Energy Management of Applications with Varying Resource Usage on Smartphones

Anway Mukherjee, *Student Member, IEEE,* and Thidapat Chantem, *Senior Member, IEEE*

***Abstract*—The split-screen mode in smartphones allows for the simultaneous side-by-side execution of multiple applications, which permits multitasking and improves users' experience. However, such technology results in simultaneously running multiple foreground processes, which increases the power consumption of a smartphone and reduces its battery lifetime. We present an integrated system-level resource management framework that aims to minimize the total energy consumption of a smartphone with negligible impact on the quality of service (QoS) of applications whose resource usage characteristics are not precisely known offline or vary over time. Our proposed solution (1) leverages applications' offline profiles to detect instantaneous phase changes (i.e., dynamic changes in resource usage patterns) of the workload of a given application at runtime, and (2) adaptively adjusts both the voltage and frequency settings of the processor and memory bandwidth to achieve the most energy-efficient configuration subject to QoS constraints. Our approach is also able to progressively reduce the energy consumption of newly installed real-world applications for which there exists no prior resource usage data. Experiments on a Nexus 6 smartphone show that our approach achieves an average energy reduction of 23% (19%) and up to 31% (27%) compared to existing work (and default Android governor) for different combinations of real-world applications running side-by-side in split-screen mode. For applications with no prior resource usage data, the proposed framework saves up to 22% (18%) of energy within at most 14 seconds when compared to existing work (and default Android governor).**

***Index Terms*—Energy management, smartphones, application profiles, phase detection, runtime adaptation, Android**

I. Introduction

Smartphones continue to have more features and functionality that have heretofore been limited to general-purpose computing systems. This shift was due, in part, by technological advances, and in part, to satisfy users' increasing demands for more performance, convenience, and versatility. To further improve application quality of service (QoS) and/or system utilization, designers have leveraged existing powerful hardware designs, which were originally geared towards general-purpose computing systems, to support multi-threaded, multi-process applications on smartphones. For example, larger dynamic random-access memory (DRAM) in smartphones allow several applications to share the device screen simultaneously. That is, a user could split the screen, viewing a web page on one side while composing an email on the other side of the screen. However, the main disadvantage of using such full featured hardware design is the increase in power consumption, which reduces battery life and may even cause overheating [1], [2].

Presently available battery technology has hit a power wall in smartphones [3]; the Li-Ion/Li-Po battery has a simple design, which is easy to mass-produce, but can keep pace with neither the current software and hardware advancements [4] nor increasing application demands. Due to the size and weight constraints, large battery packs are unsuitable. While low power modes can be enabled to extend battery life, the limited functionality, e.g., no GPS, or degraded performance, e.g., dimmed display, may not be desirable.

Energy saving solutions for smartphones often focus on enabling independent fine-grained power management of components or subsystems at different layers of abstraction [5], [6], [7], [8], [9], [10], [11]. However, micro-managing each device/component is a challenge as the number of devices/components can be large. For instance, smartphones often have heterogeneous system-on-chip (SoC) designs, e.g., multi-core CPU, graphics processing unit (GPU), other hardware accelerators such as a digital signal processor (DSP), image signal processor (ISP) etc., and low power DRAM integrated on a single chip. Peripherals may include GPS, modem, wireless local area networks (WLAN), camera, and other off-chip sensors.

In addition, optimizing the energy consumption of each component independently does not always result in minimum system-wide energy consumption, especially as the components must often inter-operate. For instance, the overall system load is often used to adjust core voltage and frequency settings to optimize application QoS. However, such a technique ignores memory usage, which has been shown to be application-specific [12] and which cannot be easily captured by monitoring system load alone. Since the power consumption of memory subsystems are now comparable to that of processors [13], effective energy-aware designs must not only consider the power consumption due to processor cores but also due to memory and bus subsystems, especially since all the peripheral devices integrated within a smartphone has to rely on the memory subsystem to communicate with the processor, and vice versa.

An effective system-level energy management solution must also consider the varying resource usage pattern of an application over time. Such patterns are difficult to determine offline but can present significant power saving opportunities without sacrificing QoS. We propose a coordinated energy manage-

Manuscript received April 3, 2018; revised June 8, 2018; accepted July 2, 2018. This paper was presented in the International Conference on Hardware/Software Codesign and System Synthesis 2018 and appears as part of the ESWEEK-TCAD special issue. This paper was recommended by Conference Chairperson A. Shrivastava. (Corresponding author: Anway Mukherjee.)

The authors are with the department of Electrical and Computer Engineering, Virginia Tech, Arlington, VA 22203. Emails: {anwaym, tchantem}@vt.edu. This work was supported in part by the National Science Foundation under Award Number 1618979.

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

Digital Object Identifier 10.1109/TCAD.2018.2857323

ment solution that, for the first time, aims to reduce the energy consumption of side-by-side applications on smartphones. The main contributions are:

1) We design an application-aware dynamic voltage and frequency scaling (DVFS) policy that jointly reduces the energy consumption of processor cores and memory subsystems with negligible impact on QoS, and which does not require application code instrumentation. The proposed policy uses a catalog of offline profiles and an online controller to select the most energy-efficient configuration of the system without sacrificing performance.
2) We develop a runtime lightweight phase detection tool, whose front-end resides in userspace while the back-end is implemented in the Linux kernel, to account for instantaneous phase changes of an application by leveraging per-core performance monitor unit (PMU) counters, memory access pattern and system bus traffic. The tool allows us to gain a better understanding of an application's system-wide resource usage in order to minimize its energy consumption. Since the proposed technique does not require application code instrumentation, it is applicable to both open-source and proprietary applications, and also assists in the energy management of newly installed applications for which no prior resource usage data exists.
3) We validate our proposed approach and assess its performance by comparing it with the most closely related work [14], as well as the default Android governor, in a multi-process environment by running typical real-world applications side-by-side in split-screen mode in Android Nougat on top of a Nexus 6 smartphone.

We implement our proposed solution on Android [15], as it is the most widely used open-source mobile OS. While we focus on the split-screen technology in this work, our adaptive energy management framework can be applied to other multi-threaded, multi-process application environments.

II. Preliminaries

We present key background concepts and review existing work on energy management of smartphones in this section.

A. Platform

Google Android [15] is an open-source mobile operating system that is extensively used in smartphones. Android 7.0 and higher supports a split-screen mode where the system fills the screen with two applications, (shown in Figure 1), showing them either side-by-side or one above the other. The user can drag the dividing line separating the two to make one application larger and the other smaller. Starting with Android Nougat, background applications are not allowed to run at all to aggressively solve the problem of unbounded energy consumption [1], [2]. The split-screen mode is a relatively new Android solution to boost performance without having background processes.

