Energy-Performance Tradeoffs in Smartphone Applications

Adrian Kwok, Arun Bharadwaj, Zachary Blair, Arrvindh Shriraman, Brian Fraser
School of Computing Science, Simon Fraser University
{adriank, abharadwaj, zblair, ashriram, bfraser}@cs.sfu.ca

Abstract

Smartphones have become ubiquitous, making it imperative that we account for their energy consumption and continuously monitor active applications on these battery-powered devices. Applications built for smartphones have varied hardware resource and energy requirements; some applications may only intermittently utilize the CPU, whereas others may be both computationally and graphically intensive. The randomness and interactivity of the applications also cause varied power profiles.

In this paper, we analyze the power usage, performance, and energy consumption of popular applications on the Android platform. Our profiling system, AppLogger, is an efficient kernel-level event tracker that gathers resource-usage information. It exhibits low overhead – on average 7% – and logs events at context switch ensuring fine-grain observation ability of up to 1000Hz. To monitor the power drawn from the battery we developed a cost-effective (< $150), phone independent, and easily replicable energy gauge. Using this framework we analyzed the effects of different CPU frequency and voltage governors. While the conventional approach of turning down hardware clock speeds saves power, it also increases execution time and negatively affects overall energy consumption and hence battery life. Running a browser with the CPU at 1GHz consumes 251J of energy at 1.5W of power; at 576Mhz, the phone consumes just 1.1W but requires 300J. We also showed that the default Android frequency governor, "Ondemand", has minimal impact on power and gives minimal energy savings (< 10%) compared to running the CPU at its maximum frequency, with potential for improvement. Finally, we analyzed the average power consumed for different application categories; outside of the games category, we found that Facebook, Skype, and Adobe Reader were the three most power hungry applications. We also observed significant fluctuations in the instantaneous power requirements of different applications, exposing the potential for fine-grained dynamic power management.

1 Introduction

Today, users are in the midst of a major transition to smartphones and mobile devices for accomplishing many of their daily tasks. Smartphones have brought with them new software stacks and are enabling programmers of varied experience and expertise to develop applications; furthermore, the applications that current mobile systems can support are mainly constrained by the lifetime of the battery. It is important that we allow developers and end-users to understand the energy and resource requirements of applications on their smartphones.

Smartphone applications are highly interactive – allowing for various forms of input – and employ rich GUIs; these applications are often designed using an event-driven methodology to enable high concurrency and interactivity, and as we show in Section 5, have varied resource requirements. Some smartphone applications may only intermittently utilize the CPU, whereas others may be both computationally and graphically intensive and exercise many different hardware components. Moreover, mobile platforms include many system services that may be running in the background yet consuming non-trivial amounts of energy.

In this paper, we characterize the power consumption, energy and performance profile of smartphone applications by conducting studies on a widely used consumer phone, the Google Nexus One. A key challenge with a complex multi-level software framework is tracking the processes that perform work on behalf of the application and attributing resource usage. For example, when playing Angry Birds, *AudioTrackerThread* is a separate process that handles audio for the game while another process – *PlaybackThread* – provides rendering services. To overcome this, we developed *AppLogger*, an OS-level logger based on the Ftrace mechanism present in linux kernel. The OS-level framework helps us track the events corresponding to the entire system, supports a cleaner implementation independent of changes to the SDK, and most importantly exhibits very low overhead, ensuring that applications are minimally perturbed by our logging. Through AppLogger, we track per-thread resource usage statistics including CPU utilization, brightness levels, and WiFi and 3G usage; we also use hardware counters on the ARM system-on-chip to understand architectural characteristics. Section 3.1 describes in more detail the overhead associated with user-level logging and compares it with our AppLogger approach.

