ThermTap: An Online Power Analyzer and Thermal Simulator for Android Devices

Mohammad Javad Dousti, Majid Ghasemi-Gol, Mahdi Nazemi, and Massoud Pedram
Department of Electrical Engineering, University of Southern California, Los Angeles, CA, USA
{dousti,ghasemig,mnazemi,pedram}@usc.edu

***Abstract*—This paper introduces *ThermTap*, which enables system and software developers to monitor the power consumption and temperature of various hardware components in an Android device as a function of running applications and processes. *ThermTap* comprises of a power analyzer, called *PowerTap*, and an online thermal simulator, called *Therminator 2*. With accurate power macro-models, *PowerTap* collates activity profiles of major components of a portable device from the OS kernel device drivers in an event-driven manner to generate power traces. In turn, *Therminator 2* reads these traces and, using a compact thermal model of the device, generates various temperature maps including those for the device components and device skin. Fast thermal simulation techniques enable *Therminator 2* to be executed in realtime. With precise per-process and per-application temperature maps that ThermTap produces, it enables software and system developers to find thermal bugs in their software. A case study is presented on identifying a thermal bug in the software running on an Android device.**

I. Introduction

In the past five years, portable devices have become an integral and indispensable part of our life. In 2011, smartphones and tablets have surpassed the sale of PCs and become the dominant part of consumer electronics [1]. Main technological barriers against advancement and further penetration of these devices into our everyday life include relatively high power consumption and their small-form factor, which limits the amount of energy storage that can be integrated into these devices. Moreover, these devices have a very strict thermal envelope. A mobile device has two types of thermal constraints. The first one (similar to PCs) is the *die temperature* constraint. This constraint makes sure that the *application processor* (AP) which contains CPU, GPU, and some other components runs below a certain temperature all the time. The second constraint is called the *skin temperature* constraint which is unique to mobile devices [2], [3]. It ensures that the temperature at the surface (or skin) of the device remains low to avoid any user discomfort or skin burn.

In order to ensure that a device adheres with the aforementioned power and thermal constraints, precise modelings and measurements are required during the design and prototyping. Due to the very limited resources available on portable devices, their software should be designed in power- and thermal-aware manners. One simple solution is to embed thermal sensors as well as current sensors into every major component of the device in order to measure their temperature and power consumption. The main drawback of this solution is that in a complex multithreading and multitasking environment, sensors are blind to which applications/processes affect the temperature and cause the temperature rise. Furthermore, sensors do not provide temperature maps for their respective component and thus, cannot determine workload-dependent hot spots. Besides, deployment of accurate temperature sensors increases the cost of the device and hence is not desirable. Moreover, use of external sensors might not be practical and indeed unaffordable for many software or system developers [4].

In this paper, we introduce *ThermTap*, which enables system and software developers to monitor the power consumption and temperature of various hardware components in a portable device as a function of running applications and processes. ThermTap comprises of two important parts: a power analyzer called *PowerTap* and an online thermal simulator called *Therminator 2*.

PowerTap collates the operating state and activity information of various system components in the device from the operating system (OS) device driver layer. This is done in an event-driven manner as opposed to a sampling-based method, which has a high overhead and tends to be slow [5], [6]. We use *SystemTap* which is an industry-standard kernel debugging and performance monitoring tool [7]. SystemTap allows dynamically adding probes inside a running kernel without any destructive side-effect. In this paper, we use the term *probing* as a method for printing (or aggregating) debugging information at specific points in the executable code. SystemTap generates loadable kernel modules and guarantees very low overhead [8]. For instance, one may place a probe at the *entry point* of a kernel function which is responsible for sending data packets over WiFi and another probe at its *return point*. Using these two probes, the time it takes to transmit the data can be determined by calculating the difference between the triggering times of the first and second probes. Moreover, the second probe reports the amount of data that is successfully sent through WiFi.

By leveraging the wealth of information that resides in the kernel, PowerTap adopts properly-tuned accurate power macro-models in order to generate power traces. Note that obtaining data from the kernel enables PowerTap to determine per-component power consumption of each application and process (an application comprises of single or multiple processes).

Subsequently, Therminator 2 (which comprises of transient- and steady-state thermal solvers) receives the power trace from PowerTap every second and produces thermal maps at the same rate, a process which we call *online thermal analysis*. Unfortunately, the state-of-the-art thermal simulators are known to be slow for this purpose. Thus, we borrow fast finite element analysis techniques to accelerate thermal simulations.

