint. The baseline controller allows the CPU to greedily operate at maximum thermal limits, which degrades the performance over time and, after 270 seconds, frequency capping starts to provide better running time. Even after the CPU reaches thermal limits at 400 seconds and running time starts to increase, frequency capping still outperforms the baseline due to lower battery temperature and less aggressive throttling. We observe that the baseline controller results in 4°C higher battery temperature, on average, compared to the proposed policy.

V. Related Work

Thermal management of multicore CPUs and MPSoCs have been well studied for conventional computer systems. Control theoretic DVFS techniques provide effective temperature control while maximizing performance [2][3]. Use of statistical predication techniques (e.g., [11]), have also been proposed to forecast thermal emergencies to prevent temperature violations. Several attempts have been made towards thermal modeling and analysis of the mobile devices in particular. Xie et al. [9] propose a resistance network based thermal simulation framework for obtaining component level steady-state temperatures. In a recent study [10], the authors derive an RC model of the thermal coupling between the battery and the application processor by disassembling the phone and extracting the thermal properties of individual components while we focus on the performance impacts of battery temperature and application duration on a working smartphone. A sample study of quality tuning in a mobile system is done by Pathania et al. [7], where the authors propose a CPU-GPU power budgeting algorithm to meet a frames-per-second constraint, without considering quality sustainability and thermal impacts.

Our work differs from the prior work in the following major aspects: (1) we address the thermal management problem in mobile devices from a performance sustainability perspective to meet the user expectations, (2) we reveal the significance of application-level QoS tuning for providing the mobile system user with longer durations of sustained performance.

VI. Conclusion

In this letter, through experiments on a modern smartphone, we showed that current DTM approaches lead to significant performance degradation as the application durations get longer and the battery temperatures increase, leaving mobile system users with diminishing performance over time. We stated the effectiveness of adapting the thermal management policies to application/user QoS requirements for substantially extending the durations of consistent performance, and proposed a QoS-aware frequency capping technique that achieved up to 56% longer performance sustainability.

References

[1] Intel open source technology center. https://01.org/linux-thermal-daemon/documentation/introduction-thermal-daemon, 2014.

[2] A. Bartolini, M. Cacciari, A. Tilli, and L. Benini. A distributed and self-calibrating model-predictive controller for energy and thermal management of high-performance multicores. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2011.

[3] J. Donald and M. Martonosi. Techniques for multicore thermal management: Classification and new exploration. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 78–88, 2006.

[4] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[5] Q. D. Network. MS Windows NT kernel description. https://developer.qualcomm.com/snapdragon-mobile-development-platform-mdp, 2014.

[6] V. Pallipadi and A. Starikovskiy. The ondemand governor. In *Proceedings of the Linux Symposium*, volume 2, pages 215–230, 2006.

[7] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra. Integrated cpu-gpu power management for 3d mobile games. In *Proceedings of the 51st Annual Design Automation Conference (DAC)*. ACM, 2014.

[8] R. Pozo and B. Miller. Scimark 2.0. *URL: http://math. nist. gov/scimark2*, 2000.

[9] Q. Xie, M. J. Dousti, and M. Pedram. Therminator: a thermal simulator for smartphones producing accurate chip and skin temperature maps. In *Proceedings of the 2014 international symposium on Low power electronics and design (ISLPED)*, pages 117–122. ACM, 2014.

[10] Q. Xie et al. Dynamic thermal management in mobile devices considering the thermal coupling between battery and application processor. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013.

[11] I. Yeo, C. C. Liu, and E. J. Kim. Predictive dynamic thermal management for multicore systems. In *DAC*, 2008.

[12] Y. Zhu and V. J. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, Washington, DC, USA, 2013.