Tracking energy consumed by a consumer phone is challenging since they lack sensors that can be instrumented for tapping into the power supply. We have devised a novel solution to tap into the lines between the battery and the phone – we utilize a custom USB interface board to log power values at a fine granularity of up to 100Hz, which is then fed into a monitoring PC. Section 3.2 elaborates on the details of this set up. Since smartphones include a number of power knobs such as Dynamic Voltage and Frequency Scaling (DVFS), fine-grained clock gating of unused hardware, and sleep modes for

various components (e.g., WiFi, GPS), we used this hardware framework to study the detailed power profile of smartphone applications, and we find that applications demonstrate significant behavioral differences at various CPU frequency levels. We have observed system power consumption reductions of up to 40% between the lowest and highest CPU frequencies. Furthermore, while the conventional approach of turning down hardware clock speeds to save power works well to minimize energy consumption for *fixed duration* workloads such as watching a video, we find that many popular applications actually perform a *fixed amount* of work, and thus turning down the CPU frequency prolongs application execution time and negatively affects overall energy consumption. In most cases, the smarter energy optimization strategy would be to *Race to Sleep* – that is, to run at maximum frequency to complete the fixed amount of work, and then idle the device. Overall, we make the following contributions:

- We have developed a framework to study the performance and energy profile of smartphone applications on a consumer phone. We have built a lightweight Ftrace-based OS-level logger based on Android's linux kernel to track all of the activities of an application scattered across different processes.
- We characterized the energy and power consumption of smartphone applications drawn directly from popular categories in the Android marketplace. Interestingly, outside of the games category, we find Facebook, Skype and Adobe Reader to be the three most power hungry applications; contrary to most expectations, the default Android video player is the least power hungry.
- We characterized the fine-grain impact of frequency and voltage scaling on both performance and energy consumption. We find that for some applications, such as web browser, when loading a webpage, running the CPU at its maximum frequency is the most energy efficient strategy. The optimal frequency and voltage is application specific and can yield up to a 20% energy reduction, as is the case with the game TapTapRevenge 4.
- Finally, we have developed a non-intrusive and cost-effective hardware framework ($<$ \$100) to directly tap into the battery supply of any phone and measure the energy consumption of smartphone applications easily and at a high frequency.

2 Background

2.1 Energy and Power

The overall power consumption of a digital chip can be summarized as: $P = \alpha.C.f.V^2 + P_{static}$ where α is the activity factor, C is the net capacitance load of the system, f is clock frequency and V is the supply voltage. The chip automatically regulates the voltage for the specific operating frequency. P_{static} indicates the power consumed when the digital chip is inactive but switched on. Typically, mobile devices

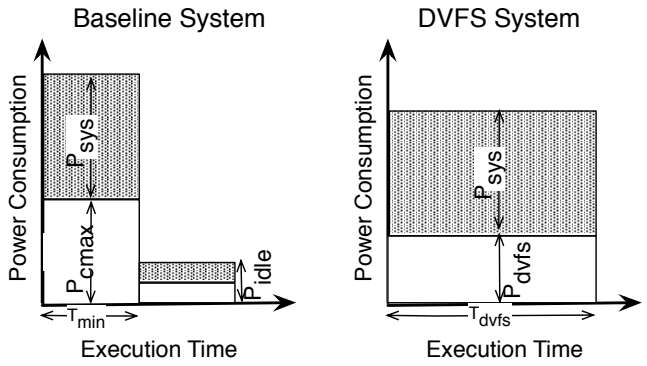


Figure 1: **Relation between Energy and Power. Impact of DVFS on power and overall energy. Area of rectangle indicates energy. Shaded region indicates power of system components not impacted by DVFS.**

seek to minimize the idle energy of the system and support low power sleep states for individual components. DVFS is a technique that can optimize for the dynamic power consumption of the system by reducing f, and thereby V as well.

