Application-Specific Performance-Aware Energy Optimization on Android Mobile Devices

ABSTRACT

Energy management is a key issue for mobile devices. On current Android devices, power management relies heavily on OS modules known as governors. These modules are created for various hardware components, including the CPU, to support DVFS. They implement algorithms that attempt to balance performance and power consumption.

In this paper we make the observation that the existing governors are (1) general-purpose by nature (2) focused on power reduction and (3) are not energy-optimal for many applications. We thus establish the need for an application-specific approach that could overcome these drawbacks and provide higher energy efficiency for suitable applications. We also show that existing methods manage power and performance in an independent and isolated fashion and that co-ordinated control of multiple components can save more energy. In addition, we note that on mobile devices, energy savings cannot be achieved at the expense of performance. Consequently, we propose a solution that minimizes energy consumption of specific applications while maintaining a user-specified performance target. Our solution consists of two stages: (1) offline profiling and (2) online controlling. Utilizing the offline profiling data of the target application, our control theory based online controller dynamically selects the optimal system configuration (in this paper, combination of CPU frequency and memory bandwidth) for the application, while it is running. Our energy management solution is tested on a Nexus 6 smartphone with 6 real-world applications. We achieve $4-31\%$ better energy than default governors with a worst case performance loss of $< 1\%$.

1. INTRODUCTION

System-on-Chips (SoC) for mobile devices have seen continued improvements in performance with the aid of modules capable of diverse functionalities: GPUs, DSPs for example. The processor performance has experienced a boost over the last few generations due to commensurate improvements in memory technologies. On the software end, the emergence of a variety of applications utilizing the hardware diversity has increased the popularity of mobile devices. Battery technology however, has not kept pace, thereby making battery life one of the top concerns of end users.

In the interest of prolonging battery life, modules in the latest SoCs are equipped with power/energy management solutions. Greedily entering low power states and Dynamic Voltage and Frequency Scaling (DVFS) are the most commonly used techniques. For example, the Linux kernel on Android devices has subsystems called *cpufreq* and *devfreq* to manage power consumption of CPU and other DVFS-capable components respectively. Within these subsystems, modules known as *governors* implement algorithms that determine clock frequencies to be used under different conditions. The governors attempt to strike a balance between performance and power dissipation. For instance, the *interactive* governor, the current default CPU governor on Android devices, will quickly ramp up the frequency when user interactions are detected and will reduce the frequency when there are no interactions.

The stock governors are designed for general purpose usage. Consequently our key observation is that, in the process of improving performance, stock governors result in higher energies for some applications, including popular ones like AngryBirds. We claim that situations like these call for *application-specific controllers*. Additionally, current state-of-the-art governors on Android mobile devices are tailored for power optimization. However, as observed in [1], governors for mobile devices must be designed for minimizing energy with performance constraints and not power because energy consumption is strongly correlated with battery life. Correspondingly, the problem statement addressed in this paper is the following:

Problem: *Choose the minimum energy system configuration while maintaining the performance target.*

Maintaining performance while minimizing energy under dynamic runtime conditions is a complex problem. However, as we elaborate in the later sections, the problem statement can be divided into two parts: (P1) Maintain the performance target and (P2) Minimize the energy consumption. The solution we adopt is implemented in two stages. We profile the application offline (Stage 1) and the online controller (Stage 2) utilizes the profiled data to minimize energy while maintaining performance. In Stage 2,

1) To maintain performance (P1), we use a perfor-

mance regulator. Based on the measured performance, the regulator computes a control signal to meet the target performance.

2) To minimize energy (P2), an optimizer uses the control signal from the regulator and chooses a system configuration from the offline profiled data in order to minimize the energy.

On the latest Android smartphones, hardware modules capable of DVFS have an associated software governor to manage power and/or performance. An important observation to be made at this juncture is that *the software governors work independently of each other.* A few studies ([2, 3, 4]) have shown that independent power/performance control strategies can lead to conflicting policies causing performance and/or power losses. Addressing this problem, researchers have investigated the co-ordinated control of different hardware subsystems on servers [2], SoCs [1] and embedded systems ([5] and references therein).

Proposals which offer energy minimization with performance maintenance by the control of multiple subsystems simultaneously, are either implemented on (1) simulators designed for servers [2] or (2) real physical devices with CPU-only DVFS [6]. Reference [1] discusses CPU and memory DVFS trade-offs for mobile devices using a `Gem5` simulator and SPEC CPU2006 benchmarks as their test cases.

We distinguish ourselves from prior works by the following contributions:

1. We make the observation that for some popular applications, the stock power manager causes excessive energy consumption on a modern Android mobile device.

2. We implement a software controller to minimize the energy consumption of such applications while maintaining a user-specified performance target.

3. Our controller achieves $4-31\%$ energy savings on 6 real-world applications with a worst case performance loss of less than 1%.

4. Unlike default governors that are independent for each subsystem, our control strategy is the co-ordinated control of CPU frequencies and memory bandwidth. Compared to a CPU-only energy minimization scheme, energy savings improved by 53%.

5. Our control strategy can be readily extended to include GPU frequencies, GPU memory bandwidth, network packet rate, etc. Furthermore, it can be implemented on any mobile device capable of DVFS.

The rest of the paper is organized as follows: In Section 2 we present the background and motivation for this work. Section 3 provides a detailed description of our methodology. It begins with a discussion on the offline profiling process for an application, followed by the performance model and the controller design. Section 4 elaborates on the experimental platform, the test applications used and the practical implementation parameters. We compare our test results with the default power management schemes in Section 5. We also discuss important issues related to the control strategy. Section 6 briefly describes research works related to this paper. Finally, Section 7 draws conclusions and points out our future research directions.

2. BACKGROUND AND MOTIVATION

In this section we start with a brief description of DVFS support at the software level for various hardware components on current Android mobile devices. Following that, we highlight the importance of co-ordination among different components for the purposes of power management. We then point out the drawbacks of the existing DVFS governors using results from previous studies as well as our own experiments, which serves as the motivation for this work.

2.1 Linux's *cpufreq* and *devfreq* Subsystems

Linux started to support DVFS for CPU in version 2.6 through a subsystem called *cpufreq* [7]. The design of *cpufreq* follows the principle of separation of policy and implementation. Modules known as *governors* represent policies that determine what DVFS actions to take under what conditions, while device drivers carry out the actual actions. On a typical Linux system, the default *cpufreq* governor is *ondemand* [7]. This governor periodically checks the CPU utilization and if it is above a pre-defined threshold, it increases the frequency to the maximum value. On the other hand, if the CPU utilization is below a specified level, the frequency is reduced gradually. In short, this governor adjusts the CPU frequency based on CPU load. Other commonly available governors include *userspace*, which allows the root user to set the frequency, *performance*, which sets the frequency to the maximum, and *powersave*, which sets the frequency to the minimum.

A similar Linux subsystem, called *devfreq*, performs DVFS on supported devices such as the memory bus, GPU, and so on. Like *cpufreq*, it too has governors to make the DVFS decisions and drivers to carry out actions.