Given the physical characteristics of the portable device, Therminator 2 builds a *compact thermal model* (CTM) of the device, and then generates temperature maps for every device component, from the device skin to the AP. These maps can be produced per application and per process, which give important insights about the device and the software it runs. More specifically, a developer may use this information to determine applications/processes causing the temperature rise. In other words, the developer can use ThermTap framework for *thermal debugging*.

To the best of our knowledge, ThermTap is the first online power analyzer and thermal simulator for Android devices that enables device manufacturers as well as developers to debug thermal issues in the system software and applications.

We would like to emphasize the fact that ThermTap only requires a USB connection to a device in order to collect the required information and generate power and temperature graphs. The source code of ThermTap can be obtained from http://sportlab.usc.edu/downloads.

The remainder of this paper is organized as follows. First, Section II reviews the previous work. Then, Section III introduces ThermTap and details how PowerTap and Therminator 2 work. Besides, it explains how ThermTap is implemented. After that, Section IV explains the process of evaluating ThermTap followed by a case study. Finally, Section V concludes the paper.

II. Previous Work

Many efforts have been conducted on the power characterization, modeling, and metering of portable devices without direct measurement. All of these work can be classified into two main categories. First, *sampling* techniques (e.g., PowerTutor [9], Sesame [10], and [11]), which rely on polling the device internal sensors, hardware performance counters, OS kernel *sysfs/procfs* contents, or the battery sensors. Second, *event-driven* methods (e.g., eprof [12], AppScope [5], and FEPMA [6]) in which the OS kernel is properly instrumented to report desired events (usually power state transitions). It is shown in [5] that the event-driven methods have lower overhead and higher accuracy compared to the sampling techniques. Moreover, it is demonstrated in [6] that event-driven methods can capture high-frequency power change events, whereas the first method lacks this capability.

We found that the CPU model characterized in AppScope was single-core and in FEPMA was dual-core. In this work, we model a quad-core processor and consider the power correlation among the cores. Moreover, GPU model is missing in [5] and is modeled in [6] using an artificial neural network which captures the dependence between the CPU workload and that of the GPU. This approach imposes a significant computational load for the power calculator and its accuracy should be improved by direct modeling. Besides, it does not consider the effect of frequency on GPU power as well as cannot reveal which application/process is accessing GPU. In contrast, PowerTap directly models the GPU power, which is fast and more accurate. Last but not least, PowerTap considers the internal flash storage of the device, which consumes substantial power during *disk intensive* operations ($\sim$0.6W).

Thermal simulation using compact thermal modeling is a well known concept. *HotSpot* [13] is a popular tool which simulates a chip with its cooling package (comprised of a heat sink and a fan) in transient and steady-state modes. *3D-ICE* [14] is another thermal simulator which captures liquid cooling. These tools have two important drawbacks. First, they are not fast and hence not suitable for realtime (online) simulations. Second, they are not designed for thermal simulations of portable devices. In our prior work, we have introduced a simulator for portable devices called *Therminator* [15]. Even though Therminator solves the second problem, the first problem still exists, i.e., it is still slow. Moreover, Therminator does not support transient-state thermal simulations. A significant speed-up can be achieved (for steady-state temperature analysis) using NVIDIA's high-end GPGPUs. In this paper, we achieve an even higher speed-up by only using a desktop-class Intel CPU.

III. ThermTap Overview

ThermTap is a system-level power analyzer and thermal simulator designed for Android-based portable devices. It requires only a USB connection (which comes with every portable device) to communicate with the device and gather the activity information of major components. Fig. 1 shows a high-level overview of ThermTap. As can be seen, ThermTap consists of two important parts. First, PowerTap (which is a power analyzer) is responsible for collecting information about the operating state and activity levels of various system components to generate per-process and per-component power traces utilizing properly tuned power models. Second, Therminator 2 (which is an online thermal simulator) takes the device physical characteristics (from the user) as well as the power trace (from PowerTap) and generates temperature maps corresponding to every component of the device. ThermTap is responsible for synchronizing PowerTap and Therminator 2. In the remainder of this section, PowerTap and Therminator 2 and their implementations are explained in detail.

A. PowerTap: A Power Analyzer for Android Devices

PowerTap has two important modules which play key roles to generate power traces. The first one is a *system state monitor*, which collects information about the operating state and activity levels of various system components. The second one is a *power profiler*, which utilizes the system state information along with well-tuned power models for system components to produce power traces.