We discuss the analytical relation between power and energy (see Figure 1) and characterize the impact of dynamic power reduction strategies. The power consumed by a smartphone can be attributed to two major components: the chip level components influenced by DVFS, and the system level components that are independent of DVFS. In the baseline system with the CPU running at peak frequency, an application's execution time is indicated by T_{min}, the average power consumed by the chip components is P_{cmax} and the system level components is P_{sys}. The total energy for the application is $E_{baseline} = (P_{sys} + P_{cmax}).T_{min}$. When the CPU frequency and voltage is tuned down, the power of the chip components reduces to P_{DVFS}, but the power consumption of other system components is unaffected; the execution time increases to T_{DVFS} ($> T_{min}$). The total energy of the DVFS system can be evaluated as $E_{baseline} = (P_{sys} + P_{dvfs}).T_{dvfs}$. As we discuss in section 5, for some smartphone applications such as the default Android browser, the increase in execution time increases the total energy consumed by the overall system.

2.2 Instrumenting the Android Software Stack

The Android SDK by design, contains multiples layers of software abstraction. A key challenge with this multi-tier framework is tracking and attributing resource usage to specific applications. Earlier works implement an application level logger [5, 8] and probe the various low level $/proc/$ and $/sys/$ interfaces to obtain the requisite information such as CPU utilization, network usage, and display brightness. Unfortunately, probing these low-level virtual file systems requires a user-level application (normally written in Java) to traverse through multiple layers of Android libraries and abstraction, introducing significant overhead and perturbing the overall application execution. To reduce these overheads, user-level loggers reduce the frequency of event logging to as low as 1Hz [8]. This results in missing many fine-grained vari-

ations in the application's execution profile and losing many opportunities for energy optimizations. Fine-grained tracking of resource usage is particularly important for smartphone applications which typically run for a short duration and demonstrate varied execution behavior.

3 System Framework

3.1 System Event Tracking

We developed a kernel-level event tracker called AppLogger – an Ftrace-based logger – to gather various resource usage statistics at process context switches. Ftrace is a function tracer utility present in the linux kernel which traces various operating system events by inserting static tracepoints at specific areas in the kernel. Ftrace has the ability to trace events such as scheduling, interrupts, timers, and many others. Of particular interest is the *sched* event Ftrace module, which has the functionality to log task statistics such as self- and parent- Process IDs and CPU utiliztion at every process context switch. We modified this module to develop AppLogger, which allows for the logging of useful statistics such as performance counters, WiFi, 3G byte and packet counts, display brightness, and CPU frequency and voltage. This information is obtained directly from the respective drivers - WiFi from the IPv4 layer, 3G data from the *msm_rmnet* driver, display brightness from the LCD driver, and the CPU frequency and voltage from the *acpuclock-qsd8x50* driver.

Ftrace is enabled by configuring the kernel to compile Ftrace and then flashing this Ftrace-enabled kernel onto the Android phone. AppLogger is enabled from the user-space through the Ftrace *sched* event by echoing 1 to */sys/kernel/debug/tracing/events/sched/enable* to generate trace file at */sys/kernel/debug/tracing/trace*. We wrote a custom Android application called AndroidAppLogger, which invokes a user-requested Android application of interest, starts AppLogger, brings the application into the foreground (executing it, if it is not already running) and waits until the user requests to stopping logging. AndroidAppLogger also coordinates this with our energy logging framework, *PowerLogger*.

3.2 Energy Tracking

Investigating the power consumption of smartphones requires accurate real-world power sensing hardware. The challenge with a phone is getting reliable current and voltage readings without damaging the phone. The solution we used was to create a 3D plastic replica of the smartphone's battery and its battery compartment. The plastic replica battery is inserted into the phone, and the real battery is placed into the plastic replica battery compartment. The electrical battery contacts in each of the plastic replicas are then wired to a power measurement board featuring the Linear Technology LTC2942 chip. Figure 2 shows our overall setup.

The LTC2942 is a coulomb counter which measures the voltage and coulombs (current[Amps] $\times$ time [seconds]) consumed by the phone and reports it to a PC via a USB communication module (FTDI UM232H). With this set up the phone and battery each have reliable connections to the sampling hardware with no modifications needed to the phone itself. Furthermore, because the original battery is still being used to supply power, the behaviour of the phone is unaffected.