Since the Android system is based on Linux, it has inherited both *cpufreq* and *devfreq* and made adaptations. On a typical Android device, the default CPU governor is *interactive*. It is similar to *ondemand* in that its decisions are also based on CPU load. However, unlike *ondemand*, which samples CPU load at a fixed rate, *interactive* is designed to be more responsive and to ramp up the frequency quickly when needed.

A typical *devfreq* governor for the memory bus is *cpubw_hwmon*, which monitors the memory accesses from the CPU and takes actions accordingly. Other governors include *userspace*, *performance*, and *powersave*, which are similar to their *cpufreq* counterparts.

2.2 Co-ordinated Control

A potential source of ineffectiveness for the existing governors is the lack of coordination among dif-

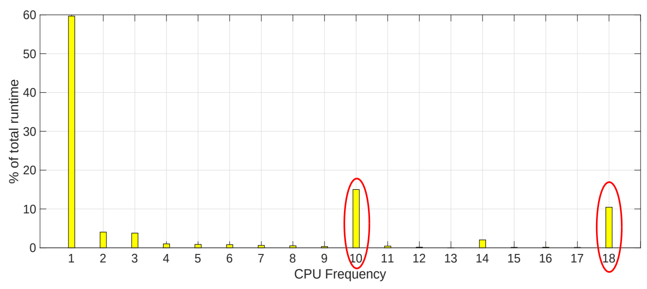

Figure 1: Histogram of CPU frequencies for eBook application

ferent components. Take the *ondemand* CPU governor mentioned above as an example. Its actions are solely based on the load of the CPU, oblivious to the state of other components such as the memory. In [2] the authors point out that independent control policies could conflict and lead to oscillations, greatly reducing the effectiveness in achieving energy savings. They further demonstrate that co-ordinated control of CPU and memory DVFS is a better strategy. Our solution in this paper simultaneously controls both CPU frequency and memory bandwidth.

2.3 Motivation

The current *cpufreq* and *devfreq* governors work well in some cases. However, they do have their limitations.

In [8] the authors compare power consumption of a mobile platform at a set of fixed CPU frequencies and when using two different governors. Four typical usage scenarios are tested: 3G, WiFi, voice call, and ebook reading. They find that the optimal CPU frequency in terms of power consumption is dependent on the use case. In addition, in two of the four cases, the *ondemand* governor consumes more power than *most* of the fixed frequencies. This suggests that for some applications at least, an application-specific DVFS strategy may be a better solution.

We too evaluated the behavior of the default governors for some applications. Fig. 1 shows the histogram of CPU frequencies chosen by the default CPU governor on a Nexus 6 smartphone for an e-book reader application when there is no user interaction such as scrolling or zooming, i.e., when the user is just reading the page. The screen brightness is fixed at the lowest level, WiFi is turned ON and there are no applications running in the background. The x-axis of the figure shows the CPU frequencies from low to high and the y-axis is the percentage of time spent in a given frequency during the test period. We can see that, even though there are no user interactions, the CPUs, under the control of the governor, spend over 10% of the time in the highest frequency, and about 15% of time in a middle frequency (No. 10), as highlighted in the figure. Running at a higher-than-necessary clock frequency results in energy wastage. Based on previous studies and our experiments as well, we have come to the conclusion that the default DVFS governors on the current Android devices, in the process of providing better performance, are not energy-optimal for many applications. This situation motivates us to investigate whether an application-specific approach that can set system configurations, e.g., DVFS, based on the characteristics of specific applications, can lead to higher energy efficiency. In the course of searching for a better solution however, one must keep in mind that performance is a top priority to end users. As a general principle, *energy savings should not be achieved at the expense of performance degradation.*

In this paper we exploit the ineffectiveness of default governors and present a strategy motivated by (1) energy minimization in contrast to power minimization, (2) meeting performance requirements and (3) coordinated control of multiple components.

3. CONTROLLER DESIGN

This section presents the design of our application-specific performance-aware energy optimization solution. As mentioned in Section 1, the solution consists of two stages: offline profiling and online controlling. Both stages are discussed in detail below.

3.1 Offline Profiling

The application-specific aspect of our solution relies primarily on the runtime utilization of offline profiled data of the target application. Before it can be controlled, in the offline profiling stage, we measure the performance and power of a target application under different *system configurations*. For each system configuration, the power and performance data are averaged over three runs for every application tested.

The term *system configuration* means hardware or software settings and combinations thereof, that could impact the performance of applications. Examples include CPU frequency, memory bandwidth, storage parameters, network packet transfer rate, thread scheduling policy and so on. In the context of this paper, we use this term to mean the combination of CPU frequency and memory bandwidth. We emphasize that our solution is not limited to controlling this particular configuration and can be extended to include other configurations mentioned above.

Profiling data of an application are organized in a table, an example of which is shown in Table 1. The performance data are normalized with respect to the value corresponding to the lowest system configuration and is termed *speedup*. The lowest system configuration in this paper refers to the lowest CPU frequency and lowest memory bandwidth of the SoC. Power data, obtained with a Monsoon power monitor [9], are the average power consumption of the entire device during the test period.
There are two issues associated with offline profiling:

1. The number of configurations that require profiling could be rather large in practice.

2. There could be discrepancy between the controller's

#	Config (GHz,MBps)	Speedup	Power (mW)
0	$(0.3, 762)$	1.0	1623.57
1	$(0.3, 1525)$	1.0038	1682.83
2	$(0.3, 3051)$	1.0077	1742.09
⋮	⋮	⋮	⋮
30	$(0.8832, 762)$	1.837	2219.22

Table 1: Sample table with performance and power data profiled offline for AngryBirds application

runtime environment and the profiling environment.

Since we consider the tuple (CPU frequency, memory bandwidth) as the system configuration, exhaustive offline profiling involves running the application for every combination of supported CPU frequency and memory bandwidth. On a Nexus 6 smartphone for example, we have $18 * 13 = 234$ combinations (18 CPU frequencies and 13 memory bandwidths). While a large number of system configurations profiled gives us fine-grained data, it also increases the profiling timespan as well as the online controller's runtime overhead due to the larger search space. Addressing the issue of space explosion, we choose to profile the applications for a maximum of $9*2 = 18$ configurations, i.e., for each alternate CPU frequency with the lowest and the highest memory bandwidths. For each profiled CPU frequency, we then linearly interpolate to get the intermediate data for the rest of the memory bandwidths. Interpolations are not performed along the CPU frequency dimension because we observed that in general, performance and power do not change by a large margin for neighboring CPU frequencies. Although this approach introduces quantization and modeling errors, we show that the controller is robust enough to handle it.