1) System State Monitor: PowerTap exploits SystemTap for collecting activity profiles. SystemTap has been developed mainly by Red Hat, IBM, Intel, Hitachi, and Oracle as a tool for debugging and analyzing the performance of the Linux kernel. It receives an input script written by the user, which specifies probing codes that must be executed before and after a set of target instructions (i.e., specific memory locations in the kernel or user space where probes should be inserted). Next, SystemTap compiles the script to produce a kernel module, which is subsequently *dynamically* loaded into the Linux kernel. Note that since Android is based on Linux, such modules can be used for Android-based devices as well. When a SystemTap-made kernel module is loaded, instructions in the specified memory locations are replaced with *breakpoints*, which redirect the program execution flow to a *user-defined* method. At the end of this method, the removed instruction followed by another user-defined method and a *return* instruction are executed. In addition, proper instructions are inserted before the return instruction to restore the state of previous code execution flow. This ensures the complete restoration of the CPU state. Fig. 2 demonstrates this process. A detailed description on how SystemTap works can be found in [7].

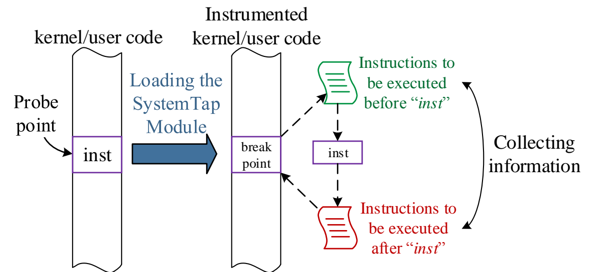

Fig. 2. SystemTap work flow.

In order to calculate the time a certain event takes from start to finish, PowerTap places probes at the *entry* and *return* points of *device driver* functions. Moreover, return probes are used to make sure that a certain action is successfully completed. For

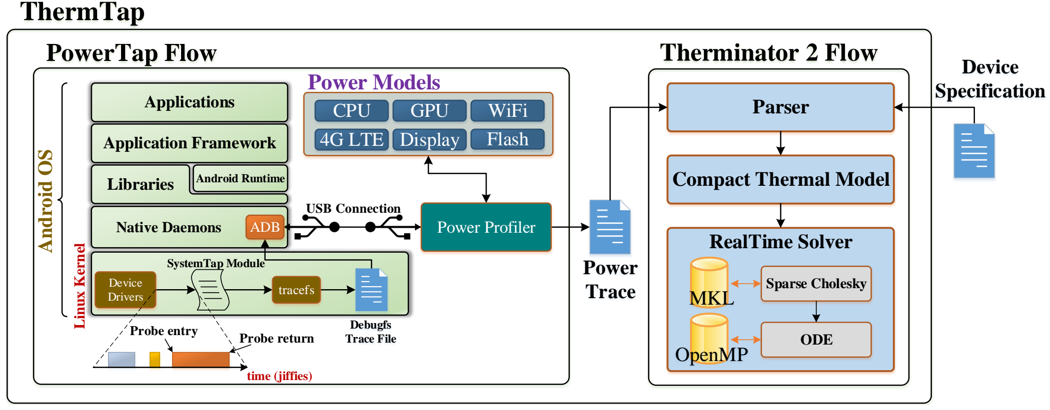

Fig. 1. ThermTap structure. On the left, the work flow of PowerTap and its interaction with Android OS is shown. On the right, Therminator 2 work flow is depicted. The user should provide a device physical specification along with the application/process that he is interested in for probing. ThermTap generates temperature maps of the selected application/process.

instance, one may try to send a packet over WiFi; however, the packet might be dropped due to weak WiFi coverage. Checking the return probe allows detection of such situations. Note that PowerTap allows multiple (different) power models to be active at the same time because it records the start and end times of events and then aggregates the relevant power models for all active events.

The information gathered by the kernel module is transferred from the kernel space to the user space in order to be read by the power profiler (see Fig. 1). We use *tracefs* to export activity log from the kernel space to the user space. Tracefs is a low-overhead in-memory file system suitable for this purpose [16]. Note that logging data directly to the disk would increase the system load significantly and hence is avoided. Finally, PowerTap connects to the device using *Android Debug Bridge* (ADB) through a USB cable, in order to collect the information stored inside the tracefs buffer.

2) Power Profiler: In this paper, we use a *Google Nexus 5* running Android 5.0 as the target device for training power models. Nexus 5 comes with a quad-core Qualcomm Snapdragon 800 processor and 2 GB memory. Please note that the power models presented in this section are general and can be applied to other portable devices.

