JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 12, No. 4, December 2018, pp. 157-169

User-Centric Thermal Management for Smartphones

Wook Song
Samsung Research, Samsung Electronics, Seoul, Korea
wook16.song@samsung.com

Jihong Kim*
Department of Computer Science and Engineering, Seoul National University, Seoul, Korea
jihong@davinci.snu.ac.kr

Abstract

For high-performance smartphones, keeping the on-chip temperature under a given critical temperature is a major concern. In order to prevent the temperature from rising above the critical point, modern smartphones widely adopt the dynamic thermal management (DTM) scheme, which limits the maximum CPU frequency when the CPU reaches high temperatures (thus making the CPU temperature drop). In this paper, we propose a novel DTM scheme based on user-perceived response time analysis called SmartDTM. Unlike existing DTM schemes that can significantly degrade the quality of user experience, SmartDTM explicitly accounts for the quality of the user experience in making DTM decisions. We divide an execution of a given user-interactive session into two intervals, one where the system response time directly affects the user experience and the other where the system response time does not affect the user experience. In the user-perceived response time interval, our proposed scheme conservatively makes DTM decisions so that the quality of the user experience is not affected by the reduced maximum CPU frequency. On the other hand, in the user-oblivious response time interval, SmartDTM aggressively lowers the CPU frequency so that the CPU temperature can be quickly decreased to a safe level without negatively affecting user experience. Our experimental results on an ODROID-XU+E board show that SmartDTM can improve the performances of user-perceived intervals by 12.2% and 21.4% over the Android's default DTM policy when the initial temperature was set to 65°C and 70°C, respectively, under the critical temperature of 85°C.

Category: Smart and Intelligent Computing

Keywords: Smartphone; Operating system; User-centric optimization; Thermal management; Dynamic voltage and frequency scaling

I. INTRODUCTION

Modern smartphones employ high-performance processors in order to satisfy the demanding computing requirements of various apps (such as mobile games and virtual reality apps) to deliver great user experiences. For example, recent smartphones (such as Galaxy S7) have octa-core processors which run at more than 2.0 GHz [1]. Although these high-performance processors are essential to support the required user-experience level, their high power densities produce excessive amounts of heat, which must be maintained under certain threshold temperatures

Open Access http://dx.doi.org/10.5626/JCSE.2018.12.4.157 http://jcse.kiise.org

This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

Received 13 November 2018; **Accepted** 02 December 2018
*Corresponding Author

Copyright © 2018. The Korean Institute of Information Scientists and Engineers
pISSN: 1976-4677 eISSN: 2093-8020

[2]. Since a highly elevated on-chip temperature negatively affects reliability, effective thermal management is a crucial design requirement.

Conventional computing systems such as PCs employ thermal management hardware devices to cool their processors. For example, a fan-cooled heat sink [3] is the most commonly used technology for on-chip temperature reduction in PCs. However, it is difficult to use this approach in a smartphone due to its form-factor limitation as well as the increased power consumption caused by this approach [4]. Therefore, for smartphones, software-based thermal management is a more practical way to address thermal problems.

As a software-based approach, the dynamic thermal management (DTM) [5] scheme is commonly used in smartphones. The DTM scheme aims to maintain the CPU temperature below a *critical temperature* (above which the processor chip could be damaged). For this purpose, the current temperature is sensed periodically, and, if necessary, the CPU frequency is reduced by interacting with the dynamic voltage and frequency scaling (DVFS) scheme. When the CPU temperature reaches a predefined *trigger temperature* (which is lower than the critical temperature), the maximum operating frequency of the processor, which is set as the limit for the current DVFS policy, is reduced. Then, when the CPU temperature drops below the trigger temperature, the maximum operating frequency of the processor is gradually increased in order to restore system performance.

Although the DTM scheme can effectively mitigate the thermal problems, it makes voltage/frequency scaling decisions based only on the current temperature without considering the user's current computing requirement. Therefore, under existing DTM techniques, the quality of the user experience can be significantly degraded if a DTM decision to lower the CPU voltage/frequency is made when meeting a high computing requirement is necessary for a great user experience. If a DTM technique can be extended to be more *user experience-aware*, it may avoid such scaling decisions in the middle of compute-intensive user-smartphone interactions. Considering that most user-interactive sessions can be divided into two parts, one where the system performance level directly affects the quality of the user experience (called the display-sensitive part) and the other where the system performance level does not affect the quality of the user experience (called the display-insensitive part), it is possible to make DTM techniques smarter if we can distinguish between the display-sensitive part and display-insensitive part of a user-interactive session.

In this paper, we propose such a novel DTM technique for smartphones, called *SmartDTM*, which improves the quality of the user experience without violating the thermal requirement. The proposed SmartDTM technique is based on two key components, which form the main contributions of this paper: a user-perceived response-time predictor (urp) and a worst-case temperature predictor (wtp). At the start of each interactive session S, urp estimates I_S^{perc}, the length of the display-sensitive part of the session S, using a history of previous I_S^{perc} values of the display-sensitive part. Based on the estimated I_S^{perc} and the current temperature, wtp predicts the temperature T_S^{end} at the end of the display-sensitive part of S. In order to provide better user experience, even though the current temperature is higher than the trigger temperature, SmartDTM does not lower the maximum operating frequency if T_S^{end} does not exceed the critical temperature. By contrast, in SmartDTM employs an aggressive DVFS policy during the display-insensitive part. Since user experience is not affected in the display-insensitive part of the session S, the CPU temperature is quickly decreased to a safe level by aggressively scaling down the maximum operating frequency with no negative impact on user experience.

In order to evaluate our proposed technique, we implemented SmartDTM on the Android platform, version 4.4.2 (Kitkat), running on an Exynos 5410-based ODROID-XU+E board [6], which employs current and voltage sensors in order to measure the power consumption of the on-board components including the big CPU cluster, the little CPU cluster [7], the GPU, and the DRAM module. The experimental results show that the Android's default DTM policy interferes with user-perceived intervals, degrading their performance by an average of 17.3% and 44.4%, when the initial temperatures were 65°C and 70°C, respectively. On the other hand, SmartDTM can avoid many DTM decisions within user-perceived response time intervals by delaying them to user-oblivious response time intervals, thus mitigating the negative effect of DTM on the performance of user-perceived response time intervals. Our experimental results also show that when the initial temperatures were set to 65°C and 70°C, SmartDTM can improve the user-perceived performance by an average of 12.2% and 21.4%, respectively, over the Android's default DTM policy under the critical temperature of 85°C.

The rest of this paper is organized as follows. In Section II, we explain the key idea behind our proposed framework. In Section III, we present an overview of SmartDTM and illustrate how the user-perceived response time and the CPU temperature can be estimated under the SmartDTM framework. In Section IV, we report experimental results. In Section V, we review related work, and in Section VI, we conclude with a summary.

II. BASIC IDEA

Since smartphones are highly interaction-oriented devices, most usage scenarios on a smartphone consist of a sequence of *interactive sessions*, S_1, …, S_N, where each interactive session S_i is defined as an interval between

two consecutive user inputs. We can further divide the execution of an interactive session S_i into two subintervals, a *user-perceived response time interval* $I_{S_i}^{perc}$ and a *user-oblivious response time interval* $I_{S_i}^{oblv}$ [8]. $I_{S_i}^{perc}$ is the period from the beginning of the interactive session S_i initiated by a certain user input to the moment when the entirety of the required user-visible interface for the next interaction is fully displayed (In other words, the length of $I_{S_i}^{perc}$ is the user-perceived response time of S_i). $I_{S_i}^{oblv}$ can be considered as the user's think time about the next user interaction (The end of $I_{S_i}^{oblv}$ is determined by the time when the next user input is initiated for the next interaction).