B. Related Work

Existing Android DVFS policies implemented through device-specific governors [16], such as `interactive`,

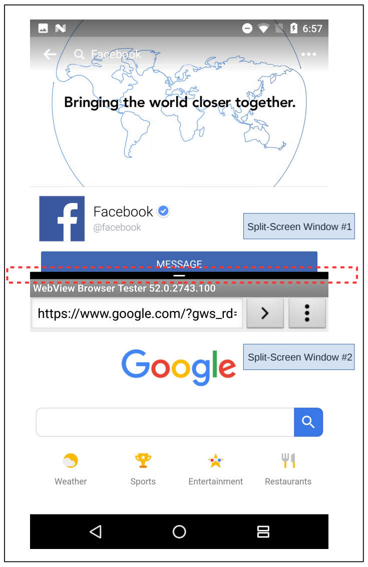

Fig. 1. A screenshot of split-screen execution in Android Nougat, where the top window runs Facebook while the bottom window simultaneously runs a web browser. The dimensions of the split-screen can be altered by dragging the black bar separating the two windows.

`ondemand` and `performance`, primarily focus on QoS instead of system-wide energy consumption and, hence, fail to achieve an optimal energy-efficient policy when used in popular applications [12], [17]. In addition, existing solutions are applicable to only a small subset of popular Android applications [18], [19] or performance benchmarks and micro-benchmarks, and were either tested on software simulators or open-source development boards [20], [21], [22].

We broadly classify energy management techniques for mobile embedded systems into three categories. In the first category, the aim is to develop an improved power-performance DVFS policy by considering one or multiple components of the system [23], [5], [6], [7], [24], [25], [26], [27]. However, to the best of our knowledge, there is no general policy that collectively manages all the components at once without requiring that application code be publicly available. In the second category, DVFS, dynamic power management (DPM) and/or dynamic thermal management (DTM) are used to develop predictive energy-aware system-level policies [9], [10], [11], [28], [29]. Drawbacks of this type of solutions are (i) backward compatibility issues and (ii) that it requires detailed knowledge of the system P-states, which may be difficult to obtain in proprietary software and devices. Finally, in the third category, the goal is to develop workload characterization techniques to support application-level analysis and optimization [30], [31], [32], [33], [34]. However, the major limitation of such an approach is the potential lack of support in both hardware and software; existing software profilers either do not capture the fine-grained changes in memory access patterns over time or require code-level instrumentation. Our solution, on the other hand, can be implemented as a lightweight userspace software that is suitable for smartphones with limited built-in functionality.

Rao et.al. [12] establishes the need for an application-specific, performance-aware energy management framework for Android devices, and showed that a coordinated control of system-wide components can save up to 32% energy compared to the default Android governors. Similar work by Liang et al. reported that reducing the CPU frequency may not always improve the energy consumption of the system [14]. However, these work either do not consider memory bandwidth management or platforms with multiple applications on split-screen.

We bridge the gap in existing research by (i) designing an integrated energy management framework that does not require application code to reduce the system-wide energy consumption of side-by-side applications on smartphones while preserving QoS, (ii) enabling further energy savings via the detection of instantaneous phase changes of applications, and (iii) conducting experiments on an actual smartphone to assess the energy savings and resultant QoS level of the proposed framework.

III. Proposed Framework

It has been shown that energy management approaches that independently consider components of a smartphone lead to sub-optimal solutions [12]. Similarly, energy management schemes that ignore changing resource usage of applications fail to optimize the total system energy consumption [14]. Our aim is to minimize system-wide energy consumption with negligible impact on QoS on smartphones running multiple applications side-by-side. As previously discussed, side-by-side execution through split-screen adds to the energy demand as opposed to the traditional foreground-background task execution model since all processes sharing the device screen run with equal activity context. While we restrict the number of applications sharing the split-screen to two in this work, our approach can readily be extended to multiple applications on split-screen. In our work, the QoS metric is the makespan of a given application, as it is applicable to both compute-intensive and memory-intensive workloads. Note that other metrics can be used. In addition, since our approach requires a fine-grained QoS metric, i.e., one that can change in a small time interval, to find an energy-efficient frequency and voltage settings for the CPU and memory subsystem, we use the normalized instructions per cycle (IPC), as described later in Eq.1, within the framework.

Our proposed application-aware integrated energy management framework is shown in Figure 2. We decouple our work into a two-step procedure: offline and online components. First, we run applications side-by-side through our offline profiling phase (Section IV) to assist in informed decision making online. We also catalog offline profile data into categories of applications for the runtime energy management of applications which do not have offline profile data (Section VII-B). Second, at runtime, an online controller framework (Section VI) is executed periodically to select the best energy-performance configuration in order to minimize the system-wide energy consumption with negligible performance degradation. The online controller relies on an online phase detector (Section V), which monitors the resource usage pattern

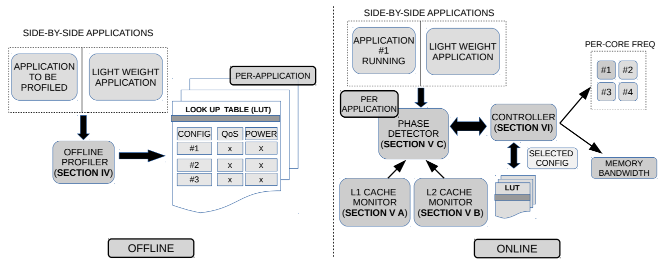

Fig. 2. The proposed framework is decoupled into (i) running applications side-by-side in offline profiling phase (left), and (ii) an online framework to select the best energy-performance configuration (right).

TABLE I
DVFS Configurations for Nexus 6

CPU Freq (GHz)	Level
0.3000	1
0.4224	2
0.6528	3
0.7296	4
0.8832	5
0.9600	6
1.0368	7
1.1904	8
1.2672	9
1.4976	10
1.5744	11
1.7280	12
1.9584	13
2.2656	14
2.4576	15
2.4960	16
2.5728	17
2.6496	18

Mem Bandwidth (MBps)	Level
762	1
1144	2
1525	3
2288	4
3051	5
3952	6
4684	7
5996	8
7019	9
8056	10
10101	11
12145	12
16250	13

of an application over time, along with the offline profiles, to select the voltage and frequency levels of associated cores and memory subsystem that minimizes the energy consumption without sacrificing the target QoS.