In order to execute (synthetic as well as standard) benchmarks while controlling/monitoring the power state of various system components, we connect the phone to a PC through a USB cable. The total power consumption of the device (P^{total}_{device}) is calculated as

$$P^{total}_{device} = V_{USB} I_{USB} + V_{bat} I_{bat}, \quad (1)$$

where V_{USB} and V_{bat} denote the voltages provided by the USB and the battery, respectively, whereas I_{USB} and I_{bat} are the currents supplied from the USB and the battery to the phone, accordingly. I_{USB} is measured by cutting the USB power lane and placing it in series with an ammeter (NI-9227) to log the current. I_{bat} is logged similarly. Besides, V_{bat} is measured and logged using a voltmeter (NI-9239). The sampling rate for both of NI-9227 and NI-9239 is set to 2kHz. Note that based on the USB 2.0 specification, V_{USB} is fixed at 5V, whereas I_{USB} can be at most 0.5A. The second term in Eq. (1) is usually negative due to the direction of I_{bat} which shows that the USB current not only supplies the phone but also charges the battery. However, the smartphone under a heavy load may draw more than 0.5A current which changes the direction of I_{bat} from negative to positive and forces the battery to provide current to the system.

Knowing the value of P^{total}_{device}, we can find the power consumption of major components as follows. First, all components (except CPU) are turned off, for instance, GPU, WiFi, 4G, and display. Note that because CPU is always ON during benchmarking, it is characterized first. Also, all background processes are stopped (this is a feature provided by Android). As a result, the power consumption of each component can be characterized individually by selectively turning it on. Next, CPU power model is characterized using the *StabilityTest* benchmark. After that, each major component is turned on using synthetic benchmarks and the total power of the smartphone is measured. Knowing the CPU power model, the CPU power consumption is calculated and subtracted from the measured P^{total}_{device} to determine the power consumption of the component of interest. The power of the remainder of system (which belongs to components that are not considered) is captured as a constant and assigned to the main PCB of the device. In the remainder of this subsection, the power models that we have derived are briefly explained.

CPU power modeling: Initially, we disable all CPU cores except one. Next, a power model for a single core is derived. Then, cores are activated one by one and their power consumptions in a fully utilized state are measured for different frequencies. We observed that the power consumption of each core changes based on the total number of active cores. Fig. 3 depicts the power consumption of a single active core with respect to the total number of active cores drawn at six different clock frequencies. For these measurements, all of the cores are fully utilized and Android power and thermal governors are turned off. As can be seen in Fig. 3, at higher frequencies, the dependence of individual core power consumption on the total number of active cores becomes more pronounced. Moreover, as we explain below, the per core power consumption is minimum when the number of active cores in the target system is two.

Before presenting the detailed CPU power consumption model, we define some terms. f_i and v_i are frequency and voltage of core i, respectively. u_{core} represents the normalized utilization of a core. We calculate u_{core} in every scheduling epoch of the operating system as

$$u_{core} = (t_{user} + t_{system})/(t_{user} + t_{system} + t_{idle}), \quad (2)$$

where t_{user}, t_{system}, and t_{idle} are times that the core spends for running user space codes, kernel space codes, and being idle,

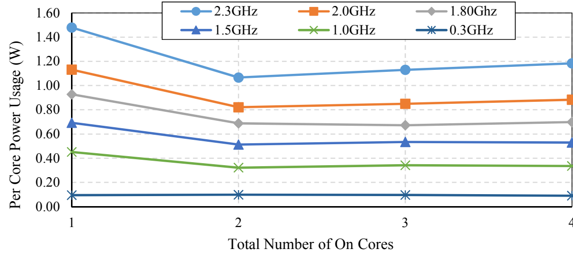

Fig. 3. Power consumption of each core in the Nexus 5 drawn with respect to the total number of active cores

respectively. Note that these values are internally determined by the OS and thus, PowerTap can simply access them for calculating u_{core} for every core i (shown by u_{core_i}). Similar to [17], we define the term *workload processing rate* for core i as

$$w_i = f_i \cdot u_{core_i}. \tag{3}$$

We attribute the behavior shown in Fig. 3 to the power consumption of other non-core components of CPU (e.g., inter-core interconnects and shared cache banks) typically referred to as the *uncore*. Assuming that *off cores* (power-gated cores) consume zero power and considering that *on cores* consume dynamic plus active leakage power during program execution but only standby leakage power when sitting idle, the total CPU power consumption (P_{CPU}) can be modeled as

$$\begin{aligned} P_{CPU} &= (P_{cores}^{dyn} + P_{cores}^{leak}) + P_{uncore} \\ &= \sum_{i \in \{\text{on cores}\}} \left(\beta_{core}^{dyn}(v_i) \cdot f_i \cdot u_{core_i} + \beta_{core}^{leak}(v_i)\right) \\ &\quad + \beta_{uncore}(w_{tot}, n_{on}). \end{aligned} \tag{4}$$

where $\beta_{core}^{dyn}(v_i)$, $\beta_{core}^{leak}(v_i)$, and $\beta_{uncore}(w_{tot}, n_{on})$ are lookup table-based fitting functions; w_{tot} is the *total workload processing rate* of CPU which is defined as