On mobile devices at any given point in time, many applications run in the background albeit most of them are in the "sleep-state". Applications such as e-mail clients perform synchronizations periodically while some applications like Spotify or similar music players continue to run even when they are minimized. Offline data collected for an application under a given background load can be rendered unusable at runtime when the load conditions differ by a large margin. A straightforward method is to profile the application under different background loads. However, the drawback of such an approach is that the profiling overhead increases significantly. Our approach to tackling this issue is to profile the application with a background load, i.e., WiFi ON, e-mail synchronization enabled and Spotify running in the background. We then test the controller performance for heavier and lower background load conditions. Our results show that the profiling data can cover the range of typical load conditions rather well.

Furthermore, we also measure the performance ($\mathbb{R}_{def}$), running time ($\mathbb{T}_{def}$) and average power ($\mathbb{P}_{def}$) of the application under the default governors. The default energy consumption ($\mathbb{E}_{def}$) of the device while the application is running is simply $\mathbb{P}_{def} * \mathbb{T}_{def}$. The default performance, $\mathbb{R}_{def}$, serves as the basis for the *target performance*, which is an input to the online controller. We compare the energy consumption of the device under our control scheme against the default energy $\mathbb{E}_{def}$.

3.2 Online Controlling

The offline profiled data are used by the online controller to run the application in an energy-efficient fashion while at the same time meeting a user-specified performance target.

3.2.1 Overview

The online controller is based on the work presented in [6] and is a feedback control loop as shown in Fig. 2. In accordance with splitting the problem statement into two parts P1 and P2 as described in Section 1, the online controller is further divided into a performance regulator and an energy optimizer.

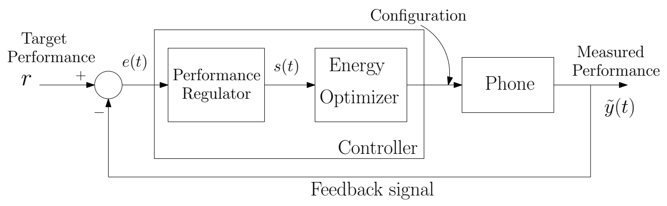

Figure 2: Block diagram of feedback controller

With reference to Fig. 2, the feedback controller is implemented in four steps. (1) Given a target performance of the application r and the measured performance of the system $\tilde{y}(t)$, compute the error $e(t) = r - \tilde{y}(t)$. (2) Calculate a control signal $s(t)$ based on $e(t)$ (3) Given $s(t)$ determine the system configuration which minimizes the energy consumption while maintaining the target performance (4) Apply the new system configuration and measure the performance. The closed-loop system is described by the repeated applications of steps (1) $\rightarrow$ (2) $\rightarrow$ (3) $\rightarrow$ (4).

The target performance r is based on the default performance $\mathbb{R}_{def}$, i.e., $r = \alpha * \mathbb{R}_{def}$ where $\alpha \in (0, 1]$. In our experiments, we set $\alpha = 1$. We also mention that one could vary α to trade performance for energy savings. At runtime, the energy optimizer, based on the control signal input $s(t)$ and the offline profiled data chooses the configuration for the device which would consume the least energy and meet the performance target. It is important to note that the term "least energy" is with respect to the offline profiled data. The feedback controller is implemented at fixed discrete time intervals until the application terminates. In what follows we give more details about the performance metric, the performance regulator, and the energy optimizer.

3.2.2 Performance Metric

Performance of an application can be quantified in many ways. Execution time is perhaps the most commonly used metric. Frames per second can be used for video playing applications. Other metrics include number of jobs completed per unit time, task latency, and so on. Android applications are distributed in a compressed format (`apk`) which contains partially to completely obfuscated code. Requiring the developer to implement modifications in the source code so as to report application performance periodically is infeasible. In this work, our objective is to obtain information about the progress of the application without any source code modifications. Fortunately, modern micro-processors, including SoCs used in Android mobile devices, generally possess a performance monitoring unit (PMU). In this paper, we use Giga-Instructions-Per-Second (GIPS) obtained from the PMU as our performance metric. We consider GIPS as a good metric because it is highly correlated with the execution time. We also note that GIPS has been used as a performance metric in earlier works as well (see [10]).

3.2.3 *Performance Regulator*

The goal of a performance regulator is to reduce the error between the target performance and the measured performance to zero, i.e., $e(t) = (r - \tilde{y}(t)) \to 0$. We model the performance of the system as

$$\tilde{y}(t) = s(t-1) * b(t-1) \tag{1}$$

where $b(t)$ is the base speed of the application and $s(t)$, the control signal, is the *speedup* with respect to $b(t)$. Base speed $b(t)$ is defined as the speed of the application when the least amount of system resources are consumed. Feedback controllers ([11]) can be designed with fixed gains or adaptive gains. We choose an adaptive gain integral controller in order to accommodate runtime variations, inaccuracies in measurement, modeling errors etc. References [12, 13, 14] have shown the practical feasibility of using adaptive gain integral controllers in a variety of computing environments. The performance regulator (adaptive gain integral controller) computes a required speedup (control signal) as follows:

$$s(t) = s(t-1) + \frac{e(t-1)}{b(t-1)} \tag{2}$$

where the adaptive gain is encoded by $\dfrac{1}{b(t-1)}$. The required speedup $s(t)$ is computed based on the history of $e(t)$ which is why Eqn. 2 is called an "integrator". We recommend the reader to refer to [15] for details on derivation and stability proofs. Different applications can have different base speeds. For example, on the Nexus 6 smartphone whose lowest possible configuration is $(300MHz, 762MBps)$, the base speed of AngryBirds is 0.129GIPS whereas for a Video Converter application the base speed is 0.471GIPS. To ensure that the controller can track these changes automatically, based on the work in [6], we use a Kalman filter [16] to continuously estimate the application base speed $b(t)$. Formal convergence proofs for the Kalman filter can be found in the appendix of [6]. In our paper, a discrete-time equivalent of Eqn. 2 is implemented.

3.2.4 *Energy Optimizer*

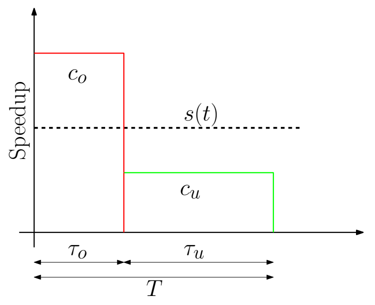

Figure 3: Pictorial representation of the energy optimization

Once the required speedup $s(t)$ is calculated by the performance regulator, the energy optimizer determines the energy-optimal system configuration that meets the performance requirements (see Fig. 2).