IV. Offline Profiling

The goal of the offline profiler is to obtain the average performance data and energy consumption data for a given application under all CPU frequency and memory bandwidth combinations (Table I). Each such combination, which is represented by the tuple *(CPU Frequency, Memory Bandwidth)*, constitutes a system configuration. A lookup table (LUT) is constructed for each application and offers information on the

TABLE II
AN EXAMPLE APPLICATION-SPECIFIC LUT

Config	**CPU Freq** (GHz)	**Mem Bandwidth** (MBps)	**Normalized IPC**	**Power** (mW)
1	0.3000	762	1	2172.9
2	0.3000	1525	1.2	2170.0
3	0.6528	1525	1.3	2288.3
4	0.6528	2288	1.5	1657.6
5	0.7296	2288	1.5	1422.8

relationship among a given system configuration, performance, and power consumption. Each entry of the LUT contains the following parameters: *CPU frequency*, *memory bandwidth*, *normalized IPC*, and *energy metric*. As discussed in the previous section, while our overall QoS metric is the makespan of an application, we use the normalized IPC, $\widetilde{IPC}_i$, for the i^{th} system configuration (see Table II), inside our framework for fine-grained tuning. The IPC is normalized with respect to IPC_1, i.e., the system configuration with the lowest CPU frequency and memory bandwidth, and defined as,

$$\widetilde{IPC}_i = \frac{IPC_i}{IPC_1}, \tag{1}$$

where IPC_i is the average IPC of the i^{th} system configuration (Table II).

The energy metric contains the system-wide power consumption obtained while running an application under the corresponding system configuration. An example application-specific LUT is shown in Table II. For the information contained in the LUTs to be useful online when applications execute side-by-side, we must account for the competing resource demands by the other side-by-side application. Therefore, we run a standard lightweight application in split-screen while performing our offline data collection. In this work, we select the Android emailing service as the lightweight application, but other applications can be used as well. The selection of the most suitable application is left for future work.

We also broadly categorize any combination of applications into *compute intensive*, *memory intensive* or *peripheral intensive*. This assists in the decision making when managing the energy consumption of applications which do not have prior offline profile data (Section VII-B). In this way, we limit the amount of data that must be stored in the LUT (Section VII-A) while allowing for judicious energy management decisions to be made online.

V. ONLINE PHASE DETECTION

Broadly, an application may have computation intensive phases and memory intensive phases. An application has a computation intensive phase if, within some time interval, the application spends most of its time on the CPU. Similarly, an application enters a memory intensive phase if, within some time interval, the application spends most of its time fetching/sending data from/into the main memory. An instantaneous phase change is defined as the transition from a computation intensive phase to a memory intensive phase, and vice versa. Detecting the instantaneous phase change of an application, along with the CPU load and memory footprint, would allow us to (a) understand the application behavior and its resource usage pattern, and (b) distinguish different application's phases to target either CPU frequency or memory bandwidth optimization.

The application-specific per-core CPU activity tool, `Perf` [8], can monitor the CPU intensive phases of the applications. In addition to the CPU load, an application memory footprint is a crucial piece of information that can be used to analyze and improve the memory bandwidth utilization of an application, which can, in turn, reduce energy consumption. This is especially important in Android applications as they are notorious for poor data and instruction locality [35]. Since many small embedded systems lack built-in support for obtaining an application's accurate memory footprint due to size, weight and power constraints, we propose a memory traffic monitoring mechanism to enable phase detection on smartphones. In this section, we describe our fine-grained measurement method of the load serviced by the L1 private cache and a shared last level cache (LLC) through cache monitors, and the proposed dynamic phase detection mechanism for applications in split-screen mode.

A. L1 Cache Monitor

A major step towards predicting phases of an Android application is to analyze and detect sharp changes in memory access patterns. Our goal is to find out per-core L1 cache misses to account for the average percentage contribution of each core on the aggregate L2 cache outgoing traffic to be serviced by the main memory. The L1 cache traffic is measured in Megabits per second (Mbps). Since `Perf` does not report on *per application per-core* L1 cache statistics on Android OS, we leverage the performance monitor units (PMUs) and use them as software event counters. Each PMU counter can be programmed to monitor and register a list of L1 cache events. (Smartphones usually are multi-core systems where each core has a set of PMUs.) In our solution, the per-core per-application data is stored in a hash table which resides inside the global stack of the Linux scheduler. A custom `syscall` redirects the data to the userspace where we implement our online controller. If such data is made available through proprietary drivers, we could have the framework implemented entirely in userspace. A flowchart depicting the L1 cache monitor is shown in the left side of Figure 3. The L1 cache monitor polls the amount of per-core, per-application memory traffic from private L1 caches to the shared L2 cache.

B. Last Level Cache Monitor

Having recorded the L1 cache traffic, we now need to determine the aggregate outbound LLC traffic serviced by the main memory to predict application phase changes. The LLC miss traffic, along with the application-specific per-core L1 cache traffic, accounts for the average percentage contribution of each core on the aggregate LLC outgoing traffic. This cache traffic is also measured in Megabits per second (Mbps). In our solution, we utilize the system specific hardware PMU (attached to the LLC), and its dedicated driver implementation

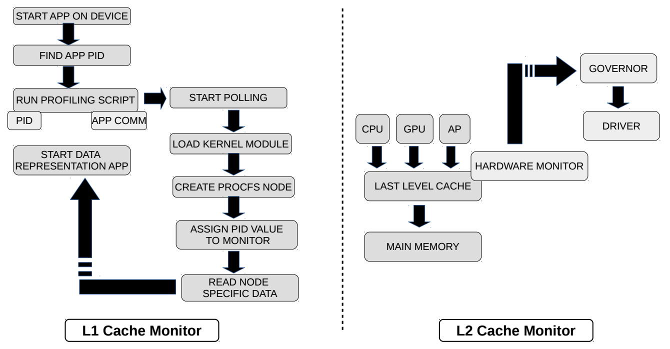

Fig. 3. Flowcharts depicting the L1 cache monitor (left) and the LLC cache monitor, with steps for determining the aggregate outbound LLC traffic serviced by the main memory (right).

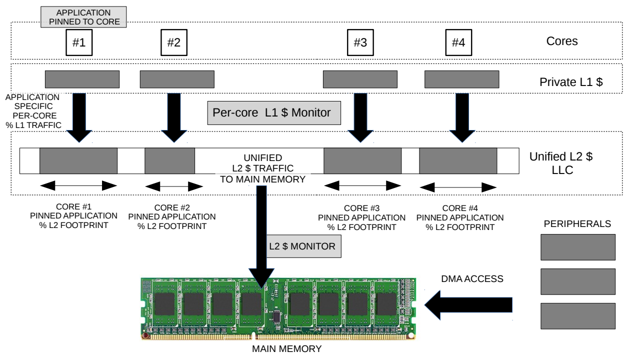
APPLICATION PINNED TO CORE
#1
#2
#3
#4
Cores
Private L1 $
APPLICATION SPECIFIC PER-CORE % L1 TRAFFIC
Per-core L1 $ Monitor
UNIFIED L2 $ TRAFFIC TO MAIN MEMORY
Unified L2 $ LLC
CORE #1 PINNED APPLICATION % L2 FOOTPRINT
CORE #2 PINNED APPLICATION % L2 FOOTPRINT
CORE #3 PINNED APPLICATION % L2 FOOTPRINT
CORE #4 PINNED APPLICATION % L2 FOOTPRINT
PERIPHERALS
L2 $ MONITOR
DMA ACCESS
MAIN MEMORY

Fig. 4. Application phase detection using (1) application-specific per-core L1 traffic directed to the shared LLC, and (2) the LLC cache traffic flowing to the main memory.

in the kernel, to export the data related to the LLC activity to userspace for further analyses, as shown in the right side of Figure 3. The LLC PMU hardware counter polls every 10 ms for the amount of traffic directed towards the system main memory. Therefore, all data collection mechanisms are also activated every 10 ms. In contrast, the polling frequency of `Perf` is limited to 100 ms[1], which is inadequate for fine-grained energy management. In addition, while in our case, the Nexus 6's LLC is the L2 cache, our approach can be extended to an arbitrary number of cache levels.