$$w_{tot} = \sum_{i \in \{\text{on cores}\}} w_i. \tag{5}$$

We expect $\beta_{core}^{dyn}(v_i)$ to be a quadratic function of v_i, whereas $\beta_{core}^{leak}(v_i)$ to be a linear function of v_i. In addition, $\beta_{uncore}(w_{tot}, n_{on})$ should be a linearly increasing function of w_{tot} and a convex function of n_{on}, where the minimum is achieved for a certain number of on cores, called n_{on}^{opt}. As mentioned before, n_{on}^{opt} in Fig. 3 is equal to 2. Note that the *non-proportionality of energy* of CPUs arises in part due to core leakage and uncore power consumption terms.

GPU power modeling: Nexus 5 has an *Adreno 330* integrated into the AP for 2D and 3D graphic processing. Adreno 330 shares main memory with the processor [18]. Hence, we only need to account for the power consumption of the GPU core. Android uses a driver called *Kernel Graphics Support Layer* (KGSL) developed by Qualcomm to provide a *Hardware Abstraction Layer* (HAL) for userspace Adreno drivers. KGSL allows various processes to create different GPU *contexts*, which are analogous to CPU processes. At each point in time, only one context can be executed on GPU. KGSL is responsible to perform *context switching*. Finally, a context is *destroyed* when its execution is finished or an exception is occurred. By tracing the *context create* request, one can simply determine which context belongs to which process, and consequently, assign the related GPU power consumption to the process. *Adreno 330* supports DVFS through a proprietary closed-source policy called *trustzone*. KGSL is responsible of applying the actions determined by the policy to the GPU hardware.

Based on the above discussion, we model the GPU power consumption as

$$P_{GPU} = \beta_{GPU}^{dyn}(v) \cdot f \cdot u_{GPU} + \beta_{GPU}^{leak}(v), \tag{6}$$

where $\beta_{GPU}^{dyn}(v)$ and $\beta_{GPU}^{leak}(v)$ are lookup-based fitting functions of the GPU voltage level (v), f is the GPU frequency, and u_{GPU} represents the normalized utilization of GPU.

WiFi & 4G-LTE power modeling: We measured the WiFi power consumption during send and receive operations. It has been observed that the power consumption of WiFi while receiving data is linearly proportional to the receive rate, whereas during the send operation, it behaves as a piecewise linear function with two thresholds; one occurs at 2Mbps, and the other one happens at 8Mbps. Similar behavior was observed for 4G-LTE with different thresholds.

Display power modeling: Nexus 5 has a full-HD IPS LCD display. As a result, the display power is linearly proportional to its brightness. This is in contrast to OLED displays, where the screen content plays a major role in power consumption [5], [6]. Thus the display power can be modeled as

$$P_{Display} = \beta_{Display} \cdot Brightness, \tag{7}$$

where $\beta_{Display}$ is the linearization coefficient and $Brightness$ is the normalized brightness value which varies from 0 to 1. According to our measurements, the IPS LCD display consumes nearly zero power when it is completely dim.

Flash storage power modeling: It is observed that when the data transfer rate (write or read) is low, the flash consumes significantly less power. We conjecture that in low transfer rates, caching and *write back* methods are used (as opposed to the *write through* technique). On the other hand, in high transfer rates, the power consumption becomes high. Hence we define a threshold for the transfer rate called β_{flash}^{thre} and model the flash power consumption (separately for read and write operations) as

$$P_{flash} = \begin{cases} \beta_{flash}^{slow}, & \text{if transfer rate} < \beta_{flash}^{thre} \\ \beta_{flash}^{fast}, & \text{otherwise.} \end{cases}, \tag{8}$$

where β_{flash}^{slow} and β_{flash}^{fast} denote the power consumption of the flash in low and high transfer rates, respectively. β_{flash}^{thre} for the read and write operation is about 70MB/s.

B. Therminator 2: An Online Thermal Simulator

Therminator 2 works based on compact thermal modeling. This is a well-known technique which utilizes the duality between the thermal and electrical phenomena; temperature, power, thermal resistivity, and thermal capacity are duals of voltage, current, electrical resistivity, and electrical capacity, respectively [19]. Thus, a system can be modeled by an RC network and analyzed to determine the nodal voltages in order to find the temperature of each component in the system.