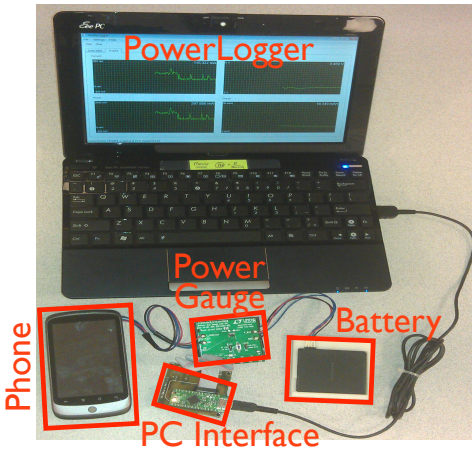


Figure 2: **Fine-grain power measurement of smartphones. Monitoring PC is running our PowerLogger Application.**

We developed a logging application, *PowerLogger* which runs on the PC connected to the measurement hardware. This application polls the LTC2942 measurement chip at a rate between 10-50hz, reading the instantaneous voltage (v_n) and the number of coulombs consumed by the phone thus far during the test (q_n). It also records the time of the sample (t_n).

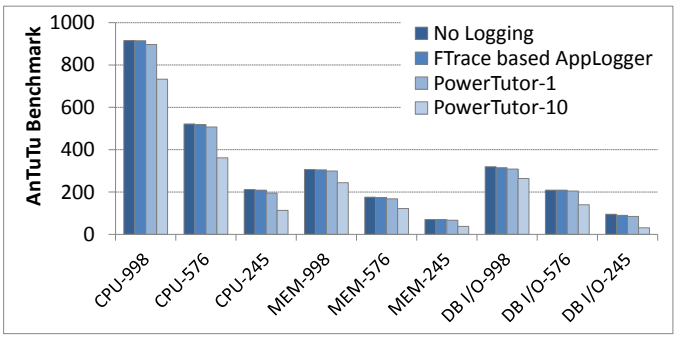
The PC based logging application converts the coulomb-counter and voltage readings from the measurement hardware into instantaneous power and instantaneous current results. It calculates the instantaneous current as: $i_n = \frac{q_n - q_{n-1}}{t_n - t_{n-1}}$ The instantaneous power (p_n) is calculated by: $p_n = i_n \times v_n$

The total hardware cost of our overall set up is $100 and is almost completely independent of the phone model. Instrumenting a different phone model simply requires creating new plastic replicas of the battery, and the battery housing ($\simeq$ $20).

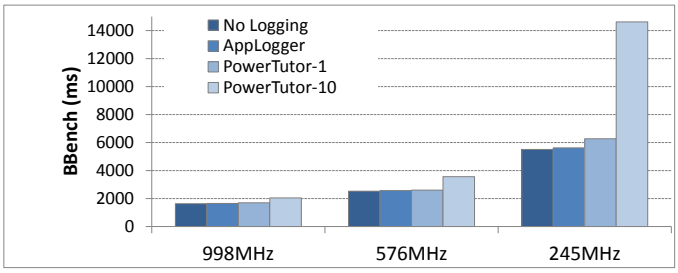
4 AppLogger Overhead

We compare the overhead of using our Ftrace-based AppLogger over existing technologies such as PowerTutor [8]. PowerTutor is an Android user-level application which logs application resource-usage, amongst other features. PowerTutor uses a polling-based approach to gather resource-usage statistics at a frequency of 1Hz. This low rate of logging is insufficient to capture the resource-usage of processes at a fine granularity. In contrast, AppLogger collects task statistics at every context switch, which is a frequency often greater than 100Hz. Even at such a high rate of logging, this approach does not add much overhead to the system. In contrast, a user-level polling based approach adds significant overhead at a logging frequency of even 10Hz since it has to traverse multiple layers of the software stack.