While most user interactions on smartphones can be characterized as an alternating sequence of $I_{S_i}^{perc}$'s and $I_{S_i}^{oblv}$'s, few power/thermal management policies used for smartphones exploit these unique interaction patterns between user and a smartphone in their decision making procedures. For example, the Android's default DTM policy makes thermal control decisions in a reactive fashion depending only on the sensed temperature. As summarized in Fig. 1, the default DTM policy controls the CPU temperature by adjusting the maximum CPU operating frequency F_{op}^{max}, based on the current CPU temperature. That is, F_{op}^{max} can range from F_{lower}^{max} to F_{upper}^{max}, where F_{upper}^{max} is set to the maximum CPU frequency with which the CPU can operate. When the current CPU temperature T_{curr}, which is computed every 100 ms, reaches the trigger temperature T_{trig}, the default DTM policy lowers F_{op}^{max} by one level. By lowering F_{op}^{max}, the peak power consumption of the CPU can be reduced because an underlying Linux DVFS policy cannot choose a frequency higher than F_{op}^{max}, thus potentially leading to a decreased CPU temperature. This gradual reduction in F_{op}^{max} continues until T_{curr} falls below T_{trig}. Once T_{curr} is lowered below T_{trig}, F_{op}^{max} is gradually increased until F_{upper}^{max}. In order to ensure an acceptable level in user experience, the DTM policy sets the lower bound, F_{lower}^{max}, when lowering T_{curr}.

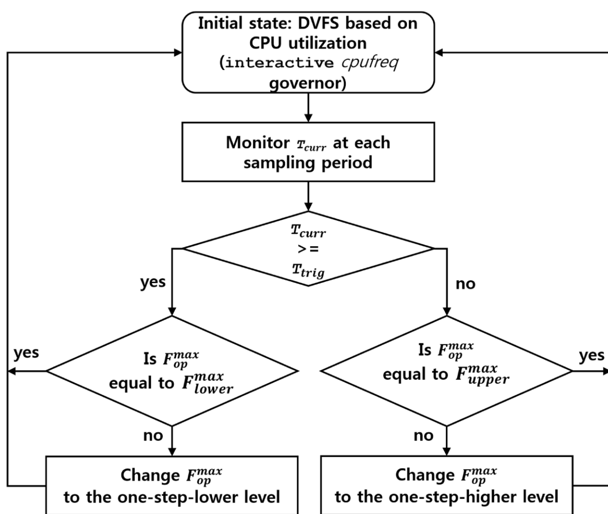
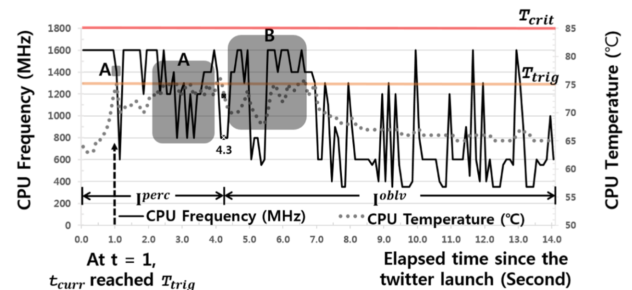
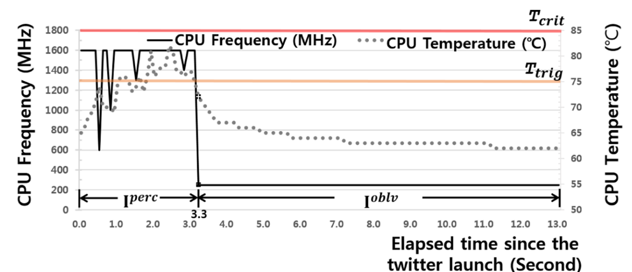


Fig. 1. An overview of the default DTM policy with the interactive *cpufreq* governor.

In order to explain the basic motivation of our proposed SmartDTM technique, we use an example interactive session S_L which launches the **twitter** app under the Android's default DTM policy with an initial temperature of 65°C. In this example, we assume that the critical temperature T_{crit} and T_{trig} were set to 85°C and 75°C, respectively. Fig. 2(a) shows how the CPU frequency and on-chip temperature change during 14.5 seconds after **twitter** is launched as measured in our evaluation board, ODROID-XU+E. The X-axis, the Y-axis on the left side, and the Y-axis on the right side represent the elapsed time, CPU frequency, and CPU temperature, respectively. At $t = 0$, the interactive session S_L is initiated by launching **twitter**. The user-visible interface is fully drawn at $t = 4.3$, and this is the user-perceived response time of S_L. However, since the CPU temperature reached T_{trig} after 1 second, F_{op}^{max} was reduced from 1,600 MHz to 1,400 MHz. The temperature then drops below 75°C, and after 1.8 seconds, the default DTM policy restores F_{op}^{max} to 1,600 MHz. Then the CPU temperature starts rising again, reaching 75°C after 2.4 seconds, when F_{op}^{max} is again reduced; but this time, the CPU temperature remains near T_{trig}, and so the default DTM policy keeps reducing F_{op}^{max} in several periods until it reaches 800 MHz. Since the big cluster only handles



(a)



(b)

Fig. 2. Changes in CPU frequency and temperature while **twitter** is launched under (a) the default DTM policy and (b) the oracle DTM policy, respectively.

the performance requirement higher than the CPU frequency of 800 MHz, the processing is switched from the big cluster to the littler cluster at this point. Although the default DTM policy can successfully control the CPU temperature below T_{trig}, on average, its decisions are not effective in two aspects. First, as shown in two areas (marked as A) in Fig. 2(a), the user-perceived launching time was significantly increased because the default DTM policy lowered F_{op}^{max} too aggressively. Second, the CPU temperature tends to drop very quickly in the I^{oblv} interval. If the DTM policy had known this thermal characteristic *a priori*, it could have avoided lowering F_{op}^{max} within the I^{perc} interval.

In order to understand the negative impact of the thermal control decisions under the default DTM policy on the user-perceived delay, we compared the default DTM policy with an oracle policy which had the complete future information. The oracle DTM policy does not lower F_{op}^{max} during the execution of I^{perc} because it knows in advance that doing so would not violate the thermal requirement (i.e., $T_{curr} < T_{crit}$) as shown in Fig. 2(b). Having this advanced knowledge on T_{curr}, the oracle DTM policy can achieve the user-perceived response time of 3.3 seconds, which is shorter than the default DTM policy by 4.3 seconds. In this example, similar to the result of the default DTM policy, T_{curr} reaches T_{trig} after 1 second. However, instead of lowering F_{op}^{max}, it F_{upper}^{max} remains at 1,600 MHz since T_{curr} would not exceed T_{crit} while executing the I^{perc} interval. As a result, the average and peak CPU temperatures of the I^{perc} interval, which are about 74°C and 82°C, respectively, are higher than those of the default DTM policy. However, considering that T_{crit} is 85°C, we can see that the thermal requirement is still satisfied in I^{perc}, even if such conservative DTM decision is applied. Moreover, under the default DTM policy, the CPU frequency often reaches 1,600 MHz, even during $I_{S_L}^{oblv}$, so that the CPU temperature reaches 76°C at t = 6.7, as shown in the area (marked as B) of Fig. 2(a).

On the other hand, as shown in Fig. 2(b), by employing F_{lower}^{max}, 250 MHz, during $I_{S_L}^{oblv}$, the oracle DTM policy could quickly reduce the CPU temperature as soon as $I_{S_L}^{perc}$ ends at $t = 3.3$.