In the steady-state thermal analysis, similar to the DC analysis of RC circuits, a system of linear equations is solved. As explained earlier, previous thermal simulators (such as [13], [14], [15]) adopt *LUP decomposition* which is a generic matrix decomposition technique for any matrix. As suggested in [20] for finite-element solvers, we use a

parallelized *Cholesky decomposition* which is proven to be much faster than *LUP decomposition* (while delivering the same solution accuracy) [21]. With this technique, Therminator 2 running on a $300 CPU beats its initial version (which takes advantage of a $3,200 GPU) by a factor of 27X on average for a system with a large number of subcomponents (>7,000). Moreover, the Therminator 2 runtime is measured to be below 0.35 seconds even for systems with very large subcomponent count (≈18,000). This delay is not noticeable by the ThermTap user because the steady-state solver is required to be called once to derive the initial temperature used for the transient-state analysis.

Based on our experiments, the system of ordinary differential equation (ODE) of thermal equations is *stiff*, which means that numerical methods for solving it are *unstable*, unless the step size for solving the ODE is taken to be extremely small [22]. Hence, we have utilized *Runge-Kutta* method with adaptive steps, where the steps are chosen by the 5th order *Dormand-Prince* technique. This technique is shown to handle stiff equations very well [22].

We tried to compare Therminator 2 transient-state solver with that of HotSpot 5; however, it turned out that it took a few hours for HotSpot's solver to simulate one second of thermal change and sometimes, the solution diverges. Please note that due to the stiffness of ODE, one cannot simply increase the step size of solver to improve the runtime speed. Hence no fair comparison is feasible here. Therminator 2 can calculate the device temperature after one second in real time (in less than one wall-clock second) when the number of subcomponents in the system is nearly 5,000. Thus, we perform all of our measurements with this maximum component count.

C. ThermTap Implementation

PowerTap is implemented in Java, whereas Therminator 2 is implemented in C++. ThermTap, which is also written in Java, synchronizes PowerTap and Therminator 2 through file system. We tested ThermTap on a Linux machine (Debian 8) with a quad-core Intel Core i7-3770 processor running at 3.4GHz and 8GB of memory.

As explained previously, Therminator 2 requires to exploit the system's maximum performance. Hence, we selected C++ for its implementation. Besides, Therminator 2 uses *Eigen* with *Intel Math Kernel Library* (MKL) as a back-end to solve steady-state thermal equations. For solving ODEs, Therminator 2 utilizes *ODEINT*, which is an open source C++ library for numerically solving ODEs.

IV. ThermTap Evaluation

We first calibrated PowerTap to make sure accurate power traces are generated. Different benchmarks are executed to train power models. We measured the CPU runtime overhead of inserting the SystemTap module as 1.7% under heavy load which is very low as expected. On average, PowerTap values differed by 15% from the measured values. In Fig. 4, we demonstrate a test case which shows how Android thermal manager works. Initially the system was in the idle state. Next, StabilityTest benchmark was executed. This benchmark heavily stresses CPU and memory. After 40 seconds, the total power consumption drops due to the overheating issue and as a result of the thermal manager throttling the CPU core frequencies. This figure also shows that how well PowerTap estimations follow the measurement values.

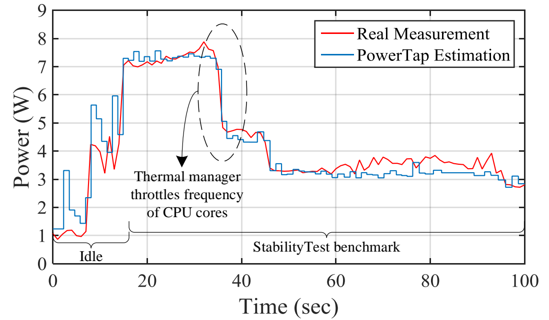

Fig. 4. Comparing the power trace generated by PowerTap with measured values

Next, we used the power values generated by PowerTap to calibrate ThermTap. Similar to the technique described in the earlier version of Therminator [15], we tore apart a Nexus 5 smartphone to build its physical model. We used temperature of three points to calibrate and verify ThermTap results; the AP internal temperature sensor and two sensors placed on the hottest spots of the rear case and the display of the phone. Omega DAQ-2408 was used to log temperatures of these two sensors. Fig. 5 shows the transient temperature change when the smartphone is cooling down. On average, an error of 0.5°C, 1°C, 1.5°C for the rear case, display, and AP were observed, respectively. Given the fact that the accuracy of the AP sensor and DAQ-2408 are ±1°C and ±0.5°C, respectively, the above error values are acceptable. Note that the AP temperature changes very quickly; however, the display and rear case temperature are varying very slowly. This shows the fact that the thermal constant of AP is small compared to that of the display and rear case.