We compare the overhead of AppLogger against PowerTutor logging at 1Hz (paraphrased as PowerTutor-1) and



(a) AnTuTu Benchmark (higher is better)



(b) BBench Benchmark (lower is better)

Figure 3: **Comparing the overhead associated with the Ftrace-based AppLogger when running different workloads. PowerTutor-1: user-level logging at 1Hz, PowerTutor-10 : user-level logging at 10Hz.**

a version of PowerTutor with the logging rate increased to 10Hz (PowerTutor-10). We note that PowerTutor was not designed to log at 10Hz, but we chose it to improve the resolution of the logged data towards the accuracy we see with Ftrace. We compare the results of AppLogger and PowerTutor with the baseline Android system which has all logging turned off. We used several popular benchmarks, including AnTuTu, Quadrant Pro, and BBench [3], to determine the overhead as shown in Figure 3. AnTuTu and Quadrant Pro include multiple synthetic microbenchmark tests for the CPU, memory, and internal storage, while BBench employs a more realistic workload setting and is a web-page rendering benchmark comprising of offline-versions of many popular websites on the internet (e.g., Youtube, CNN). With AnTuTu, AppLogger demonstrates negligible overheads for the CPU and memory tests and shows only 1.5% degradation for the I/O test, PowerTutor-1 shows performance losses of 3% for the CPU, 2% for memory, and 6% for database I/O, and PowerTutor-10 fared much worse with over 20% performance loss in each of these tests. Quadrant Pro demonstrated similar trends, except for the CPU test on which AppLogger and PowerTutor-10 exhibited a $\simeq$ 20% performance degradation. With BBench, AppLogger shows less than 1% performance loss, while PowerTutor-1 and PowerTutor-10 show around 4% and 20% performance loss respectively at 998Mhz; however, as we lower the frequency to save power, PowerTutor's overhead dramatically increases, and at 245Mhz PowerTutor-10 requires nearly 2.5$\times$ longer to complete the BBench run over the baseline.

5 Performance and Energy

Tasks performed on a smartphone can often be characterized as being either: 1) fixed work, or 2) fixed duration. An example of a fixed work task is rendering a set of webpages [3], which does not require the application to be run for a certain amount of time, but rather that the application finish rendering a fixed number of webpages. In contrast, examples of fixed duration tasks include listening to music, watching videos, or playing certain video games.

It is critical to realize that with mobile devices the key objective is to minimize the total energy an application consumes to complete a task. We installed a custom firmware on the smartphone that unlocked 6 levels of frequency voltage scheduling, varying from 245MHz to 1GHz and 0.95 to 1.00V. We tested the effect of throttling back the speed of the CPU to complete a number of tasks for different applications. Figure 4 shows the power usage and total energy consumption profiles for the BBench benchmark with the Opera Mobile web browser, and Figure 5 shows the same profiles for the Tap Tap Revenge 4 rhythm game. The top of each figure shows the power being consumed by the smartphone during the active task and, at the bottom, the total energy consumed to complete the task. The total energy is calculated by integrating the area under the power consumption waveforms

Figure 4 shows that lower CPU frequencies are correlated with lower power consumption. However as the CPU frequency drops, the completion time for the webpage rendering extends quite significantly. At 768MHz, the average power consumption is 1300 mW but execution time is 220s, whereas at 576MHz the average power consumed 14.8% less (1107 mW), but it runs for 30% longer (286s). The total energy consumed at the slower CPU speed is 10% higher than at the higher clock speed. At 245MHz the execution time increases by 4$\times$ causing 2$\times$ the energy consumption as opposed to at 998MHz.

The Tap Tap Revenge 4 (TTR4) game (Figure 5) demonstrates a contrasting phenomenon. The TTR4 test is comprised of a short initial phase where the game loads up and a longer game-play phase where the user plays the game while listening to the song. The power trace shows that for all of the CPU frequencies there is a noticeable change in power consumption between the initial loading phase and the game-play phase. The actual game-play phase is approximately same length at all frequencies because the level lasts the fixed length of the song. The major difference in run-times for the different CPU frequencies is the time it takes to load the level; at 998MHz it takes 15s to load, whereas at 245MHz it takes 50s.