Fig. 3 compares the user-perceived delay of the oracle DTM policy and the default DTM policy using 10 launching interactive sessions of various Android apps (For better accuracy, each app was launched 30 times and the averages of these 30 user-perceived delay values were shown in Fig. 3). In this experiment, all of the launching interactive sessions are initiated at 65°C, which is the temperature of Exynos 5410 processor [9] in our ODROID-XU+E board in a normal state. The X-axis and Y-axis denote various Android apps and their user-perceived delays, which are normalized to those of the results of the oracle DTM policy. In this scenario, the default DTM policy degrades the user-perceived performance by an average of 17.3% and up to 30.0% as compared to the oracle policy. If the initial temperature is 70°C, which is closer to T_{trig}, we can see (in Fig. 3(b)) that the user-perceived performance of the default policy is decreased by an average of 49.5% as compared to the oracle policy. Our experimental results strongly suggest that if we knew the future performance requirement of an interactive session, a much better DTM technique can be developed. In Section 3, we propose such an efficient DTM technique which depends on whether the current execution is in the user-perceived interval or in the user-oblivious interval of an interactive session.

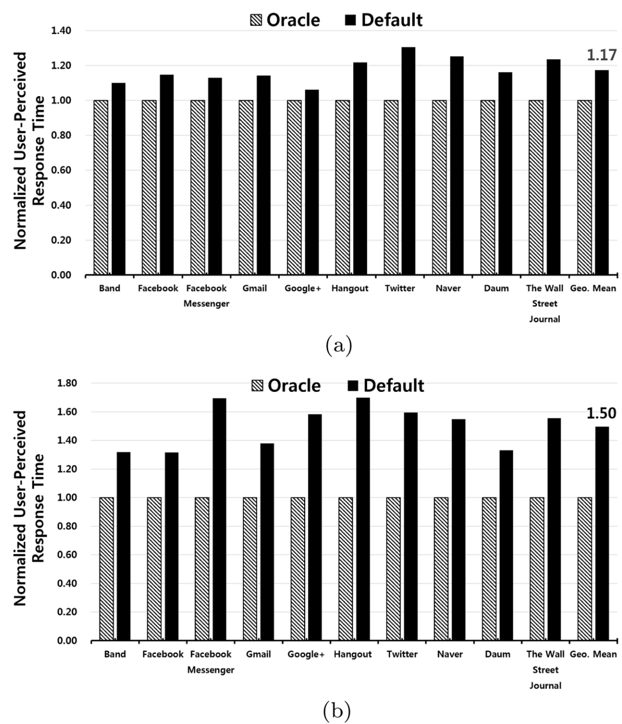


Fig. 3. Normalized user-perceived response time comparisons between the oracle and default DTM policies. (a) When each interactive session is initiated at 65°C. (b) When each interactive session is initiated at 70°C.

III. DESIGN AND IMPLEMENTATION OF SMARTDTM

A. Architectural Overview

The proposed SmartDTM consists of three main components: **ura** [8], platform-side and kernel-side modules. Fig. 4 shows an architectural overview of SmartDTM within the Android platform and kernel. In the proposed SmartDTM, **ura** is responsible for identifying the end of $I_{S_i}^{perc}$ during run time from the execution of S_i. In detail, for a given S_i, **ura** works as follows:

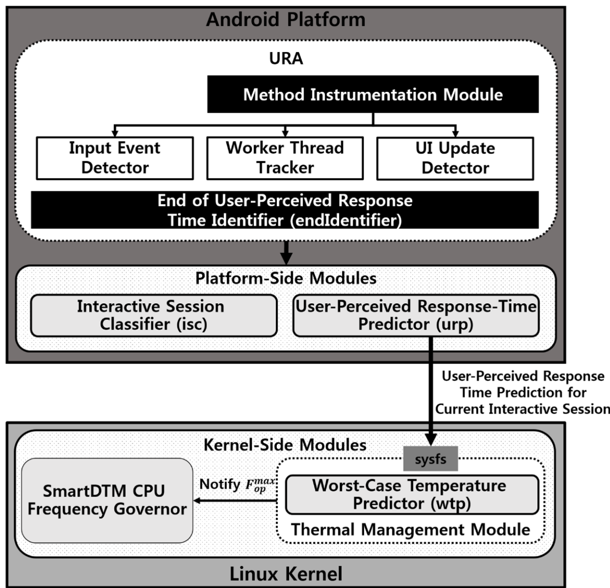


Fig. 4. An architectural overview of SmartDTM.

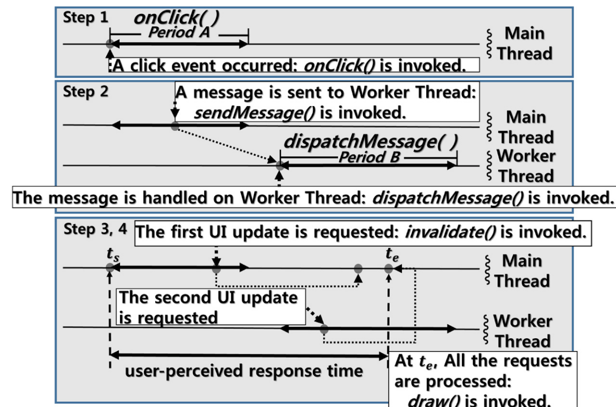


Fig. 5. An example of identifying the user-perceived response time.

Step 1. Detect the user input which initiates S_i.
Step 2. Keep track of any threads spawned by the user input.
Step 3. Detect requests to update the display which are related to handling the user input.
Step 4. $I_{S_i}^{perc}$ ends when all the display-update requests have been processed. If the next user input has already been initiated before all of the display-update requests are processed, the end of $I_{S_i}^{perc}$ can also be determined at this time.

Ura consists of two main modules, the method instrumentation module, which performs steps 1, 2, and 3, and a module which identifies the end of the user-perceived response time, called endIdentifier. The method instrumentation module consists of three submodules; the input event detector, the worker thread tracker, and the UI update detector. The input event detector captures events related to a particular user input. The worker thread tracker traces newly spawned threads, called worker threads, while processing the user input. In addition to the spawning of the worker threads, the worker thread tracker is also responsible for tracing all of the messages exchanged between the main thread and the worker threads during the user input processing. If there are any messages sent by the main thread to the worker threads that are already spawned before the user input, these messages are also traced by the worker thread tracker. The UI update detector tracks display-update requests made by serving the user input.

Fig. 5 illustrates how **ura** identifies the user-perceived response time using an example. In order to handle a particular interaction with UI components such as the Widget and View packages in the Android framework, it is required to implement callback methods in the event listener interface and register them to the UI components. And then, when a user interacts with a certain UI component, the registered callback method is invoked to process that user input. Corresponding to different types of user interactions, the Android SDK provides various callback methods. For example, user interactions such as a touch, a click, and a long-click are handled by *onTouch()*, *onClick()*, and *onLongClick()* methods, respectively. In the example, the callback method, *onClick()*, is invoked because the user clicks the user interface resource such as the BUTTON Widget. The input event detector first traces all the method invocations related to the callbacks for the user input, so as to identify the start t_s of the current interactive session S_i. In this example, the input event detector catches the *onClick()* invocation. The input event detector also tells the worker thread tracker and the UI update detector about all the method invocations issued by the execution of *onClick()*.

When an app is launched, a special thread, called a main thread, is created by the Android system. This is the only thread that can update the user-visible contents, while compute-intensive work is performed by worker threads, for better responsiveness. If a worker thread needs to update the user interface, it sends a request to the main thread. The Android SDK allows *Message* and *Runnable* objects to be exchanged between the main thread and a worker thread. By exploiting the information (which is provided by the input event detector) on the method invocations during the execution of the callbacks for the user input, the worker thread tracker traces newly spawned worker threads and all of the methods that they invoke. For example, as shown in step 2 of Fig. 5, the main thread wants to perform compute-intensive work via the worker thread, the main thread invokes *sendMessage()*, and the worker thread responds with *dispatchMessage()*. The worker thread tracker catches both the *sendMessage()*

and *dispatchMessage()* invocations. Then, in order to detect UI update requests created by the worker thread, all of the information regarding the method invocations during the execution of *dispatchMessage()* is fed to the UI update detector.