C. Phase Detector

We now discuss how to use the data from the L1 and LLC cache monitors, and application specific per-core CPU activity using `Perf`, to detect the phase change of an application, which will permit the controller (Section VI) to save energy without severely affecting the QoS of an application. Since running side-by-side applications simultaneously poses a problem of data coalescence on shared resource similar to multi-process execution, we pin an application to specific core-group(s) to isolate the memory footprint of each application.

[1]For the Nexus 6, `Perf` taps into the PMU framework of the Krait 450 processor through system calls, which is a major reason for its coarse-grained polling intervals.

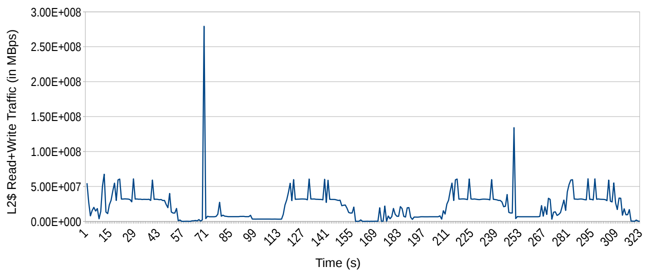

Fig. 5. Memory traffic data directed to and from the main memory from the L2 cache due to Youtube while running side-by-side with the default Android emailing service application on Nexus 6 over time. The data clearly shows periodic surges in data traffic going into the main memory.

The application-specific per-core data traffic between the CPU and the main memory can now be dynamically monitored by (1) collecting per-core L1 traffic directed to the shared LLC, and (2) the LLC cache traffic flowing to the main memory, as shown in Figure 4.

Figure 5 reports the amount of traffic that is directed to the main memory from the L2 cache due to Youtube while running side-by-side with the default Android emailing service application on Nexus 6. It shows the distinct, periodic memory traffic pattern when we run the application for an extended period of time. We can define each region of high memory activity as a memory intensive phase of the application in question. Similarly, the CPU activity monitor (using `Perf`) also shows the CPU intensive phases of the applications. Distinguishing consecutive phases of an application requires the following steps. We record offline the minimum change in cache traffic that would cause a change in the memory bandwidth under the default Android governor, and quantify it as the *sensitivity level* (S) of the application under consideration. At runtime, we use `Perf` over a constant time interval to monitor the percentage usage of the CPU for computation by the application, which is reflected by the IPC value. Similarly, we use the memory footprint monitors (L1 and LLC cache monitors) over the same time interval to record the application-specific data exchange. We then calculate the variance in application sensitivity level S with respect to *both* CPU computation and memory traffic as

$$U_j = S \times \frac{IPC_i}{\left(\frac{out\_L1\_traffic}{out\_L2\_traffic}\right)}, \tag{2}$$

where the resource utilization U_j, $p_j \in \mathbf{P}$, represent the per-core per-application resource utilization of a processor core p_j. If $\frac{U_j}{S} \leq 1$, a memory intensive phase of the application is detected. Otherwise, when $\frac{U_j}{S} > 1$, a CPU intensive phase of the application is observed. For applications for which no offline profile data exists, we classify them based on the closest matching offline profile.

Once we can distinguish the different phases of an application, the next logical step would be to optimize these phases for a system-wide minimal energy consumption configuration with negligible impact on QoS, as discussed next.

VI. Controller

The goal of the online controller is to achieve energy optimization while maintaining an acceptable application QoS level. The controller maintains the target QoS of an application by using the information from the phase change detector and the LUTs to calculate the current QoS and energy consumption values, and prescribe corrective measures. We construct our online controller based on the approach taken by Imes et al. [24]. The inputs to the controller are (1) the LUT table for a given application (Table II), and (2) a reference QoS value, P_{ref}, for the controller to maintain. To maintain QoS, a proportional integral (PI) controller is used. (Based on our results, a more sophisticated controller is not necessary but can be applied.) Our online PI controller provides the output according to

$$co = co_{bias} + k_c \cdot e(t) + \frac{k_c}{t} \cdot \int e(t)dt, \tag{3}$$

where co is the controller output, co_{bias} is the controller bias, k_c is the controller gain, $e(t) = |P_{ref} - P_{measured}|$ is the controller error, and t is the controller period. The training period (offline) for the controller consists of running each application (for which an offline profile exists) multiple times until the controller selects a configuration that results in P_{ref} on average. This allows us to record the tuning parameter values, k_c and co_{bias}, for each application beforehand.

At runtime, the controller is executed as a daemon task while the Android applications of interest run in the foreground in split-screen mode. At the beginning of each controller period t, the controller computes $e(t)$ and uses it to react accordingly in the next controller cycle. In contrast to the work by Imes et al.[24], we replace the Kalmann filter with our online phase detector. The controller output, co, is the required QoS value an application must attain during the current period to maintain P_{ref}. The value co translates to the desired configuration level of the application and is used as a reference to select the most energy-efficient system configuration from the lookup table (Table II). As will be shown in Section IX-D, the overhead of our proposed online phase detector and controller is negligible. We now discuss two methods to make our work more practical in the next section.

VII. Practical Factors

We now discuss two potential limitations to our proposed framework, and present solutions to tackle them.

A. Limiting LUT Storage Cost

A major driving force for a learning-based offline profiling approach proposed by Rao et al.[12] is to prevent the LUTs from occupying considerable memory space. A major concern especially for smartphones with limited memory would be the limitations on the space required to store performance, memory traffic, and energy consumption data in a LUT for a potentially large number of application combinations which may run side-by-side simultaneously.

Dong et al.[36] analyzed the memory access patterns of Android applications and reported the extensive use of native shared libraries among a large number of Android applications. That is, for any pair of applications, one application's shared libraries are often accessed by the other, resulting in similar memory access patterns. These results suggest that by judiciously managing shared libraries, instruction access efficiency as well as the overall performance can be improved. Our approach relies on these findings [36]. Specifically, we limit the storage requirement of our approach by maintaining a catalog of offline profiles, each of which encompasses a set of applications with similar resource usage patterns (CPU usage, memory, peripheral IP usage, etc.). That is, we classify each set of applications as compute-intensive, memory-intensive or peripheral-intensive. While compute-intensive and memory-intensive applications have high CPU utilization and high CPU-memory interactions, respectively, peripheral-intensive applications, on the other hand, attribute neither to high CPU utilization nor high CPU-to-memory bandwidth usage. Rather, these applications execute on peripheral IPs while accessing the main memory through a separate system bus. The actual number of offline profiles for each set of applications can be selected based on the available memory space on a given system. Note that each entry into the catalog is accompanied by an application's unique identifier (`app_comm`) and its corresponding reference sensitivity level (S). This facilitates a unique index to each LUT entry.