Interestingly, for all but the minimum CPU frequency, energy is saved by lowering the frequency because although additional time is needed to complete the task, the system consumes less power. For example, at 998MHz, the average power consumption is 1713mW and it takes 115s to complete; whereas at 384MHz it consumes 25.8% less energy on average (1270mW), but only takes 13.8% longer (131s). This results in 15.8% savings in total energy consumed at the lower

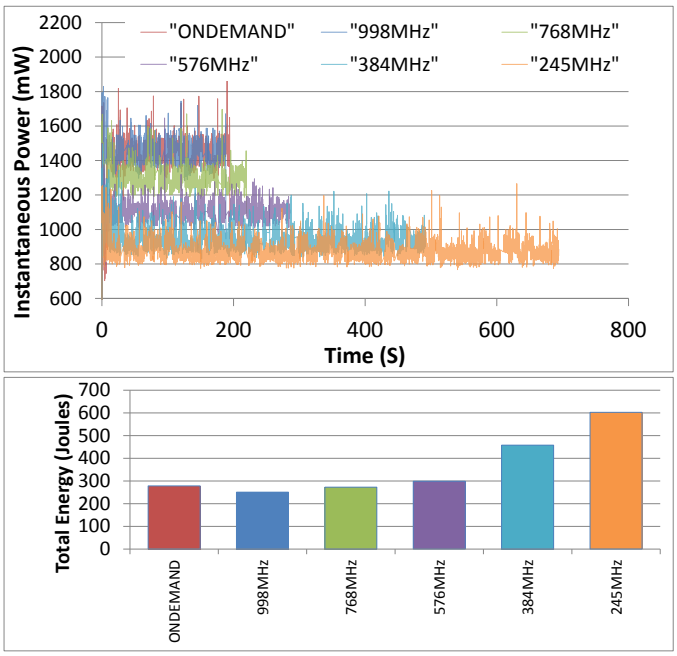


Figure 4: **Opera Mobile running BBench [3]. Top: Power consumption trace to complete one BBench iteration. Y-axis starts at 600mW. Bottom: Total energy consumed to complete that iteration. (Best viewed in color)**

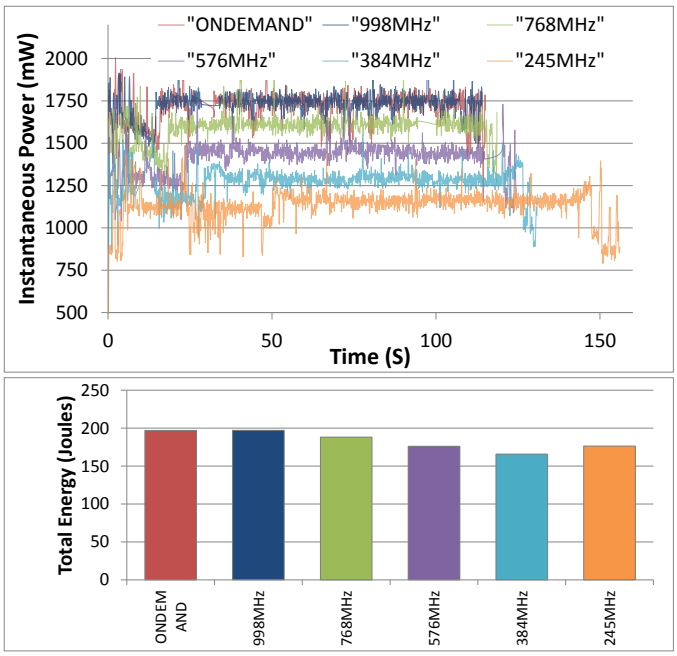


Figure 5: **Tap Tap Revenge rhythm game. Top: Power consumption trace to complete one level. Y-axis starts at 500mW. Bottom: Total energy consumed to complete one level. (Best viewed in color)**

CPU frequency. At 245MHz the additional load-time for the test negates the energy saved by consuming less power.

A good power optimization strategy for TTR4 would be to have the CPU run at a higher speed while loading the level, in order to load quickly, and then during the game-play phase run at a slower CPU frequency to minimize power consumption.