In order to recognize the changes in the user-visible contents, **ura** traces UI update requests issued by the user input and captures the moment at which the last request is handled. For example, at step 3 in Fig. 5, the *invalidate()* methods are invoked twice during the execution of both *onClick()* and *dispatchMessage()*. At these points, the UI update requests are posted to the main thread. The UI update detector catches the *invalidate()* invocations and watches the event queue for the UI update requests. Subsequently, when the main thread processes the last update request from its event queue, and invokes *draw()* to handle it (at t_e in step 4 of Fig. 5), endIdentifier determines that t_e *is* the end of I^{perc}. In this example, the user-perceived response time is estimated to be (t_e-t_s).

The interactive session classifier (isc) and the user-perceived response-time predictor (urp) are the platform-side modules in SmartDTM. When a user interacts with the UI components, isc creates a unique identifier at the start point of the interactive session. Once the unique identifier is created, all of the interactive sessions that have an identical identifier are grouped so that urp predicts the user-perceived response time for each group of the interactive sessions. When endIdentifier has determined that $I_{S_i}^{perc}$ has ended, it informs urp, which adds the user-perceived response time to the total associated with the corresponding session identifier. In order to predict the user-perceived response time, urp uses a statistical analysis of the accumulated user-perceived response time information. On the kernel side of SmartDTM, there are two modules, the worst-case temperature predictor (wtp) and the SmartDTM CPU frequency governor. wtp in the thermal management module, which is responsible for applying the DTM decisions, estimates the time at which the CPU temperature will reach T_{crit} by performing the worst-case temperature estimation whenever T_{curr} reaches T_{trig}. If T_{curr} is not expected to exceed T_{crit} during the execution of $I_{S_i}^{perc}$, the module does not change F_{op}^{max} in order to improve the user-perceived performance. Otherwise, as is the case with the default DTM policy, F_{op}^{max} is reduced by the thermal management module. When F_{op}^{max} is changed or endIdentifier detects the end of $I_{S_i}^{perc}$, the thermal management module notifies it to the SmartDTM CPU frequency governor.

B. Predicting User-Perceived Response Time

In order to make a unique identifier for classifying the interactive sessions, isc takes advantage of the Android's View hierarchy system. Fig. 6 is a snapshot of the **Twitter** GUI (top) and the corresponding View hierarchy (bottom).

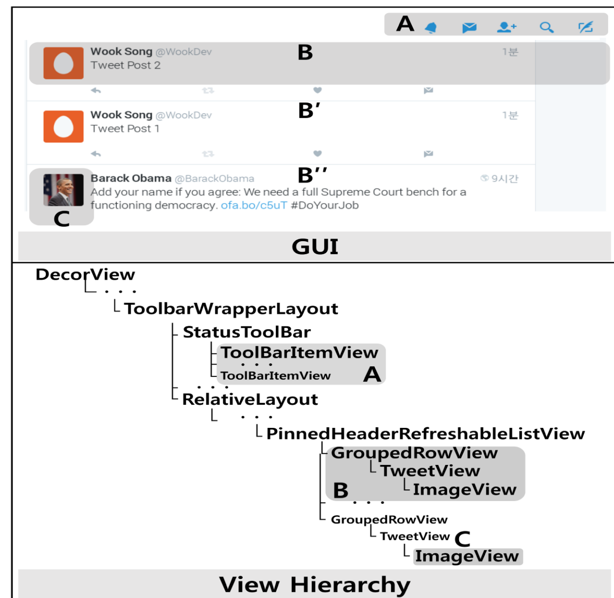


Fig. 6. A GUI example of the **twitter** app and its View hierarchy.

It can be seen that the GUI is as a tree structure, the terminal nodes of which correspond to visible UI components. For example, an image button on the toolbar, a tweet post in the timeline, and a profile image correspond to a ToolBarItemView node, a GroupedRowView node, and an ImageView node, respectively (marked as A, B, and C, respectively, at the top and bottom of Fig. 6). An interactive session can therefore be identified by the unique concatenation of the names of the terminal node and its parents back to the root of the tree corresponding to the GUI component used to start the session. In further detail, whenever the input event detector of **ura** captures the events related to the user input, isc traverses the View hierarchy from the terminal node, which handles the user input, to the root, DecorView, and makes a long string, which is the concatenation of the names of all of the visited nodes. Interactive sessions related to UI components with the same identifier are put into the same group. In this example, the interactive sessions initiated by selection of tweet (marked as B, B' and B" at the top of Fig. 6) will be grouped together.

The 30 most recent user-perceived response times are stored by urp, and the mean and standard deviation are calculated. Assuming that the user-perceived response times for a specific interactive session group are normally distributed, the particular percentile (e.g., 95.05% in the current implementation) of the distribution can be obtained using the probit function [10], then used as the expected user-perceived response time of the interactive session.

C. Worst-Case Temperature Estimation Model

Since CPU temperature is strongly related to power

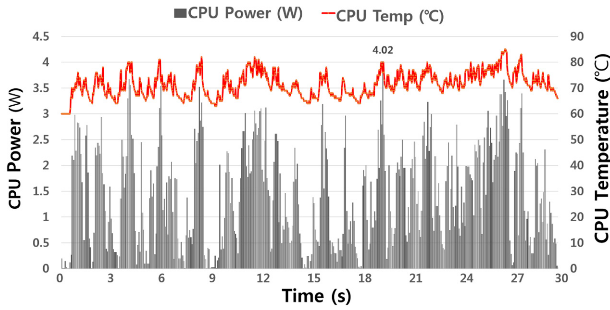


Fig. 7. Changes in the CPU power consumption and temperature under heavy usage scenarios.

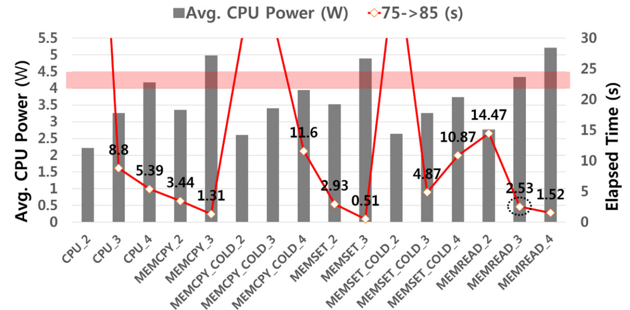


Fig. 8. Differences of the average CPU power consumption and the elapsed times corresponding to the CPU temperature changes from 75°C to 85°C between the various workload characteristics.

consumption, we built a worst-case temperature estimation model by measuring the power consumption during heavy usage scenarios. Fig. 7 shows how the CPU power consumption and temperature of our Exynos 5410-based ODROID-XU+E board change under heavy usage scenarios. The workload involves 10 sequential launches of the apps profiled in Fig. 3. The X-axis, the Y-axis on the left side, and the Y-axis on the right side represent the elapsed time, CPU power consumption, and CPU temperature, respectively. We can see that the CPU power consumption varied dramatically with workload, up to 4.02 W. Based on this observation, we made the conservative assumption that the CPU power consumption of our ODROID-XU+E board would range from 4.00 W to 4.50 W under the heavy usage.

Although CPU temperature is significantly dependent on power consumption, the workload characteristic is also an important factor affecting temperature fluctuations. To observe the changes in the CPU temperature caused by varying a workload characteristic, we measured the average CPU power consumption and the time that the CPU temperature took to change from T_{trig} (i.e., 75°C) to T_{crit} (i.e., 85°C) for each workload in the *micro_bench* binary, from the Android Open Source Project. The characteristics of these workloads are summarized in Table 1 and the results are shown in Fig. 8. The X-axis, the Y-axis on the left side, and the Y-axis on the right side denote various workloads, their average CPU power consumption, and the elapsed times, respectively. Since the CPU power consumption for two workloads, **memcpy_4** and **memset_4**, exceeded 5.50 W, which is extremely high for the target system, we excluded these workloads. The **cpu_4** and **memread_3** workloads draw 4.18 W and 4.34 W, respectively, which are in the 4.00 - 4.50 range, but their thermal characteristics differ substantially. The CPU temperature rose from 75°C to 85°C within 5.9 seconds during the execution of **cpu_4**, but with in 2.53 seconds during the execution of **memread_3**. We therefore chose **memread_3** to build a method for estimating the worst-case temperature $T(t)$ using linear-log regression:

$$T(t) = 5.947 \cdot ln(t) + 64.386 \qquad (1)$$

where t is elapsed time in units of 100 ms, and where $T(0)$ is the initial temperature of Exynos 5410 processor of our ODROID-XU+E board in a normal state.