B. Applications without Offline Profiles

For applications that are newly introduced into the system and for which profiles are not readily available, we propose to leverage the existing catalog of offline profiles and the online controller to achieve an improved performance-energy configuration.

For newly installed applications, we select the most similar profile from the catalog of existing offline profiles by (i) running the newly installed application to analyze its processor utilization characteristics, as well as memory access patterns (and S) and, (ii) browsing through our list of readily available offline profiles stored in the system to find the closest match to our newly installed application. We then use the online controller to adapt to the new applications by tuning the controller output to converge to the targeted QoS in the least number of iterations. As will be shown in Section IX-B, an application with no offline profile will only need to run for at most 14 seconds before the proposed framework can achieve significant energy reduction with negligible QoS degradation. Once complete and the controller tuning parameters recorded, later reruns of the application will not require a second tuning, provided the side-by-side execution behavior pattern and system workload do not change.

VIII. Experimental Setup

We next describe our experimental platform, as well as the application benchmarks and evaluation criteria that were used to assess the effectiveness of our approach.

TABLE III
NEXUS 6 HARDWARE SPECIFICATIONS

Component	Specification
SoC	Qualcomm Snapdragon 805
CPU	Krait 450 (quad core running at 2.7GHz)
RAM	3GB LPDDR3
Flash	32GB
Sensors	Accelerometer, GPS, Gyro, Baro
GPU	Adreno 420
Wifi	802.11 a, b, g, n, ac, dual-band
Battery	Li-Po 3220 mAh

A. Experimental Platform and Settings

We implemented and tested our energy management framework on a Nexus 6 smartphone, whose hardware specifications are listed in Table III. The Nexus 6 extends a user-level driver support to implement various DVFS policies on the main memory. The Qualcomm Snapdragon chip set is augmented by a piece of hardware attached to the LLC, which serves to monitor real-time LLC traffic, though this feature was not exposed to users. Our implementation exports the hardware-dependent statistics of cache subsystems to userspace. Nexus 6 runs Android Nougat, which introduced side-by-side execution in split-screen mode where multiple applications can simultaneously run with similar foreground context. In our target system, i.e., the Nexus 6 smartphones, the LLC is the L2 cache.

We now provide some specific details for the purpose of reproducibility of results. Before performing offline profiling, the following options were disabled: `USB charging` to record accurate system-wide power consumption and `mpdecision` to prevent CPU-hotplugging. In addition, the `CPU_freq_boost` configuration flag was disabled to make sure touching the screen does not inadvertently cause the CPU frequency to increase, as this may lead to erroneous data collection. We reduced screen brightness to 50% and the `Wifi` module was kept on to simulate a common *de facto* smartphone runtime interference. We run our side-by-side application simultaneously with the default Android mail service application on the split screen. This allows us to simulate a system environment with application interference on shared resources. We pin each side-by-side application to specific core-group(s), as this allows us to monitor application memory footprint in isolation. In addition, we use `Monkey` [37] to generate pseudo-random user behaviors in our experiments for reproducibility.

In our setup, the energy consumption is measured using the OS-level battery readings; Android exports the battery status through system-level `sysFS` nodes, both for instantaneous current (mA) and instantaneous voltage (mV). We deployed a daemon module to record the current and voltage readings to obtain the instantaneous power usage, and subsequently, the system-wide energy consumption during our experiments.

B. Applications with Offline Profiles

A set of five real-world applications was selected as our benchmark suite, as each application has unique CPU and memory requirements. We briefly introduce each application next and note its length of evaluation. We run each application side-by-side simultaneously with the default Android emailing service application to generate an offline profile, categorize the applications, and record and catalog the offline data in a LUT for each application as shown in Table II.

1) VidCon: A video converter application which uses a specific library to convert videos to different formats. VidCon is primarily a memory-intensive application. It relies less on the CPU and more on hardware accelerators. For our experiments, we converted an mp4 video using the default configurations.

2) MobileBench: An established browser benchmark. This application shows patterns of stark switches between CPU-intensive and memory-intensive phases. The benchmark loads a collection of website contents onto the phone memory. It offers automatic horizontal and vertical zooming and scrolling as well. The benchmark uses the Chrome browser application to run the tests.

3) Pokemon Go: A popular Android gaming application. This application shows constant increase in memory bandwidth usage throughout its run with sharp shifts to the GPU-intensive phase. Since this application is GPU-intensive, we have very low CPU usage with very high memory bandwidth usage. The game is manually played for 200 s during our experiments.

4) Facebook: A popular internet based social media and networking application. We selected the videoplayer feature of this application and initiated long videos for our experiments. As this is a highly optimized application, Facebook shows sharp increase in CPU and memory usage only when the application is active. We tested the application for 200 s during our experiments.

5) Spotify: An established audio and video streaming application. This application is tested for 200 s with songs being changed every 20 s. This application shows very high CPU usage with sharp switches to memory-intensive phases during its run.

C. Applications without Offline Profiles

For applications whose performance-energy profiles are not available (Section VII-B), we selected another set of five real-world applications as our test suite. We assume that we have prior knowledge of the already existing applications (Section VIII-B) which have their offline profiles stored in the system and that each new application has similar resource usage patterns as the applications listed in Section VIII-B. (In Section IX-B, we discuss cases where this assumption does not hold.)

1) Media Converter: A video converter application which uses a specific library to convert videos to different formats.
2) Android Browser: The default browser in the Android vanilla OS. The application is used for browsing, viewing images and video playback.
3) Fruit Ninja: Another popular Android gaming application, similar to Pokemon Go mentioned in Section VIII-B.

4) Youtube: A video sharing application which allows its users to upload, share and view videos online. It offers a diverse category of both amateur and professional video archive.
5) Amazon Music: A popular audio and video streaming application.

D. Evaluation Metrics

The performance metrics of interest are the resultant QoS level and system-wide energy consumption. As previously discussed, the QoS of an application can depend on a number of factors such as IPC, latency, throughput, etc. Since most Android applications are available as code obfuscated APKs, we focus on IPC and application makespan, as they take into account both CPU and memory performance and are sensitive to system workload. They are also application neutral.