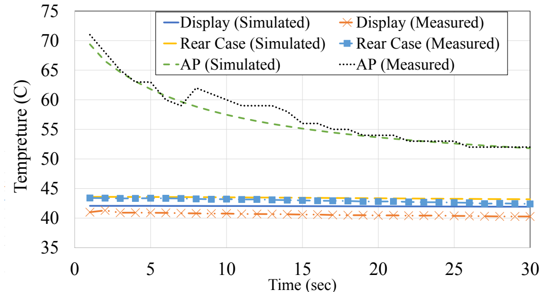

Fig. 5. Comparison of measured and simulated temperatures

Case study: We considered executing two video players on Android, namely *VLC* and *QQPlayer*. An HD-quality video called *Big Buck Bunny* was selected as the benchmark. The ambient temperature during the experiment was about 25°C. From an end-user point view, we observed that this video runs smoothly on VLC, whereas it has a noticeable lag on QQPlayer.

Next, we used ThermTap to study power and thermal behavior of these two applications. Figures 6a and 6c show ThermTap results while running QQPlayer and VLC, respectively, when the power and thermal impacts of all processes are considered. As can be seen, Nexus 5 burns about 3W when running QQPlayer, whereas it only consumes 2W when executing VLC. Moreover, unlike the second scenario, the GPU was heavily stressed (shown as a blue slice in the pie chart) when QQPlayer had been executed. Temperature maps show that the maximum temperature of AP reaches 51°C and 42.5°C when QQPlayer and VLC were executed, respectively.

As explained earlier, the user can further study the behavior

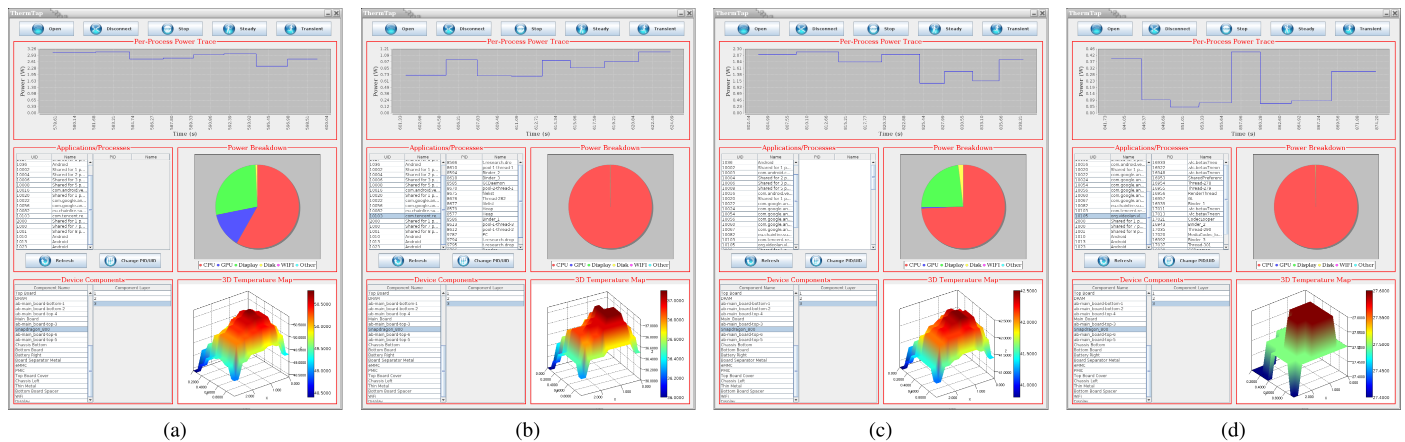

Fig. 6. ThermTap results while running QQPlayer (a) showing the entire system and (b) showing only the impact of the player process. ThermTap results while running VLC (c) showing the entire system and (d) showing only the impact of the player process.

of QQPlayer and VLC processes (as opposed to studying the accumulated effect of all processes) using ThermTap. The results are shown in Figures 6b and 6d. These figures show the power and thermal impact of the aforesaid processes. Note that the temperature impacts are reported with respect to the ambient temperature. For instance, the maximum AP temperature of 37°C reported in Fig. 6b means that the AP temperature is increased by 12°C only due to the QQPlayer process. One interesting fact is that unlike what is shown in Fig. 6a, the QQPlayer process did not use GPU. We investigated other processes in the system while running QQPlayer and it turned out that another process called *SurfaceFlinger* had been utilizing GPU. SurfaceFlinger is a display server developed by Google for Android devices. QQPlayer utilizes SurfaceFlinger APIs to communicate with GPU. We conclude that despite the fact that QQPlayer heavily stresses CPU and GPU, it does not efficiently utilize it, which leads to hotter AP temperature and performance lag.