D. Thermal Management Module

Fig. 9(a) illustrates how SmartDTM can avoid reducing F_{op}^{max} by predicting the end of $I_{S_i}^{perc}$ and t_{crit}, which is the time when the current temperature will reach T_{crit}. At t_{init}, which is the beginning of S_i initiated by a certain user input, SmartDTM estimates t_{end}, which is when $I_{S_i}^{perc}$ ends. Each time the CPU temperature is measured, t_{curr}, is compared with t_{end} in order to estimate the time remaining

Table 1. Summary of the workloads in the *micro_bench* binary

Benchmark name	Description
CPU_*n*	Multi-threaded CPU-intensive workload using *n* threads
MEMCPY_*n*	Multi-threaded cache-intensive workload, which repeatedly calls the *memcpy* function using *n* threads
MEMCPY COLD_*n*	Multi-threaded memory-intensive workload, which repeatedly calls the *memcpy* function using *n* threads
MEMSET_*n*	Multi-threaded cache-intensive workload, which repeatedly calls the *memset* function using *n* threads
MEMSET COLD_*n*	Multi-threaded memory-intensive workload, which repeatedly calls the *memset* function using *n* threads
MEMREAD_*n*	Multi-threaded cache-intensive workload, which repeatedly reads values from an array using *n* threads
MEMREAD COLD_*n*	Multi-threaded memory-intensive workload, which repeatedly reads values from an array using *n* threads

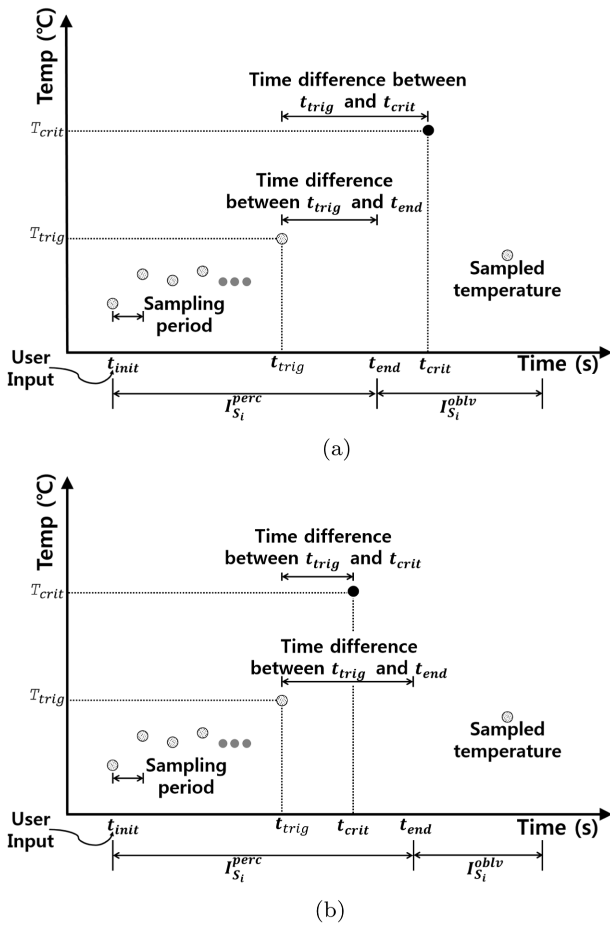


Fig. 9. Two cases of how SmartDTM makes the DTM decisions based on the estimated values of t_{end} and t_{crit}. (a) F^{max}_{op} is not reduced during the execution of $I^{perc}_{S_i}$. (b) F^{max}_{op} is reduced during the execution of $I^{perc}_{S_i}$.

in $I^{perc}_{S_i}$. If T_{curr} reaches T_{trig} at t_{trig} during the execution of $I^{perc}_{S_i}$, SmartDTM does not immediately reduce F^{max}_{op}. Instead, it predicts t_{crit} using its worst-case temperature estimation model. If t_{crit} is later than t_{end}, F^{max}_{op} is not reduced to ensure the quality of user experience during the execution of $I^{perc}_{S_i}$. Furthermore, when $I^{oblv}_{S_i}$ starts, the SmartDTM CPU frequency governor employs the lowest CPU frequency, which rapidly reduces the CPU temperature while executing $I^{oblv}_{S_i}$.

Algorithm 1 describes how the `SmartDTM` *cpufreq* governor decides the CPU frequency. As is the case with other Linux *cpufreq* governors, the CPU frequency is updated at each sampling period (e.g., 20 ms). The `SmartDTM` governor relies on endIdentifier of **ura** in order to keep track of whether the current execution is in I^{perc} or I^{oblv}, as described on the first line in Algorithm 1. Whenever a new interactive session S_i is started, the current execution interval type is set to I^{perc}. By contrast, when endIdentifier detects the end of I^{perc}, it is changed to I^{oblv}. By exploiting this information, the `SmartDTM` governor employs the minimum operating CPU frequency, F^{max}_{lower}, while executing I^{oblv}. Otherwise, when the current execution is in I^{perc}, the CPU frequency is decided by the `interactive` *cpufreq* governor [11], which is the default governor in most kernels for the Android Open Source Project. In particular, for higher responsiveness, when the CPU load exceeds the predefined upper threshold (e.g., 99, L_{go_high} in Algorithm 1), the `interactive` *cpufreq* governor quickly switches to the maximum operating CPU frequency, F^{max}_{upper}. On the other hand, if the CPU is less loaded, F^{max}_{upper} is multiplied by the percentage of the current CPU load in order to determine the CPU frequency for the next sampling period.

ALGORITHM 1: Pseudo code for the `SmartDTM` CPU frequency governor.

```
1: function compute_next_freq(cur_exec_interval)
2: {
3:     if cur_exec_interval is I^perc
4:         return compute_next_freq_interactive( );
5:     else if cur_exec_interval is I^oblv
6:         ▷ F^max_lower is the minimum operating CPU frequency.
7:         return F^max_lower;
8:     endIf
9: }

10: function compute_next_freq_interactive( )
11: {
12: ▷ The interactive cpufreq governor starts from this point.
13:     cur_load = get_cpu_load()
14:     if cur_load is greater than L_go_high
15:         ▷ F^max_upper is the maximum operating CPU frequency.
16:         return F^max_upper;
17:     else
18:         next_freq = F^max_upper × cur_load ÷ 100;
19:         return next_freq;
20: }

21: cur_exec_interval = get_cur_exec_interval();
22: next_freq = compute_next_freq(cur_exec_interval);
23: set_freq(next_freq);
```

Sometimes, in order to avoid a thermal violation, SmartDTM has to reduce the CPU frequency during the execution of $I^{perc}_{S_i}$, as shown in Fig. 9(b). In this example, at t_{trig}, SmartDTM predicts that t_{crit} will come before t_{end}, which means that if we do not reduce the CPU frequency, then T_{curr} will soar above T_{crit} at t_{crit}. Therefore, in this case, the default DTM policy is applied in order to avoid the thermal violation.