We compared our approach against Android's default governors (**DG**), as our work is the first to consider the side-by-side execution model. While other governors exist in Android, the default policies were selected since changing governors require root privilege. In addition, the other Android governors are unsuitable for energy-QoS optimization for a number of reasons. Namely, both `OnDemand` and `Performance` have the tendency to use the maximum frequency, resulting in excessive energy usage. On the other hand, `Powersave` tends to select the lowest frequency, thus sacrificing QoS. `Interactive`, which is the CPU default governor, is more responsive to changing system loads. Similar arguments can be made for device governors. To further validate our approach, we compare against the most closely related work by Liang and Lai [14], heretofore referred to as **CS**, which is a critical-speed based DVFS technique that adjusts CPU frequency but not memory bandwidth.

IX. Results

Our experimental results pertaining to various applications and test scenarios, as well as a discussion on the overhead of the proposed approach, is provided in this section.

A. Applications with Offline Profiles

For applications that were profiled offline, we report the average value for performance and energy over 10 runs per benchmark in Table IV. Figures 6 and 7 compare the percentage of total execution time spent on different frequency and bandwidth levels between our approach and DG. Note that the frequency and bandwidth levels for CS are not shown to maintain readability, as using CS results in a very limited number of CPU frequencies for a large percentage of time and 100% of time on a single memory bandwidth. In general, our approach, when compared to the default governor, is able to save a significant amount of energy for all applications considered, albeit with small hits in IPC of less than 2% and with an average increase in makespan of up to 3%. In contrast, our approach, shows an overall improvement of 23%, 4% and 6% on energy savings, application IPC, and makespan, respectively, when compared to CS.

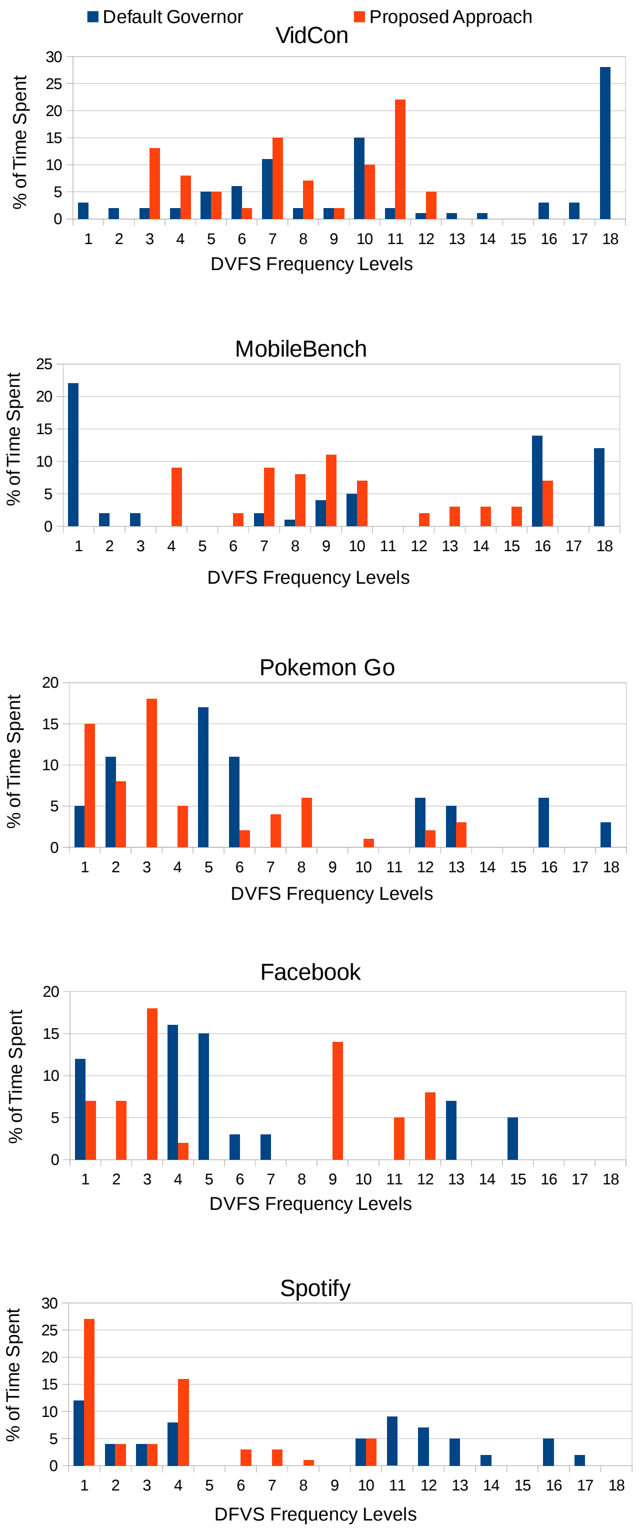

Fig. 6. Percentage of total execution time spent on different CPU frequency levels (Table I) when using DG and the proposed approach. For CS, a very limited number of CPU frequencies are selected and thus not included to maintain readability.

1) VidCon: As shown in Figure 6, using the default governor, the cores spent more than half of the time executing at the highest frequency. This is because DG blindly sets the same

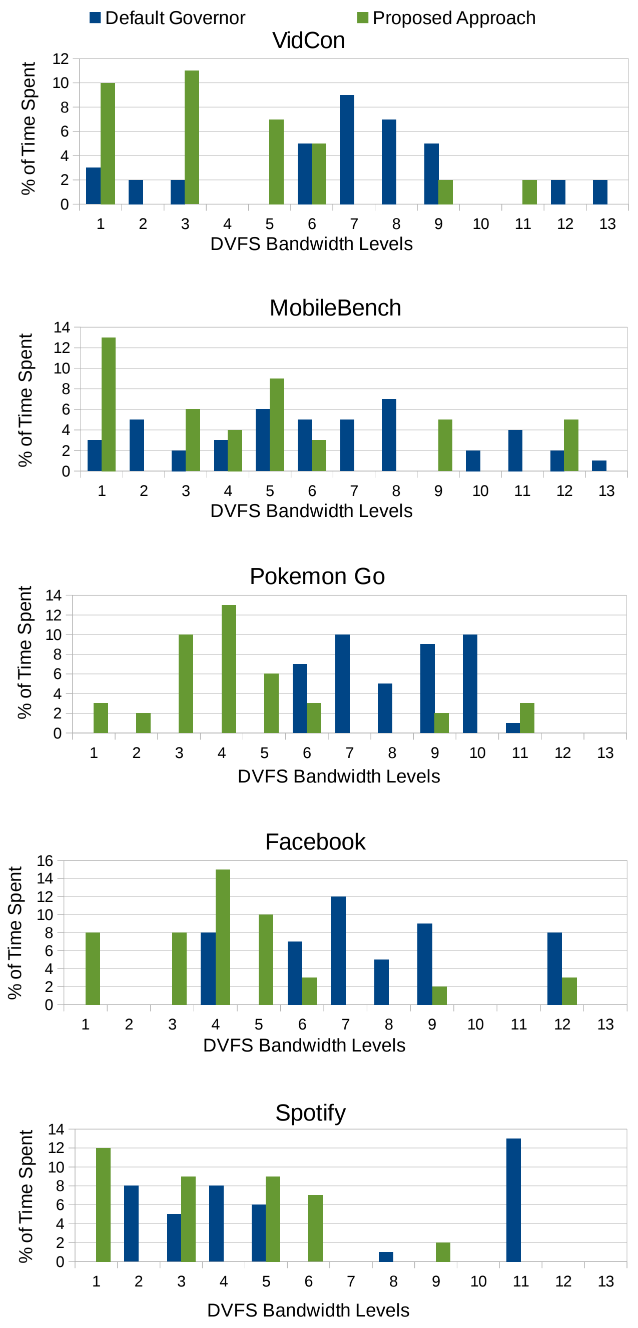

Fig. 7. Percentage of total execution time spent on different memory bandwidth levels (Table I) when using DG and the proposed approach. Note that CS does not adjust memory bandwidth.