Given a set of configurations $\mathbb{C} = \{0, 1, \ldots, C-1\}$ where $C = |\mathbb{C}|$ is the total number of available configurations, each configuration $c \in \mathbb{C}$ is associated with an average speedup s_c and an average power p_c. Suppose that we are given a time horizon T during which the system has to maintain a performance of r GIPS, the optimization problem can be encoded as follows:

$$\min \sum_{c=0}^{C-1} \tau_c \cdot p_c \tag{3}$$

$$s.t. \sum_{c=0}^{C-1} \tau_c \cdot s_c \cdot b(t) = r \cdot T \tag{4}$$

$$\sum_{c=0}^{C-1} \tau_c = T \quad 0 \le \tau_c \le T \quad \forall c \in \mathbb{C} \tag{5}$$

where τ_c is the duration for which a configuration c is applied by the energy optimizer. Equations 3 - 5 represent a linear programming problem with equality constraints. Eqn. 3 is the objective which captures the energy minimization, Eqn. 4 represents the performance constraint and Eqn. 5 is a constraint on the time horizon. The solution to the above problem determines a set of optimal system configurations $C_{opt} \subseteq \mathbb{C}$ and the corresponding duration $\tau_{c_i}, \forall c_i \in C_{opt}$ such that it minimizes the energy consumed by the system for a time period of T seconds while maintaining a performance r. In [6] it is shown that, there exists an optimal solution to Eqn. 3 with at most two non-zero τ_cs. The energy optimizer therefore selects at most 2 configurations c_u and c_o such that $s_u \le s(t) < s_o$ and $\tau_u + \tau_o = T$. The subscripts u and o represent "under the required speedup" and "over the required speedup" respectively. This is pictorially described in Fig. 3. Since there are at most C configurations, the runtime complexity of the energy minimization is $O(C^2)$.

4. EXPERIMENTAL SETUP

In this section we describe the experimental setup for evaluating our solution, including the test environment

CPU Frequency (GHz)				Mem Bandwidth (MBps)			
1	0.3000	10	1.4976	1	762	10	8056
2	0.4224	11	1.5744	2	1144	11	10101
3	0.6528	12	1.7280	3	1525	12	12145
4	0.7296	13	1.9584	4	2288	13	16250
5	0.8832	14	2.2656	5	3051		
6	0.9600	15	2.4576	6	3952		
7	1.0368	16	2.4960	7	4684		
8	1.1904	17	2.5728	8	5996		
9	1.2672	18	2.6496	9	7019		

Table 2: List of CPU frequencies and memory bandwidths on Nexus 6

and the applications tested. We also point out some implementation challenges.

4.1 Platform and Tools

We use a Nexus 6 (*N6*) smartphone running Android Marshmallow 6.0 with Linux kernel v3.10 as our experimental platform. The Android OS running on *N6* is downloaded from the Android Open Source Project (AOSP)[17] and built in the *userdebug* mode. The *N6* has a Qualcomm Snapdragon 805 SoC, which has a quad-core Krait 450 CPU, an Adreno 420 GPU, and 3 GB of RAM. Each CPU core supports 18 clock frequencies that can be modified dynamically. The main memory has 13 distinct bandwidths supported. Table 2 lists all the CPU frequencies and memory bandwidths.

As discussed in Section 2.1, DVFS of CPU and memory in Linux is controlled by the *cpufreq* and *devfreq* subsystems respectively. For our purposes, we need to set the CPU frequency and memory bandwidth to certain values. This is achieved by writing to pertinent files in the `sysfs` [18] to first set the governors to *userspace*, and then to set the CPU frequency and memory bandwidth. Since applications can have multiple threads running on any one of the four CPUs, we set all four CPUs to the same frequency in our experiments.

In order to measure the power consumed by the device, we use the Monsoon Power Monitor [9]. The sampling frequency of the power monitor is 5KHz. We use `adb` (Android Debug Bridge [19]) to communicate with the phone via the USB interface. A USB interface between the host computer and the phone results in automatic battery charging. We disable USB charging for all our experiments to prevent power measurement bias. During all the experiments, the phone is laid flat on a desk with a fixed display brightness and is not disturbed. We disable a kernel module called `mpdecision` during our experiments to prevent CPU hot-plugging which can lead to inaccurate measurements. A kernel compilation feature which causes CPU frequency boost on a screen touch event is also disabled to help record reliable power data.

Finally, performance is measured by reading the hardware counters with the `perf` tool [20]. We read the instruction counter and derive the GIPS metric.

4.2 Implementation Challenges

In contrast to previous works, we test the controller on a physical device with real applications and runtime conditions. We list a few challenges faced during the course of this work and the solutions adopted.

Previous work ([6]) required source code modifications to enable the controller to monitor the application performance. Specifically, the "application being controlled" reports its performance periodically to the controller. We use GIPS derived from a PMU counter as mentioned above which does not require application developers to modify their code.

However, a commercial phone does not come preloaded with the `perf` tool, neither does it provide root access to the Linux kernel. We therefore built the *userdebug* version of Android Marshmallow 6.0 along with `perf`. This enables (1) measuring performance at runtime and (2) changing CPU frequency and memory bandwidth. The `perf` tool on the *N6* has the lowest sampling period of 100ms. Furthermore, the computation overhead at this sampling period is 40%. Therefore, we choose a control cycle duration of 2 seconds for all our experiments, i.e., with reference to Eqn. 3, $T = 2$. We discuss the overhead of `perf` and the controller in Section 5.1.1.

Unlike commercial Intel and AMD processors, the Snapdragon 805 SoC does not support hardware power and energy counters. Moreover, our current setup allows us to record the power consumption of the entire device only. Although the control algorithm ideally requires only the power consumed by the CPU and the memory, we rely on the robustness of the controller to handle these modeling inaccuracies.

A typical desktop/server class processor supports multiple processes running in parallel. While ARM based SoCs do support the same, the process consuming most of the resources in an Android device corresponds to the application being currently displayed on the screen. Background application threads are in the "sleep state" in general, woken up periodically depending on the nature of the application. Following suit, our strategy is to control the application *only* while it is running in the foreground.

4.3 Applications

A set of 6 real world applications is chosen where each application demonstrates unique characteristics. The applications are individually described below.

VidCon is a video converter application which uses the `FFmpeg` library [21] to convert videos to different formats. For our experiments, we choose a fixed size `mp4` HD video and use the default conversion settings.

MobileBench [22] is an established browser benchmark based on BBench [23]. The benchmark loads a collection of websites whose content is available in the phone memory. It offers automatic horizontal and vertical zooming and scrolling as well. The Chrome browser application on the phone is used for running the tests.

AngryBirds is a gaming application with over 100 million downloads on Android alone. We choose this as a representative gaming application to test the controller performance. The game is manually played for 200 seconds during our experiments.

Application Name	Performance	Energy
VidCon	−0.4%	25.3%
MobileBench	4.1%	15.3%
AngryBirds	0.6%	14.9%
WeChat Video Call	−0.4%	27.2%
MX Player	0.0%	4.2%
Spotify	−0.4%	31.6%

Table 3: Summary of performance difference and energy savings obtained by the controller

WeChat is an Internet based text messaging, voice communication and video conferencing application. With over 700 million active users, it is among the most downloaded applications in the communication application segment. We choose the video conferencing feature of this application and initiate a 100-second long video call for our experiments.

MX Player is a video player application which can play videos encoded in a variety of formats and has over 100 million downloads. It also supports hardware accelerated decoding and high speed rendering for ARM NEON compliant processors. We test the controller performance when playing a 137-second long HD video.