IV. EXPERIMENTAL RESULTS

A. Experimental Environment

We implemented the SmartDTM technique on the

Table 2. Scenario descriptions of 10 apps used in the experiments

App name (category)	Interactive session ID	Interactive session description	User-perceived response time with oracle (s)
Band (social networking)	S1	Launching	1.98
	S2	Viewing an article	0.53
Facebook (social networking)	S3	Launching	2.01
	S4	Clicking the search button	0.46
Facebook Messenger (messenger)	S5	Launching	0.59
	S6	Changing tabs	0.24
Gmail (mail)	S7	Launching	3.48
	S8	Reading a mail	3.42
Google+ (social networking)	S9	Launching	1.38
	S10	Viewing today's recommended featured collection	0.78
Hangout (messenger)	S11	Launching	2.22
	S12	Opening a chat session	0.35
Twitter (social networking)	S13	Launching	3.35
	S14	Viewing a tweet post in the timeline	0.23
Naver (web portal)	S15	Launching	8.89
Daum (web portal)	S16	Launching	3.27
The Wall Street Journal (news)	S17	Launching	1.28

Exynos 5410-based ODROID-XU+E board running Android 4.4.2 (Kitkat). lsc and urp were implemented in the Android platform. The thermal management module, which makes the DTM decisions based on the prediction of the user-perceived response time, was added to the Linux kernel, version 3.4.5. The predicted user-perceived response time is exported to the thermal management module using a Linux kernel's *sysfs* file [12]. We experimented with 10 apps under different usage scenarios. Since our isc exploits Android's View hierarchy system, it cannot support certain apps based on web pages; therefore, only launch scenarios are used for those apps. The scenarios for the other apps consist of two consecutive interactive sessions. Table 2 summarizes the apps and their usage scenarios.

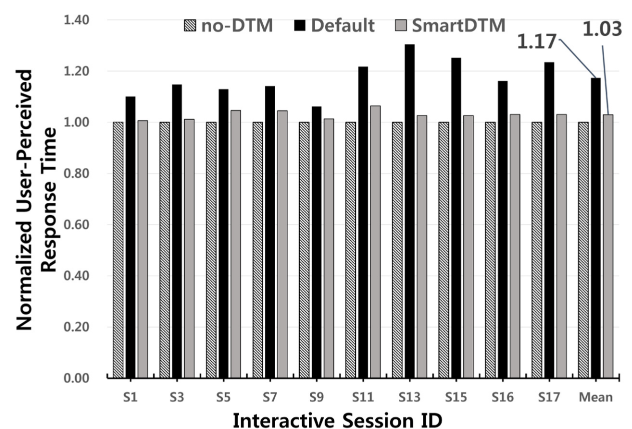


Fig. 10. A comparison of normalized user-perceived response times for 10 launching interactive sessions when the initial temperature was 65°C.

B. Performance Evaluation

Fig. 10 shows the effect of SmartDTM on the user-perceived response times for 10 launching interactive sessions. In this experiment, at the start of each session, the CPU temperature is 65°C, which is considered normal. Since only launching interactive sessions raise the CPU temperature above 75°C from the initial 65°C, these sessions are selected for evaluation. On these interactive sessions, SmartDTM improves the performance of user-perceived response time intervals by an average of 12.2% compared to the default DTM technique. For S13 (**twitter**), the proposed SmartDTM achieves the maximum improvement in the user-perceived response time of 21.3%. For 7 out of 10 scenarios, the drop in user-perceived performance using SmartDTM was less than 3.0%. Even in the worst-case, S11 (**hangout**), the user-perceived response time was 6.2% more than it is with the oracle policy, while the default DTM policy increases the user-perceived response time by 21.8%.

As the initial temperature increases, more reductions in

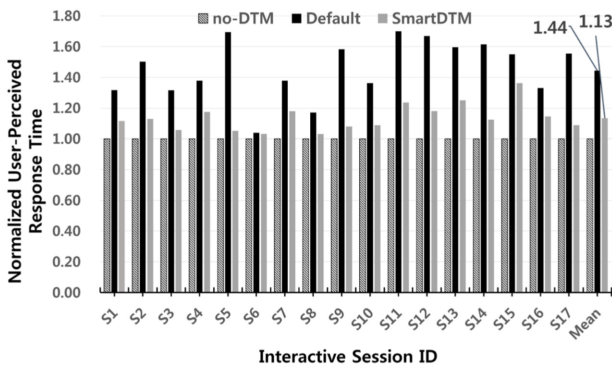


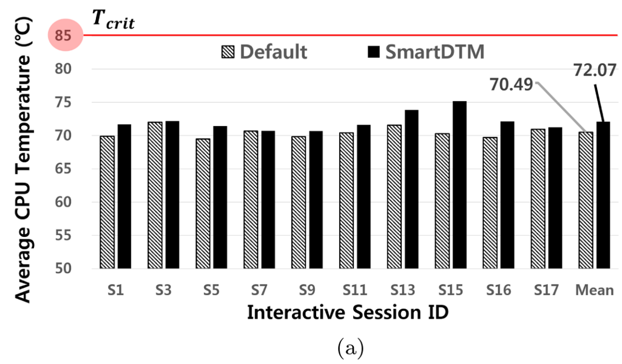
Fig. 11. A comparison of normalized user-perceived response times for 17 interactive sessions when the initial temperature was 70°C.

CPU frequency are required in order to avoid thermal violations during the execution of I_S^{perc}, which reduces user-perceived performance. Fig. 11 shows the normalized user-perceived response times for the 17 interactive sessions in Table 2, when the initial temperature is 70°C. The user-perceived performance drops whether the default or SmartDTM is used. However, the reduction with SmartDTM is 21.4% less on average than the default DTM technique. Moreover, for 9 of 17 interactive sessions, the default DTM policy increases user-perceived response times by at least 50.0%, while SmartDTM only increases these delays by more than 23.6% for S11, S13, and S15; and for the other scenarios the delay is 18.1% or less.

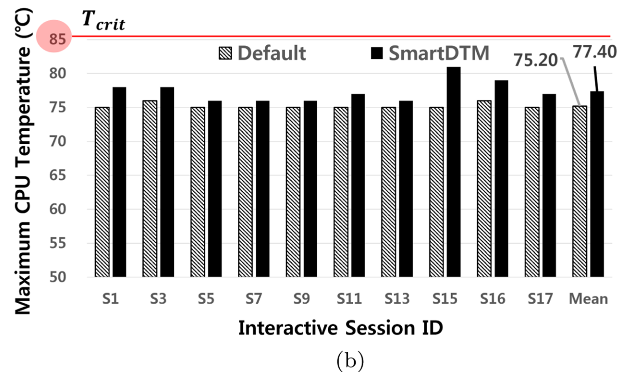
C. Temperature Evaluation

The average and maximum CPU temperatures under the default and SmartDTM policies during the execution of I_S^{perc} are compared in Fig. 12. The X-axis and Y-axis denote the interactive sessions and their average or maximum CPU temperatures while executing the user-perceived response time intervals, respectively. When the initial temperature is 65°C, SmartDTM raises the average CPU temperature by 1.70°C more than the default policy. The maximum temperatures observed during I_S^{perc} under the SmartDTM could be acceptable, if the critical temperature is 85°C. As shown in Fig. 12(b), for S13 (a launching interactive session of **twitter**, the highest CPU temperature observed during I_S^{perc} was 82°C.

If the initial temperature is close to the trigger temperature, higher CPU temperatures are observed. Nevertheless, SmartDTM can avoid thermal violations using the worst-case temperature estimation. In order to evaluate the effectiveness of the proposed SmartDTM on the thermal management, Fig. 13 shows the average and maximum CPU temperatures during I_S^{perc} for the default and SmartDTM policies when the initial temperature is 70°C. The X-axis and Y-axis are the same as those shown



(a)



(b)

Fig. 12. Variations in the average and maximum CPU temperatures under the SmartDTM and default policies when the initial temperature is 65°C. (a) A comparison of the average CPU temperature during the execution of the user-perceived response time interval. (b) A comparison of the maximum CPU temperature during the execution of the user-perceived response time interval.

in Fig. 12. SmartDTM produces average CPU temperatures which are 2.09°C higher than those for the default policy. The maximum CPU temperatures shown in Fig. 13(b) are much higher, but SmartDTM still avoids thermal violations.