frequency for all the cores without considering the individual core loads. This drawback was addressed in our approach, resulting in an energy improvement of 24.5%, IPC hit of 0.2%, and increased makespan of up to 1%. With respect to CS, we see an improvement of 3.6%, 31.1% and 7.2% in IPC, energy savings, and application makespan, respectively. The improvement can be attributed to our holistic approach, which varies both CPU frequency and memory bandwidth.

2) MobileBench: This application is a representative of application with multiple phases. During a CPU-intensive phase, our solution selects higher core frequencies, as shown in Figure 6. Core frequencies are reduced and memory bandwidth is increased during memory-intensive phases (Figure 7), resulting in an energy improvement of 18.2% with an IPC degradation of 1.6% and an increased makespan of 6%. When compared to CS, we observe an overall improvement of 0.3%, 11.6% and 1% in IPC, energy savings and application makespan, respectively. The result confirms that the application has more CPU-intensive phases than memory-intensive phases, as this allows CS to optimize the application's critical speed.

3) Pokemon Go: Since this application is memory- and GPU-intensive (with bursts of high CPU loads), an energy-efficient solution can be obtained by reducing core frequencies while increasing memory bandwidth, as shown in Figures 6 and 7. Our approach improves the energy consumption over DG by 19.1% with an IPC degradation of 0.9% and an increased makespan of up to 2%. The slight IPC degradation was due in part by the sharp shifts to CPU-intensive phases, which causes our controller to mis-predict, and, in part, by GPU bottleneck. However, when compared to CS, we see an overall improvement of 5%, 19.6% and 3.2% in IPC, energy savings and application makespan, respectively.

4) Facebook: From Figure 6, our approach selects very different core frequencies compared to DG, resulting in an energy improvement of 7% and IPC degradation of 1.1% with an increased makespan of 2%. Here, less energy can be saved, as Facebook has multiple resource-intensive processes. When compared to CS, we observe an overall improvement of 1.3%, 22.8% and 8.5% in IPC, energy savings and application makespan, respectively. Since Facebook has more memory-intensive phases than CPU-intensive phases, CS fails to optimize the application performance by just calculating the critical speed.

5) Spotify: Our approach resulted in 27.6% energy improvements with an IPC hit of no more than 1.2% and an increased makespan of up to 3%. Since this application is fairly CPU intensive, core frequencies cannot be further reduced in general (Figure 6). However Figure 7 shows that the choice of memory bandwidth can be improved to attain both performance and energy gains. In addition, we see an overall improvement when compared to CS of 2.2%, 29.1% and 9.2% in IPC, energy savings and application makespan, respectively. This results highlights a drawback of CS, which adopts an independent, module-based DVFS policy, even though the critical DVFS speed is computed with an informed knowledge

TABLE IV
SUMMARY OF IPC HITS, ENERGY SAVINGS, AND MAKESPAN INCREASE OF PROFILED APPLICATIONS USING PROPOSED APPROACH COMPARED TO THE DEFAULT ANDROID GOVERNORS (DG) AND CRITICAL-SPEED BASED DVFS TECHNIQUE (CS) [14]

Application	**IPC hits (%)**		**Energy savings (%)**		**Makespan increase (%)**	
	DG	CS	DG	CS	DG	CS
VidCon	-0.2	+3.6	+24.5	+31.1	+0.5	-7.2
MobileBench	-1.6	+0.3	+18.2	+11.6	+6.1	-1.0
Pokemon Go	-0.9	+5.0	+19.1	+19.6	+1.4	-3.2
Facebook	-1.1	+1.3	+7.0	+22.8	+ 2.1	-8.5
Spotify	-1.7	+2.2	+27.6	+29.1	+2.5	-9.2

TABLE V
SUMMARY OF IPC HITS, ENERGY SAVINGS, AND MAKESPAN INCREASE OF APPLICATIONS WITHOUT OFFLINE PROFILES USING PROPOSED APPROACH COMPARED TO THE DEFAULT ANDROID GOVERNORS (DG) AND CRITICAL-SPEED BASED DVFS TECHNIQUE (CS) [14]

Application	**IPC hits (%)**		**Energy savings (%)**		**Makespan increase (%)**	
	DG	CS	DG	CS	DG	CS
Media Converter	+0.8	+5.6	+18.7	+22.5	-8.2	-11.1
Android Browser	-0.1	+2.1	+12.4	+13.4	+5.7	-6.0
Fruit Ninja	-3.1	+2.3	-1.4	+15.8	+22.7	-0.7
Youtube	-1.2	+3.0	+15.9	+12.3	+3.1	-2.9
Amazon Music	-2.1	+4.3	+18.3	+20.7	+7.4	-6.8

of application's memory access rate.

B. Application without Offline Profiles

To demonstrate the effectiveness of using existing profiles on an application with no offline profile, we performed an additional set of experiments. Here, we assume that the offline profiles for VidCon, MobileBench, Pokemon Go, Facebook and Spotify are readily available. Now, we present a set of newly installed applications, namely, Media Converter, Android Browser, Fruit Ninja, YouTube and Amazon Music, all of which had not previously been profiled offline. Each of the newly installed applications have a unique software signature. That is, VidCon, MobileBench, Pokemon Go, Facebook, and Spotify are expected to have similar resource usage patterns as Media Converter, Android Browser, Fruit Ninja, YouTube and Amazon Music, respectively. We tested our proposed framework on the new applications using the offline profiles of the existing applications. This idea can be extended to group multiple applications of similar software signatures and matching them to a generic set of unique offline profiles.

Again, we compared our approach against DG and CS, and reported the average value of application performance and energy consumption over 10 runs per application in Table V. From our experiments, compared to the default governor, we found that reusing the profiles of most of the applications helps to save a significant amount of system wide-energy, albeit with small hits in IPC (a little over 3%) and an average increase in makespan of up to 9%. On the contrary, compared to CS, we see an overall improvement in system-wide energy, application IPC, and makespan. We now provide more details on our findings by discussing selected applications (two with the best performance and one with the worst performance) as case studies, when compared with the default governor.