Spotify is an audio, podcast and video streaming application with over 100 million subscribers. We use a premium version of the application which avoids advertisements between songs. This application is tested for 100 seconds with songs being changed every 20 seconds.

5. EVALUATION

In this section we present test results of our energy management scheme described in Section 3 against the default settings on the *N6*. We provide detailed analysis of the results and discuss a few important issues, including (1) application scope, (2) the effect of varying background application loads on the controller performance, and (3) comparison with a CPU-only DVFS strategy.

5.1 Results and Analysis

Table 3 summarizes the performance and energy savings achieved by our controller as compared with the default governors. Each number is the average of three runs. The background load used for the results in Table 3 is the same as discussed in Section 3.1. In what follows, we refer to this background load as *baseline load*. VidCon, MobileBench browser benchmark and MX player are deadline critical. Even though the controller measures performance for these applications in GIPS, performance numbers in Table 3 are based on execution time. For the rest of the applications, performance in Table 3 is measured in GIPS.

We can see that for all the applications tested, our controller is able to save energy while meeting the performance target. A worst case performance degradation of 0.4% is observed with our control technique. At the same time, compared to the default, we save 14.9 – 31.6% of energy with 5 out of the 6 applications and 4.2% with MX Player. The results in Table 3 clearly demonstrate the effectiveness of our application-specific approach in achieving substantial energy savings while maintaining performance.

To a large extent, the effectiveness of our approach is due to the co-ordinated control strategy. The default CPU governor `interactive`, changes the CPU frequency based on the CPU load. The default memory bandwidth governor `cpubw_hwmon`, on the other hand, monitors the L2 cache read and write events to decide the required bandwidth. Both governors work independently and the results we obtain demonstrate the drawbacks of such an approach.

To help analyze and understand the experimental results, in Figs. 4 and 5 we compute the percentage of time spent in each of the 18 CPU frequencies and 13 memory bandwidths during the application execution, and compare the choices made by the default governor and our controller. Fig. 4 shows some of the key characteristics of the default CPU governor. Firstly, in all 6 cases, it spends a considerable amount of time (12.7 − 27.9%) at CPU frequency 10 (1.4976 GHz). Fig. 5 illustrates the characteristic behavior of the default bandwidth governor which implements an exponential back-off algorithm while reducing the bandwidth. The offline profiled performance data for AngryBirds, MX Player and Spotify show an improvement of less than 5% for frequencies between 5 and 10 whereas power increases by more than 36%. Secondly, in 3 out of the 6 cases, the highest frequency is used for a significant amount of time (9.7 – 57.3%). With our approach, in 5 out of the 6 cases, the high frequencies are not included in the profiling table supplied to the controller, based on the performance/power characteristics of the profiled data.

We observe that in Fig. 4 (b), (c), and (e), with the default governor, the CPUs are at frequency 1 for the largest amount of time, whereas our controller selects higher frequencies. Intuitively, this should lead to higher energy consumption by our controller. But the results in Table 3 show that energy consumption with the controller is lower than default in all 3 cases for the same performance. This phenomenon is a result of the following: (1) The controller is designed to *maintain* a performance target (2) The controller trades higher CPU frequencies against increasing the bandwidth (see Fig. 5) and (3) In our current solution, the smallest duration for the CPUs to stay at any given frequency is 200ms. Choosing frequency No. 1 even for a duration of 200ms impacts the performance heavily. In fact, for MobileBench and MX Player, the lower frequencies are not even included in profiling data provided to the controller. In Fig. 4 (b), (c), and (e), even though it appears that with the default governor the CPUs spend most of the time in the lowest frequency, it should be noted that this is an accumulated time. The CPUs spend short durations (of the order of 10s of milli-seconds) in this frequency before moving on to higher frequencies. The conclusion we can draw is that lower CPU frequencies may reduce power consumption but it does not translate to lower energy. Similar or

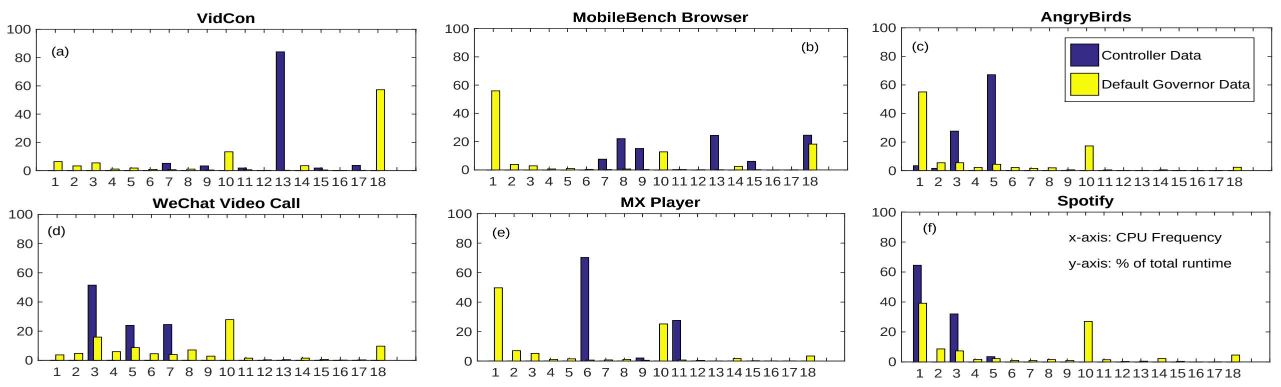

Figure 4: Histogram of CPU frequencies: controller vs. default

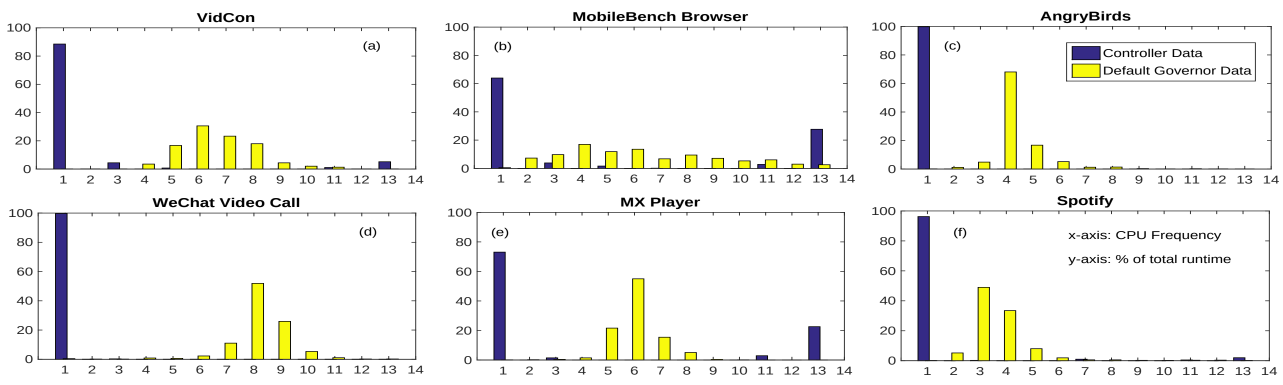

Figure 5: Histogram of memory bandwidths: controller vs. default

better performance with lower energy can be attained by choosing higher CPU frequencies. Now we discuss the 6 applications individually.