In order to evaluate the effect of the SmartDTM *cpufreq* governor on the reduction in CPU temperature during the user-oblivious response time interval, we measured the length of time between the end of the user-perceived response time and the time at which the temperature drops to 65°C, with the results shown in Fig. 14. The X-axis and Y-axis denote the interactive sessions and the elapsed times of these from the time when the execution of the user-perceived response time interval ends to the time when the temperature drops to 65°C, respectively. When the initial temperature is 65°C, the SmartDTM technique reduces this interval by 58.3%. When tasks are executed during the user-oblivious response time interval, the default DTM policy fails to significantly reduce the CPU temperature during that interval. As shown in Fig. 14(b), SmartDTM is also effective in reducing the CPU temperature during the user-oblivious response time interval when the initial temperature is 70°C. In this experiment, SmartDTM requires, on average, 45.2% less

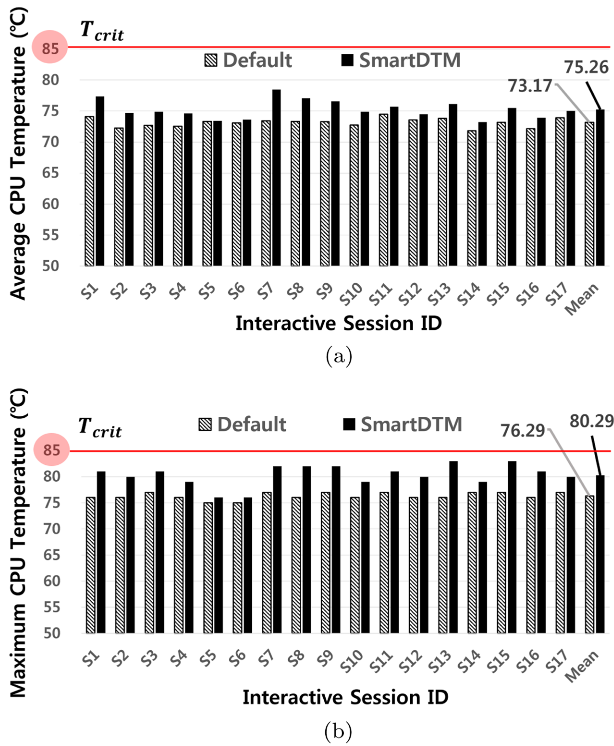


Fig. 13. Variations in the average and maximum CPU temperatures under the SmartDTM and default policies when the initial temperature is 70°C. (a) A comparison of the average CPU temperature during the execution of the user-perceived response time interval. (b) A comparison of the maximum CPU temperature during the execution of the user-perceived response time interval.

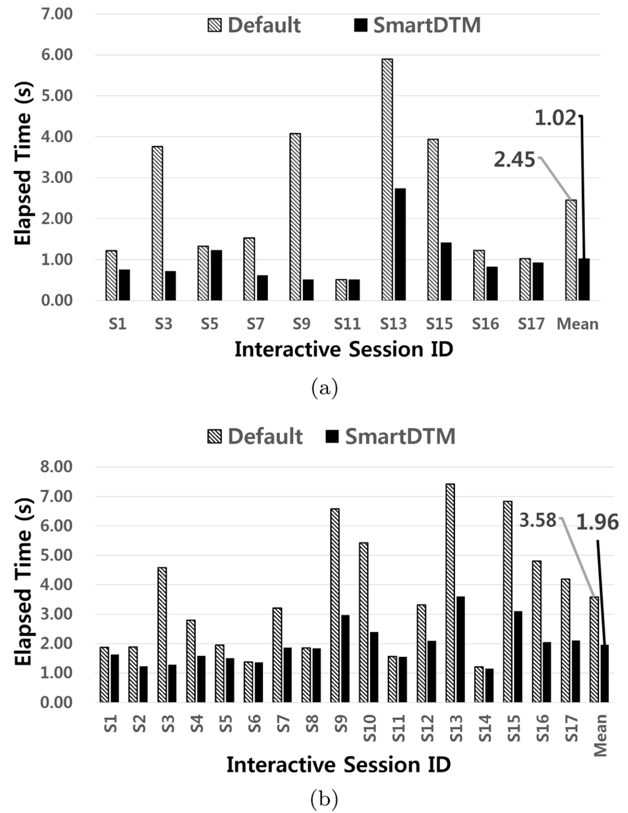


Fig. 14. Distributions of the elapsed times from the end of the user-perceived response time interval ends to the time when the temperature drops to 65°C. (a) The initial temperature was 65°C. (b) The initial temperature was 70°C.

time to bring the CPU temperature back to 65°C during that interval. For S13, 7.42 seconds are required to decrease the CPU temperature to 65°C, while the proposed SmartDTM only requires 3.60 seconds. Moreover, in 10 out of 17 interactive sessions, SmartDTM reduced the CPU temperature to 65°C within 2 seconds.

V. RELATED WORK

Since the highly elevated on-chip temperature negatively affects the reliability and energy consumption of the system, many research groups have investigated thermal management in various levels [13, 14]. In order to control the on-chip temperature from the mechanical side, several thermal management techniques have been proposed. Air cooling techniques based on fan-cooled heat sinks [15] and liquid cooling techniques [16-18] are representative techniques. These techniques are useful in solving thermal problems without any performance degradation, but it is difficult to adopt these techniques to smartphones because of the limited space of the form factor of the smartphone and the extra power consumption caused by these techniques. Therefore, many researchers have also focused on software-based thermal management techniques.

Hanson et al. [19] reported that DVFS has an immediate influence on the on-chip temperature. Based on their analysis, Hanson et al. suggest that DVFS could be a viable mechanism for software-based thermal management. Liu et al. [20] and Hanumaiah and Vrudhula [21] have also focused on reducing the on-chip temperature using DVFS. Liu et al. [20] have proposed design-time thermal optimization techniques for real-time embedded systems. They showed that their techniques can effectively prevent run-time thermal emergencies while optimizing the cooling cost and performance by selecting an optimal voltage and frequency for each task at design time. Hanumaiah and Vrudhula [21] take a similar approach to that of Liu et al. [20]. In order to satisfy timing and temperature constraints in hard real-time systems, they have proposed an analytical thermal model suitable for optimal DVFS and task assignment/allocation. However, since these techniques are only focused on controlling on-chip temperature in the presence of real-time constraints, it is difficult to apply them to general purpose systems.

For general purpose systems, instead of the design

time thermal management techniques, many groups have proposed on-line thermal management techniques [5, 22, 23]. For example, Brooks and Martonosi [5] have explored policies and mechanisms for implementing DTM in high-performance computing systems. Skadron [22] took a hybrid approach that combines fetch gating and DVFS in order to minimize the performance degradation. Lee et al. [23] have proposed a predictive temperature-aware DVFS scheme using hardware performance counters. By taking advantage of simple regression analysis, which uses the performance counters, they can detect localized thermal problems that usually go undetected because of the limited number of on-chip thermal sensors. While these techniques are effective in reducing the on-chip temperature during run time, they are limited in that these DVFS-based DTM schemes inevitably degrade the system performance when the CPU frequency is reduced so as to avoid the thermal violation.

In order to overcome the limitation of the existing DTM techniques, Kim et al. [24] have proposed a temperature-aware DVFS scheme, which can improve both the power efficiency and performance of the system. Their scheme provides two optimization options: the power optimizing option that saves energy and the performance optimizing option that enhances the performance. In particular, when the performance optimizing option is activated, the CPU frequency is increased by 10 MHz at every sampling period even if the current temperature remains above the trigger temperature. On the other hand, when the current temperature finally reaches the predefined thermal threshold, the CPU frequency is scaled down for the purpose of system reliability. Our work is fundamentally different from their technique in that we take advantage of the user-perceived response time as a main guideline for making the DTM decisions.