Table 1: **Tradeoff between FPS and CPU frequency. Numbers in brackets indicate % improvement in power consumption relative to 998Mhz. Negatives indicate lower power consumption.**

	ONDEMAND.	998Mhz	576Mhz	245Mhz
Angry Birds	54 (-6.1%)	54 (1.63W)	50 (-11%)	37 (-26%)
Tap Tap	44 (-6.2%)	44 (1.75W)	35 (-17%)	16 (-36%)
Blood & Glory	17 (-1.1%)	17 (1.74W)	*14 (-18%)	*4 (-34%)
Tetris	31 (-0.6%)	31 (1.69W)	31 (-18%)	19 (-31%)

*Blood & Glory: at 576Hz, some swipes are not registered. At 245MHz, no swipes are registered and unable to complete level.

5.1 Gaming QoS and DVFS Tradeoff

To investigate game playability, we connected the smartphone to a PC and used the Qualcomm's Adreno Profiler [4] to measure the Frames Per Second (FPS) of the game-play phase at different CPU frequencies, shown in Table 1.

Across the four games that were tested at different CPU frequencies, only Blood & Glory was found to suffer a serious QoS degradation when the CPU frequency was reduced; at 576MHz, the game dropped to 14FPS and became difficult to play, and was even worse at slower speeds. However, for other games, our informal QoS analysis suggests that there is a significant amount of energy to be saved by reducing the CPU frequency as long as the game is able to maintain acceptable frame rates. For example, Angry Birds could maintain 37 FPS at 245MHz while still being completely playable and reducing the power consumption by 26%; running it at 576Mhz would have still saved a noteable amount of power (11-18%) with a minimal – and difficult to preceive to the naked eye – loss in FPS (10-20% decrease). In comparison, the *Ondemand* governor conserves much less power (only 6%).

5.2 Power Consumption of Daily Applications

Figure 6 shows the average power consumption of popular applications on the Android market. Overall we find that the default *Ondemand* frequency controller has minimal effect on saving power. In all applications other than the default video player and Skype, the smartphone with *Ondemand* consumes more energy than the phone running at 768Mhz, with an average savings of only 4.5% over the 988Mhz setting. DVFS has the least impact on the video player since other power hungry components in the system (e.g., display, WiFi) dominate the overall power consumption. Facebook, Skype and Adobe are the most power hungry applications when the CPU is running at its maximum frequency. However, DVFS has significant positive impact on these applications; at 576Mhz the power consumption reduces by 19% for Facebook, 11% for Skype and 28% for Adobe. Facebook and Skype are significantly influenced by the communication with their online servers; Facebook transmits $\simeq$12KB/s and Skype transmits $\simeq$6KB/s on average. With Adobe Reader, initial investigation indicates that the parsing phase is the primary power hog. The PDF parsing phase utilizes up to 50% of the CPU and has a high branch misprediction rate (43.5 mispredictions/10K instructions). We find the video player to be highly optimized and

has the lowest power consumption amongst all applications investigated. Interestingly, Twitter (which transmits 3KB/s over the network) consumes as much power as Google Maps. Our analysis suggest suggests that significant improvements can be made to reduce the network behavior of Facebook and Twitter. Finally, we find that the Opera Mobile browser consumes marginally less power (3-4%) than the default browser. Overall, our analysis suggests that application-specific fine-grain auditing of energy consumption and tuning of dynamic power knobs is needed to eliminate waste and improve overall system energy efficiency.