VidCon has a uniform power and performance profile during its execution. Fig. 4 (a) shows that the default governor spends nearly 60% of the time in the highest CPU frequency and takes 59 seconds to convert a sample video. The controller, however, chooses a much lower frequency (No. 13) for 80% of the time and is able to convert the same video with 25.3% less energy. The time it takes to convert the video is only 0.4% or about $0.24s$ longer, hardly noticeable by a human user. For this application we use CPU frequencies 7-18 because frequencies below No. 7 resulted in a performance drop of over 50%.

MobileBench browser benchmark, unlike VidCon, has a varying power and execution profile. For a fixed CPU frequency, we notice an average increase of 7% in the relative speedup between the lowest and highest memory bandwidths. Due to the zooming and scrolling actions, the performance in GIPS too shows a steady increase as CPU frequencies are increased. However, the data used by the online optimizer is restricted between CPU frequency 7 and 18 (See Fig. 4). The justification is similar to VidCon in that, when the CPU frequency is fixed at No. 7, the performance is 30% worse than the default. Any lower frequency would incur a larger performance loss resulting in a lower user experience. The controller chooses CPU frequency 18 for a duration longer than the default governor, yet achieves a 15.3% improvement in energy. Although this seems counter-intuitive, the reason for this phenomenon is that the objectives of the default governor and our controller are orthogonal. While the default governor tries to maximize performance whenever possible, the aim of our controller is to maintain a fixed performance. The default governor chooses to assign a higher CPU frequency when it senses a load increase whereas our controller only assigns a higher frequency when it senses a performance drop. The energy savings we achieve is on account of a shorter runtime.

With AngryBirds, the controller is provided with a smaller CPU frequency range because the offline profiling data shows that performance (in GIPS) does not improve beyond CPU frequency No. 5 but power consumption increases steadily for higher frequencies. Compared with the default governor, which spends nearly 20% of the time in frequency No. 10 and some amount of time in the highest frequency, our controller selects frequencies 3 and 5, as shown in Fig. 4 (c). The end result is a performance (in GIPS) that is slightly (0.6%) better with an energy saving of 19.3%. AngryBirds involves the GPU for image rendering, but, despite the fact that GPU frequency is not part of the controlled system configuration, we observe no change in the game experience when our controller is deployed. Moreover, the default governor chooses a higher frequency when advertisements get loaded between individual levels, resulting in higher power consumption[1].

When profiling WeChat video call, we find that for CPU frequencies 1 and 2, the camera fails to record and transmit video reliably and hence we exclude them from the power and speedup table. Additionally, the performance (in GIPS and subsequently video quality) does not show significant improvement beyond CPU frequency 7. However Fig. 4 (d) shows that frequencies 10 and 18 get chosen for close to 40% of the time by the default governor. The controller is able to provide comparable performance by choosing lower CPU frequencies (3, 5, and 7) with No. 3 being used for over 50% of the time. This results in a significant energy saving of 27.2% compared with the default governor.

MX Player is not CPU intensive because it performs video decoding using a hardware decoder and bypasses the GPU to render the image on the screen. MX player has a performance vs. CPU frequency profile similar to WeChat in that, beyond frequency 5, the performance varies very little (0.4%). Furthermore for frequencies between 1 and 4, the video does not play smoothly regardless of the memory bandwidth chosen. Hence we do not include CPU frequencies 1 - 4 in the offline profiling table. Due to the nature of the application and the fact that our controller can only manipulate CPU and memory bandwidths, we can only save 5% energy. The implication is that the default governor indeed does a good job for this application.

Spotify is another case where a limited range of CPU frequencies is included in the profiling table. In fact, only 3 frequencies on the low end are used: frequencies 1, 3, and 5. We note that even when the CPU frequency is fixed at the lowest, the audio quality does not degrade. However, the default governor, as shown in Fig. 4 (f), spends a considerable amount time in the much higher frequencies 10 (27%) and 18 (4.6%). In contrast, our controller spends 64.5% of time in the lowest frequency and 32% in frequency No. 3. Compared with the default governors, the controller saves 31.6% energy with a minor performance loss in GIPS of 0.4%.

5.1.1 *Controller Overhead*

As mentioned in Section 3 the controller consists of three parts: (1) measurement (2) performance regulation and energy optimization and finally (3) actuation. Accordingly, we present the overhead for each part of the controller. The controller measures performance twice in each control cycle. On an average, the measurement is done every 1s when the control cycle duration is 2s. The `perf` tool takes 1.04s on average, i.e., a 4% computation overhead, to report the measurement. The power consumption overhead for `perf` at a sampling period of 1s is 15mW, a relatively negligible number. The execution time of the performance regulator and the energy optimizer together is less than 10ms per control cycle with an average power consumption of 25mW. Changing the CPU frequency and memory bandwidth requires writing into the appropriate `sysfs` files. The CPU frequency transition latency is of the order of micro-seconds whereas the shortest duration between frequency changes in our controller is 200ms. Finally, the power overhead for changing CPU frequencies is 14mW. In summary, the implementation overhead for our controller is negligible even when the number of system configurations is large.

[1]Advertisements consume close to 0.5W of power and an application with several ads will result in rapid battery discharge.

5.2 Application Scope

Not all applications are amenable to our current solution. We identify two types of applications that are not well suited for the current strategy.

The first type includes applications for which the default CPU governor either selects the lowest frequency most of the time due to low CPU requirements or the highest frequency most of the time due to CPU-intensive computations. For the former case it is hard to obtain additional energy savings through CPU DVFS and for the latter it is hard to save more energy without performance degradation. For such applications, other components of the system such as network packet transfer rate etc. should be explored to save energy. Our controller framework, as mentioned in Section 3, is generic enough to be able to control other parameters.

The second type includes applications with multiple rapidly varying phases (e.g. MobileBench browser benchmark), i.e., the application has very different CPU, memory, or I/O characteristics at different points in time. These applications pose a few very challenging problems. Firstly, how do we define and identify application phases? This problem has been studied earlier on desktops/servers [24] and for simulators [25]. For example, in [24] six phases were defined based on the ratio of "memory access / uop". We have yet to study whether this kind of metric can be used to classify application phases on our target platform. A practical concern is the lack of OS and/or hardware support for PMU counters. More serious problems are caused by the fact that the duration of phases could be very short. In such situations, experiments show that PMU-based performance measurements could have high variations, which in turn could mis-guide the controller. Furthermore, the shorter the duration, the more difficult it is for the controller to catch up. Phase prediction, as proposed in [24], might help, but is only one step towards addressing these problems.