We plan to extend the power models of PowerTap such that ThermTap can be used for probing and finding thermal bugs in a wider range of hardware components. Memory is one of the major power consumers that is not modeled explicitly in this work. Currently, the cache power consumption is captured in the CPU power and the DRAM power is distributed between CPU and GPU power values. Another considerable power consumer is the battery. GPS is also a substantial power consumer in smartphones, which has a significant impact on the temperature of the device. Finally, there are other components that contribute to the total power consumption like power management IC(s) and PCB tracks and pads. These components will be taken into account in the future revisions of ThermTap.

V. Conclusion

This paper introduced ThermTap, which enables system and software developers to monitor the power consumption and temperature of various hardware components in a portable device as a function of running applications and processes. ThermTap comprise of a power analyzer, called PowerTap, and an online thermal simulator, called Therminator 2. PowerTap generates power traces, whereas Therminator 2 produces various temperature maps including those for the device components and device skin. Advanced numerical methods are used to enable Therminator 2 to be executed in realtime. Experimental results confirmed that with the aim of PowerTap andTherminator 2, ThermTap can generate accurate temperature maps. Besides this, developers can use ThermTap to find thermal bugs in a system or application.

Acknowledgement

This research was supported in part by the National Science Foundation and the Semiconductor Research Corporation.

References

[1] C. Albanesius, "Smartphone shipments surpass PCs for first time. what's next?" http://www.pcmag.com/article2/0%2c2817%2c2379665%2c00.asp.

[2] L. Brown and H. Seshadri, "Cool hand linux - handheld thermal extensions," in *Linux Symposium*, 2007.

[3] S. Thomas and Z. Rui, "Thermal management in user space," in *Linux Symposium*, 2008.

[4] S. O. Memik *et al.*, "Optimizing thermal sensor allocation for microprocessors," *TCAD*, vol. 27, no. 3, pp. 516–527, 2008.

[5] C. Yoon *et al.*, "Appscope: Application energy metering framework for android smartphone using kernel activity monitoring." in *USENIX ATC*, 2012, pp. 387–400.

[6] K. Kim *et al.*, "FEPMA: Fine-grained event-driven power meter for android smartphones based on device driver layer event monitoring," in *DATE*, 2014, pp. 367:1–367:6.

[7] B. Jacob *et al.*, *SystemTap: instrumenting the Linux kernel for analyzing performance and functional problems*. IBM, 2008.

[8] F. C. Eigler and R. Hat, "Problem solving with Systemtap," in *Proc. of the Linux Symposium*, 2006, pp. 261–268.

[9] L. Zhang *et al.*, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *CODES+ISSS*, 2010, pp. 105–114.

[10] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *MobiSys*, 2011, pp. 335–348.

[11] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone." in *USENIX ATC*, 2010, pp. 271–285.

[12] A. Pathak *et al.*, "Fine grained energy accounting on smartphones with eprof," in *EuroSys*, 2012, pp. 29–42.

[13] K. Skadron *et al.*, "Temperature-aware microarchitecture: Modeling and implementation," *ACM TACO*, vol. 1, no. 1, pp. 94–125, 2004.

[14] A. Sridhar *et al.*, "3D-ICE: fast compact transient thermal modeling for 3D ICs with inter-tier liquid cooling," in *ICCAD*, 2010, pp. 463–470.

[15] Q. Xie *et al.*, "Therminator: a thermal simulator for smartphones producing accurate chip and skin temperature maps," in *ISLPED*, 2014, pp. 117–122.

[16] A. Aranya *et al.*, "Tracefs: A file system to trace them all." in *FAST*, 2004, pp. 129–145.

[17] I. Hwang and M. Pedram, "A comparative study of the effectiveness of cpu consolidation versus dynamic voltage and frequency scaling in a virtualized multi-core server," Department of Electrical Engineering, University of Southern California, Tech. Rep., 2013.

[18] "Qualcomm 2D/3D graphics driver," http://lwn.net/Articles/394665/, [Online; accessed Oct 31, 2014].

[19] M. Pedram and S. Nazarian, "Thermal modeling, analysis, and management in VLSI circuits: Principles and methods," *Proc. of the IEEE*, vol. 94, no. 8, pp. 1487–1501, 2006.

[20] Y.-C. Li *et al.*, "Direct finite-element-based solver for 3D-IC thermal analysis via h-matrix representation," in *ISQED*, 2014, pp. 386–391.

[21] W. Ford, *Numerical Linear Algebra with Applications: Using MATLAB*. Academic Press, 2014.

[22] J. C. Butcher, *Numerical Methods for Ordinary Differential Equations*, 2nd ed. Wiley, 2008.