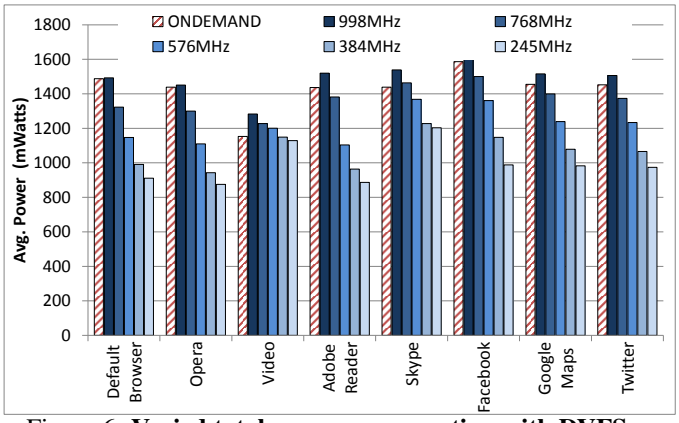


Figure 6: **Varied total power consumption with DVFS.**

6 Related Work

The ECOsystem [7] work was one of the earliest to present a case for energy management in the operating system. Zeng et. al. analyzed the influence of hardware knobs on the energy profile of hardware components (e.g., memory) and proposed that the policy be embedded into the OS. We have highlighted that energy is the key focus of mobile systems and that techniques that lower the power consumption in current systems adversely affect total energy consumption. Flinn and Satyanarayanan [2] studied the benefits of lowering the QoS of an application on its overall energy consumption. We only highlight the power-saving configurations that cripple the ability of the application to deliver acceptable interactability.

Recently researchers have analyzed the power consumption of the various hardware components in a smartphone [1]. This work used an open hardware development phone to tap into the power supply lines of the individual components. Others [5, 8] have estimated the power consumption of the various components using a linear regression of various system events (e.g., packets sent over WiFi). These works use a user-level logger program to ensure easy installation and deployment. Unfortunately, as we have quantified, these user level logging programs encounter severe overheads and degrade battery life and application performance when trying to record statistics at a fine granularity. These tools are more useful for analysis during application development rather than in fine-grained online power management.

In this study, we find that for many smartphone workloads, lowering the CPU frequency increases execution time, which offsets any gain in power savings and increases energy consumption overall. For battery operated devices *energy* should be the focus, since the devices are implicitly low power by design. Others have observed a similar trend in webservers [6].

7 Conclusion

We considered two competing energy optimization approaches: *Race-to-Sleep*, and CPU frequency reduction. Our analysis of different Android applications shows that no single approach yields energy savings across all situations. We found that for tasks which have a fixed amount of work, such as loading a webpage, running the CPU at maximum frequency is the most efficient because it reduces the time the tasks take, during which the screen and other power hungry components have to be powered up. On the other hand, we found that for tasks which run for a fixed duration, such as playing a video or a game, running the CPU at a lower frequency is the most efficient because the nature of the task dictates the execution time.

Our analysis of playability of different games at different CPU frequencies shows that there is quite a bit of flexibility in terms of reducing the CPU frequency without compromising game playability or fidelity. This should be an important heuristic to use in DVFS algorithms on smartphones to minimize game energy consumption and maximize battery life. Finally, our study of various workloads reveals that Facebook, Skype, and Adobe Reader are the top three power hungry non-gaming applications. Many applications that periodically sync up with a remote server (e.g., Facebook, Twitter) tended to be power hungry. The default video player was found to be most power efficient application as it exploits dedicated video decoding hardware on the smartphone.

References

[1] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proc. of the 2010 USENIX*, 2010.

[2] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, 1999.

[3] A. Gutierrez, R. G. Dreslinski,, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Pave. Full-System Analysis and Characterization of Interactive Smartphone Applications. In *Proc. of the IEEE Intl. Symp. on Workload Characterization*, 2011.

[4] Qualcomm. `https://developer.qualcomm.com/develop/mobile-technologies/graphics-optimization-adreno`. 2010.

[5] A. Shye, B. Scholbrock, and G. Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *Proc. of 42nd Intl. Symp. on Microarchitecture*, 2009.

[6] E. L. Sueur and G. Heiser. Slow Down or Sleep, that is the Question. In *Proc. of the 2011 USENIX*, 2011.

[7] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: managing energy as a first class operating system resource. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[8] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proc. of the Hardware/Software Codesign and System Synthesis*, Oct. 2010.