5.3 Effect of Different Background Loads

Section 3.1 discusses the issue of discrepancy between controller runtime environment and the profiling envi-

App Name	Performance (%)			Energy (%)		
	BL	NL	HL	BL	NL	HL
VidCon	0.8	0.2	-8.0	25.3	28.0	11.4
MobileBench	4.0	-3.5	-2.0	15.3	-4.9	4.6
AngryBirds	0.6	1.0	-2.0	14.9	12.8	10.0
WeChat Video Call	-0.4	2.0	3.6	27.2	19.4	27.0
MX Player	0.0	0.0	0.0	5.0	2.9	5.0
Spotify	9.3	-1.7	-1.3	31.6	7.2	6.0

Table 4: Summary of performance difference and energy savings obtained for the tested applications under Baseline Load (BL), No Load (NL), Heavier Load (HL) conditions

ronment. In the following we evaluate the controller performance under different loading scenarios. The controller is tested under two different runtime conditions: (1) No-Load (NL) and (2) Heavier-Load (HL) while utilizing the offline profiling data and target performance obtained under the baseline load (BL).

In the NL condition only the application being controlled runs on the phone. In HL, a few more applications as compared to BL are opened but minimized. The background applications are: Gallery, eBook Reader, Chrome browser, FaceBook, e-Mail client, MX player and Spotify. WiFi is turned ON for both loading scenarios. We note that the most significant difference among the different loads is the memory usage. The amount of free memory is 500 MB, 1 GB, and 134 MB, for BL, NL and HL respectively. In contrast, the corresponding CPU loads as indicated by the file `/proc/loadavg` are similar: 6.3, 6.7, and 6.6 respectively.

Table 4 shows the controller's performance and energy results in the three different loading conditions. In 4 cases, i.e., VidCon, AngryBirds, WeChat, MX Player, the controller performs relatively well in terms of energy savings when running under an environment different from the profiled environment. VidCon under HL test condition experienced a performance loss of 8% but still achieved 11.4% energy savings. The controller performs the best for WeChat in NL and HL, saving 19% and 27% energy respectively.

Spotify displays a significant decrease in energy savings in both NL and HL. On further analysis, we find that in NL and HL, the default governor uses CPU frequency No. 10 less than 10% of the runtime as compared to 25% with the baseline load. This directly translates to lower overall power consumption of 1.43W in NL and HL, versus 1.7W with the baseline load. The average power consumed by Spotify with the controller is 1.3W for all the loading cases which results in the varying energy savings shown in Table 4.

MobileBench browser has rapidly varying GIPS and power data on account of multiple websites being loaded in quick succession. Due to the lower bound on the time taken to measure application performance (200ms), the controller is unable to respond to these rapid variations. While the average power in the NL case is similar to the average power consumed by the default governor, the performance loss of 3.5% leads to the excessive energy consumption by the controller.

Application Name	Performance	Energy
VidCon	2.8%	13.1%
MobileBench	−2.9%	7.6%
AngryBirds	−2.6%	9.6%
WeChat Video Call	4.7%	22.3%
MX Player	0.0%	0.4%
Spotify	3.3%	33.3%

Table 5: Summary of performance difference and energy savings obtained by the CPU-only DVFS controller

Although in a majority of cases the profiling data obtained under baseline load can be used to achieve good results in different load conditions, we observe that better results can be achieved if the profiling condition closely matches the runtime environment. As an example, we re-profiled MobileBench for the NL case and the controller is re-tested, this time with a new target performance obtained from the offline data. The controller now saves 11.1% energy with no performance loss. A possible approach is to profile the application under a few different background loads and let the controller select the appropriate offline data by measuring the background load at runtime.

We also note that the performance and power data for NL has the same trend as that for BL but with a small increase in the absolute value. We envision a method which involves a power and performance model which uses the system load as the variable parameter. At runtime, the controller can track the background load and, using the models, generate power and performance data for different configurations. Such an approach would not require additional profiling thereby expanding the scope of our method. We leave these extensions to future work.

5.4 Comparison with CPU-only DVFS

To evaluate the effectiveness of co-ordinated control of CPU frequency and memory bandwidth, we created another version of the controller which controls only the CPU frequencies and allows the memory bandwidth to be controlled by the default governor, i.e., `cpubw_hwmon`. The controller does not communicate with the default memory bandwidth governor and hence takes decisions in an independent and isolated manner.

For this controller, we re-profile the applications with CPU frequency set to fixed values while memory bandwidth is left in the control of the default governor. For each application, the same set of CPU frequencies as in the co-ordinated controller case is selected. Table 5 lists the energy savings and performance of the 6 tested applications when only the CPU frequencies are controlled. Excluding MX Player which practically does not save energy, on an average, we observe a 53% increase in energy consumption as compared to the co-ordinated control of CPU frequency and memory bandwidth. For WeChat and Spotify, the CPU frequencies chosen and their durations are similar. For other applications however, the default bandwidth governor selects a higher-than-necessary bandwidth for over 60% of the application runtime thus resulting in a higher power consumption. In AngryBirds, for example, the bandwidth governor increases the bandwidth to the highest whenever advertisements are loaded between game levels, which results in a peak power of 6W. In general, we observe that CPU-BW DVFS controller trades higher CPU frequency over higher bandwidth at the same CPU frequency which is a direct consequence of the profiling table (see Fig.5). For example with Mobilebench, the average power and performance for the pair of CPU frequency and memory bandwidth $(7, 13)$ is $(2.128, 2.687)$ while the same parameters for the pair $(11, 1)$ are $(2.125, 2.9705)$. The controller chooses (11,1) rather than (7,13) because for the same power consumption the performance of (11,1) is much higher. This is exactly why our controller chooses the bandwidth No. 1 for over 60% in all 6 test cases.

6. RELATED WORK

Research works that are most closely related to our work are those that optimize energy consumption under performance requirements. We briefly describe a few works selected from the literature in this section.

Reference [26] presents a model-based DVFS governor for Android systems. At first, offline profiling is performed on a set of benchmarks and for each benchmark the *critical speed* (CS), i.e., the energy-optimal CPU frequency is obtained along with the corresponding memory access rate (MAR), which in turn is obtained from the hardware performance monitoring unit. Statistical methods are then used to derive a model for CS with regard to MAR. This model is called the MAR-based CS Equation, or MAR-CSE. A DVFS governor is created that uses MAR-CSE to select the optimal CPU frequency based on the runtime MAR values. This approach is application-agonostic in the sense that it is independent of the running application. Furthermore, it is designed to optimize energy without considering performance. Our method is application-specific and strives to minimize energy while maintaining a certain level of performance.

In [27], the authors propose an energy-saving scheme for video decoding on the Android platform. This involves a two-level table structure: the Frame Decoding Complexity History Table (FDC-HT), which is based on the Global Phase History Table in [24], and the Decoding Time Table (DTT). The FDC-HT saves the *decoding complexity* history of decoded frames and each entry has a corresponding DTT which stores the CPU frequencies and decoding times for that complexity level. The recent history is used to predict the complexity of the next frame. After this step, the DTT is consulted to find the lowest frequency that meets the decoding deadline. Our method, although application-specific, is not limited to one particular type of application and we control more system components than just the CPU.