VI. CONCLUSIONS

We have presented a novel DTM scheme for smartphones, called SmartDTM. Based on an estimation of a user-perceived response time for a given interactive session and a prediction of the worst-case temperature at the end of the user-perceived response time interval, the proposed SmartDTM technique can improve the quality of the user experience without violating thermal requirements. In order to mitigate the negative effect of DTM on the user-perceived performance, even though the current temperature is higher than the trigger temperature, SmartDTM avoids DTM decisions within the user-perceived response time interval if the worst-case temperature during the execution of the user-perceived response time interval does not exceed the critical temperature. On the other hand, SmartDTM employs an aggressive DVFS policy while executing the user-oblivious response time interval where the system performance level does not affect the user-perceived performance so that the CPU temperature is quickly decreased to a safe level. The experimental results show that when the initial temperatures were set to 65°C and 70°C, SmartDTM can improve the user-perceived performance by an average of 12.2% and 21.4%, respectively, over the Android's default DTM policy under the critical temperature of 85°C. The SmartDTM scheme could be further extended in several directions: For example, we can extend our proposed scheme to control the temperatures of system components such as GPU and the memory subsystem, which also make significant contributions to system temperature. By exploiting the user-perceived response time analysis, we can make smarter DTM decisions in the system-wide level.

ACKNOWLEDGMENTS

This study was based on Core Component Technology Development for HMA-based System Optimization (July 1, 2012 - June 30, 2014) funded by Samsung Electronics Co., Ltd. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT) (NRF-2018R1A2B6006878). The ICT at Seoul National University provided research facilities for this study.

REFERENCES

1. Samsung, "Galaxy S7 edge and Galaxy S7," 2016, http://www.samsung.com/us/explore/galaxy-s7-features-and-specs/#specs.
2. J. Henkel, S. Pagani, H. Khdr, F. Kriebel, S. Rehman, and M. Shafique, "Towards performance and reliability-efficient computing in the dark silicon era," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, Dresden, Germany, 2016, pp. 1-6.
3. K. S. Kim, M. H. Won, J. W. Kim, and B. J. Back, "Heat pipe cooling technology for desktop PC CPU," *Applied Thermal Engineering*, vol. 23, no. 9, pp. 1137-1144, 2003.
4. C. Nelson and J. Galloway, "Package thermal challenges due to changing mobile system form factors," in *Proceedings of 2018 34th Thermal Measurement, Modeling & Management Symposium (SEMI-THERM)*, San Jose, CA, 2018, pp. 98-106.
5. D. Brooks and M. Martonosi, "Dynamic thermal management for high-performance microprocessors," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, Monterrey, Mexico, 2001, pp. 171-182.
6. Hardkernel, "ODROID-XU+E," https://www.hardkernel.com/?s=ODROID-XU%2BE.
7. P. Greenhalgh, "big.LITTLE Technology: The Future of Mobile," ARM, White paper, 2013, https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf
8. W. Song, N. Sung, B. G. Chun, and J. Kim, "Reducing energy

consumption of smartphones using user-perceived response time analysis," in *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, Santa Barbara, CA, 2014.
9. Samsung, "Exynos 5 Octa (5410)," 2013, https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-5-octa-5410/.
10. S. L. Zeger, K. Y. Liang, and P. S. Albert, "Models for longitudinal data: a generalized estimating equation approach," *Biometrics*, vol. 44, no. 4, pp. 1049-1060, 1988.
11. M. Chan, "Interactive CPU frequency governor," https://android.googlesource.com/kernel/common/+/android-3.4/drivers/cpufreq/cpufreqinteractive. c, 2010.
12. P. Mochel, "The sysfs filesystem," 2005, https://mirrors.edge.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf.
13. J. Kong, S. W. Chung, and K. Skadron, "Recent thermal management techniques for microprocessors," *ACM Computing Surveys*, vol. 44, no. 3, article no. 13, 2012.
14. Y. G. Kim, J. Kong, and S. W. Chung, "A survey on recent OS-level energy management techniques for mobile processing units," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2388-2401, 2018.
15. T. Nguyen, M. Mochizuki, K. Mashiko, Y. Saito, and I. Sauciuc, "Use of heat pipe/heat sink for thermal management of high performance CPUS," in *Proceedings of 16th Annual IEEE Semiconductor Thermal Measurement and Management Symposium*, San Jose, CA, 2000, pp. 76-79.
16. J. M. Koo, S. Im, L. Jiang, and K. E. Goodson, "Integrated microchannel cooling for three-dimensional electronic circuit architectures," *Journal of Heat Transfer*, vol. 127, no. 1, pp. 49-58, 2005.
17. T. Brunschwiler, B. Michel, H. Rothuizen, U. Kloter, B. Wunderle, H. Oppermann, and H. Reichl, "Forced convective interlayer cooling in vertically integrated packages," in *Proceedings of 11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*, Orlando, FL, 2008, pp. 1114-1125.
18. H. B. Jang, I. Yoon, C. H. Kim, S. Shin, and S. W. Chung, "The impact of liquid cooling on 3D multi-core processors," in *Proceedings of IEEE International Conference on Computer Design*, Lake Tahoe, CA, 2009, pp. 472-478.
19. H. Hanson, S. W. Keckler, S. Ghiasi, K. Rajamani, F. Rawson, and J. Rubio, "Thermal response to DVFS: analysis with an Intel Pentium M," in *Proceedings of ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, Portland, OR, 2007, pp. 219-224.
20. Y. Liu, H. Yang, R. P. Dick, H. Wang, and L. Shang, "Thermal vs energy optimization for DVFS-enabled processors in embedded systems," in *Proceedings of the 8th International Symposium on Quality Electronic Design*, San Jose, CA, 2007, pp. 204-209.
21. V. Hanumaiah and S. Vrudhula, "Temperature-aware DVFS for hard real-time applications on multicore processors," *IEEE Transactions on Computers*, vol. 61, no. 10, pp. 1484-1494, 2012.
22. K. Skadron, "Hybrid architectural dynamic thermal management," in *Proceedings of the Conference on Design, Automation and Test in Europe*, Paris, France, 2004.
23. J. S. Lee, K. Skadron, and S. W. Chung, "Predictive temperature-aware DVFS," *IEEE Transactions on Computers*, vol. 59, no. 1, pp. 127-133, 2010.
24. J. M. Kim, Y. G. Kim, and S. W. Chung, "Stabilizing CPU frequency and voltage for temperature-aware DVFS in mobile devices," *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 286-292, 2015.

Wook Song

Wook Song is a staff engineer at Samsung Electronics, where he has been working on software platform development for on-device AI projects. He received his PhD and MS degrees in computer science and engineering from Seoul National University in 2016 and 2009, respectively, and earned his BE degree in computer engineering from Sungkyunkwan University in 2007. Towards his Ph.D., he worked on developing user-centric optimization techniques for modern mobile operating systems. He is a third-place winner of the 2011 Android Competition in Embedded Systems Week for his work on personalized optimization, and also a recipient of the 2017 Samsung Best Paper Bronze Award for his research on CPU thermal management using user-perceived response time analysis.

Jihong Kim

Jihong Kim is a Professor in the department of Computer Science & Engineering, Seoul National University. Before joining Seoul National University, he was a Member of Technical Staff in the DSPS R&D Center of Texas Instruments in Dallas, Texas, USA. Jihong Kim received his BS in computer science and statistics from Seoul National University in 1986, and MS and PhD degrees in computer science and engineering from the University of Washington in 1988 and 1995, respectively. His research interests include low-power systems, NAND flash-based storage systems, computer architecture and mobile computing.