The CoScale method presented in [2] shares some features with our work. Both our works attempt to minimize energy consumption while meeting a performance target, and both control DVFS for CPU and memory system simultaneously. However, there are important differences. Firstly, CoScale is application-agnostic while we take an application-specific approach. Secondly, CoScale uses both a performance and a power model, while we use profiling data obtained in advance. Thirdly, CoScale uses a gradient-descent heuristic to select the optimal configuration while our optimization is based on linear programming. Finally, CoScale targets servers while we target mobile devices.

The POET system [6] is perhaps the most relevant to our current work. Its overall strategy, like ours, is to minimize energy consumption of an application while attempting to meet its performance requirement. The system requires two inputs before it starts: a performance target, and performance and power data for different system configurations. At runtime, it repeatedly measures the actual performance, and uses feedback control and linear programming to select energy-optimal configurations that meet the performance target. POET consists of a C library and a runtime system, and is designed for traditional embedded systems with soft realtime constraints. Because of the diversity of such systems, one of the key design goals of POET is portability. The problem it tries to solve is to create an application- and platform-independent resource allocation framework. Our focus is quite different. Although our control and optimization scheme is an adaptation from POET, we target Android mobile devices, which have a much more diversified software space than traditional embedded systems, where the software typically performs just one or very few tasks. Our goal is to provide, under performance constraints, more energy savings for certain applications than the default system. The application-specific approach we take is primarily due to the target software environment.

7. CONCLUSION AND FUTURE WORK

In this paper we make the observation that the default DVFS governors on current Android mobile devices are designed for general-purpose usage, focus on power savings, and are in general not energy-optimal for many applications. We establish the need for investigating an application-specific energy optimization strategy and stress that any energy optimizer should be mindful of performance impacts. Furthermore, we point out the advantage of co-ordinated control among different components such as CPU and memory. We then present a detailed description of our application-specific, performance-aware energy optimization solution targeting Android devices. Our solutions have been implemented on a Nexus 6 smartphone. Tested with 6 real-world applications, including highly popular ones, we have achieved $4-31\%$ energy savings with a worst-case performance loss of less than 1%. Our next steps are to include GPU frequencies, network packet rate, etc. into the control system framework and develop a model based performance and power data generation scheme to accommodate variable loads.

8. REFERENCES

[1] R. Begum, D. Werner, M. Hempstead, G. Prasad, and G. Challen, "Energy-performance trade-offs on energy-constrained devices with multi-component dvfs," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pp. 34–43, IEEE, 2015.

[2] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "Coscale: Coordinating cpu and memory system dvfs in server systems," in *45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 143–154, 2012.

[3] I. Paul, W. Huang, M. Arora, and S. Yalamanchili, "Harmonia: Balancing compute and memory power in high-performance gpus," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 54–65, IEEE, 2015.

[4] B. Su, J. Gu, L. Shen, W. Huang, J. L. Greathouse, and Z. Wang, "Ppep: Online performance, power, and energy prediction framework and dvfs space exploration," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 445–457, IEEE Computer Society, 2014.

[5] C. Ykman-Couvreur, P. Avasare, G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria, "Linking run-time resource management of embedded multi-core platforms with automated design-time exploration," *IET Computers & Digital Techniques*, vol. 5, no. 2, pp. 123–135, 2011.

[6] C. Imes, D. H. Kim, M. Maggio, and H. Hoffmann, "Poet: A portable approach to minimizing energy under soft real-time constraints," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 75–86, 2015.

[7] V. Pallipadi and A. Starikovskiy, "The ondemand governor," in *Proceedings of the Linux Symposium*, vol. 2, pp. 215–230, 2006.

[8] P. T. Bezerra, L. A. Araujo, G. B. Ribeiro, A. C. d. S. B. Neto, A. G. Silva-Filho, C. A. Siebra, F. Q.B. da Silva, A. L. Santos, A. Mascaro, and P. H. Costa, "Dynamic frequency scaling on android platforms for energy consumption reduction," in *Proceedings of the 8th ACM Workshop on Performance Monitoring and Measurement of Heterogeneous Wireless and Wired Networks*, PM^{2}HW^{2}N '13, pp. 189–196, 2013.

[9] "Monsoon power monitor." `https://www.msoon.com/LabEquipment/PowerMonitor/`.

[10] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: maximizing performance for a given power budget," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 347–358, 2006.

[11] F. Golnaraghi and B. Kuo, "Automatic control systems," *Complex Variables*, vol. 2, pp. 1–1, 2010.

[12] X. Chen, H. Xiao, Y. Wardi, and S. Yalamanchili, "Throughput regulation in shared memory multicore processors," in *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, pp. 12–20, IEEE, 2015.

[13] K. Rao, W. Song, S. Yalamanchili, and Y. Wardi, "Temperature regulation in multicore processors using adjustable-gain integral controllers," in *2015 IEEE Conference on Control Applications (CCA)*, pp. 810–815, IEEE, 2015.

[14] T. Patikirikorala and A. Colman, "Feedback controllers in the cloud," in *Proceedings of APSEC*, 2010.

[15] N. Almoosa, W. Song, Y. Wardi, and S. Yalamanchili, "A power capping controller for multicore processors," in *2012 American Control Conference (ACC)*, pp. 4709–4714, IEEE, 2012.

[16] L. Cao and H. M. Schwartz, "Analysis of the kalman filter based estimation algorithm: an orthogonal decomposition approach," *Automatica*, vol. 40, no. 1, pp. 5–19, 2004.

[17] "Android open source project." `https://source.android.com/index.html`.

[18] P. Mochel, "The sysfs filesystem," in *The Ottawa Linux Symposium*, pp. 321–334, 2005.

[19] "Android debug bridge." `https://developer.android.com/studio/command-line/adb.html`.

[20] "perf: Linux profiling with performance counters." `https://perf.wiki.kernel.org/index.php/Main_Page`.

[21] "Ffmpeg library." `https://ffmpeg.org/about.html`.

[22] D. Pandiyan, S.-Y. Lee, and C.-J. Wu, "Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite-mobilebench," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pp. 133–142, IEEE, 2013.

[23] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-system analysis and characterization of interactive smartphone applications," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pp. 81–90, IEEE, 2011.

[24] C. Isci, G. Contreras, and M. Martonosi, "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 359–370, 2006.

[25] J. Lau, S. Schoemackers, and B. Calder, "Structures for phase classification," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 57–67, 2004.

[26] W.-Y. Liang and P.-T. Lai, "Design and implementation of a critical speed-based dvfs mechanism for the android operating system," in *The 5th International Conference on Embedded and Multimedia Computing (EMC)*, pp. 1–6, 2010.

[27] W.-Y. Liang, M.-F. Chang, Y.-L. Chen, and C.-F. Lai, "Energy efficient video decoding for the android operating system," in *IEEE International Conference on Consumer Electronics (ICCE)*, pp. 344–345, 2013.