We start with Amazon Music and Media Converter which have the best overall performance when using our proposed solution. Amazon Music has an uneven CPU utilization with one core ending up with very high utilization for a long period of time while the other cores show marginal utilization, with short bursts of high usage. Our solution utilizes the performance-bandwidth offline profile of Spotify for use with Amazon Music. The application reports an IPC degradation of as little as 2% with a 18.3% system-wide energy improvement. On the contrary, compared to CS, the application reports an IPC and makespan improvement of 4.3% and 6.8% respectively, with a 20.7% system-wide energy improvement. Both Spotify and Amazon Music work under a similar client-server module, which likely attributes to their similar CPU usage and memory access pattern. Similarly, Media Converter, which utilizes the offline profile of VidCon, reports an IPC degradation of as little as 1% with a 18.7% energy improvement when compared to DG. When compared to CS, we see an IPC and system wide energy improvement of 5.6% and 22.5% respectively.

While our proposed approach always outperforms CS, Fruit Ninja shows the worst performance when comparing the default governors with our proposed solution. The primary reason for such QoS degradation can be attributed to a very low degree of shared libraries between the two gaming applications, the other one being Pokemon Go. That was evident from the distinct difference in the memory access pattern of the two applications under consideration. Hence, we observed a negative QoS of 3% and a reduced energy consumption when compared with the default governor's performance. In such a case, we can (i) increase the granularity of the offline profile to improve the phase detection mechanism, or (ii) create an additional LUT at runtime for future use.

To summarize, our approach is able to save a significant amount of energy (up to 18.77%) for almost all the applications considered and which do not have offline profiles, albeit with a hit in IPC of up to 3% and an average increase in makespan of up to 9% when compared to DG. Against CS, we see an overall improvement of 22.5%, 5.6% and 11.1% on energy savings, application IPC, and makespan, respectively. It is interesting to note that CS performs worse than both DG and our approach. However, CS was validated against demo benchmarks and not real-world applications.

C. Variable Usage Patterns

Most smartphone applications are reactive to user behaviors. In order to validate the robustness of our approach, we experimented on the effect of varying end users' usage pattern on application performance and energy savings. We run Amazon Music (a representative of applications without offline profiles) under two different scenarios: (i) 5 runs with `Monkey` [37] to generate pseudo-random interactions with the application; and (ii) 5 runs using an actual user to vary the usage pattern of the application. We report our findings in Table VI. The results indicate fairly negligible differences for the different metrics between `Monkey` and an actual user. Since it is difficult to precisely predict users behaviors and their specific interactions with an application, we rely on the online controller to adapt to dynamic changes. As future work, we plan on leveraging our framework to incorporate online learning [38] to better

TABLE VI
IPC HITS, ENERGY SAVINGS, AND MAKESPAN INCREASE OF AMAZON MUSIC WITH VARYING APPLICATION USAGE PATTERNS

Application	**IPC hits (%)**		**Energy savings (%)**		**Makespan increase (%)**	
	Monkey	User	Monkey	User	Monkey	User
Amazon Music	-2.1	-2.5	+18.3	+15.6	+7.4	+7.6

TABLE VII
SUMMARY OF IPC HITS, ENERGY SAVINGS, MAKESPAN INCREASE, AND OVERHEAD OF SPOTIFY USING PROPOSED APPROACH COMPARED TO DEFAULT GOVERNOR WITH VARYING CONTROLLER PERIODS

Period (in s)	**IPC hits (%)**	**Energy savings (%)**	**Makespan increase (%)**	**Overhead**
1	-1.7	+27.6	+2.5	4%
2	-1.5	+22.8	+2.4	4%
3	-1.5	+21.5	+2.4	6%
4	-2.9	+9.6	+ 8.0	6%
5	-7.2	+5.0	+11.4	7%

TABLE VIII
SUMMARY OF IPC HITS, ENERGY SAVINGS, MAKESPAN INCREASE, AND OVERHEAD OF AMAZON MUSIC USING PROPOSED APPROACH COMPARED TO DEFAULT GOVERNOR WITH VARYING CONTROLLER PERIODS

Period (in s)	**IPC hits (%)**	**Energy savings (%)**	**Makespan increase (%)**	**Overhead**
1	-2.1	+18.3	+7.4	4%
2	-2.3	+14.7	+7.9	5%
3	-4.1	+11.0	+9.4	5%
4	-6.8	+5.2	+12.2	5%
5	-9.7	+5.0	+17.5	6%

predict resource usage patterns over time. Work on hardware-software co-design for application load monitoring [39] can also be applied to our framework.

D. Overhead

The overhead of our approach depends on the period of the controller (Section VI). In this work, said period is set to 1 s, which is a safe lower bound on the controller period, as changing CPU frequency and memory bandwidth takes time on the actual hardware. Combining with online monitoring (less than 10 ms), LUT search (about 7 ms on average), and actually setting new core frequencies and memory bandwidth through `sysfs` node writes, the total overhead of a single period is no more than 25 ms, which is a little over 4% of the period of the controller. This makes the computation overhead of our approach fairly negligible.

To experimentally assess the overhead of our approach, we select two applications, Spotify (a representative of applications with offline profiles) and Amazon Music (a representative of applications without offline profiles), and compare the performance metrics and energy savings of the applications as functions of the controller period, as shown in Tables VII and VIII, respectively. We also report the computation overhead herewith. We see a greater degradation in IPC, energy saving and makespan with an increase in controller period over 3 s. This is attributed to our design decision; we collect the CPU related PMU data in hash tables inside the kernel. Similarly, the PMU data recorded for cache traffic resides in a loadable kernel module. Both the data structures are limited by their size and structural integrity. Increasing the controller period beyond 3 s leads to data overflow on the aggregate buffer. A workaround was to truncate the data bits to the first five significant digits, which introduces enough errors and results in fairly poor system configuration decision making. A non-linear growth in computational overhead as a function of the controller period can be attributed to a higher accumulation of online data, which needs to be processed and sorted through. Note, however, that detecting an application's phase change and determining the appropriate energy-efficient configuration take constant time irrespective of the controller period.

X. CONCLUSIONS

In this article, we presented an integrated system-level energy management approach that uses a software support framework to obtain application-specific performance, memory, and energy data and find the most energy-efficient CPU frequencies and memory bandwidth without sacrificing QoS. In addition, our approach detects instantaneous phase changes of applications to further save energy and can be used to minimize the energy consumption of new applications. Experiments on a Nexus 6 smartphone revealed that our approach can help save significant system-wide energy with negligible loss in QoS. In the future, we plan on extending our framework to consider multiple applications that are simultaneously running on the same processor core and optimizing energy usage of peripherals.

REFERENCES

[1] Android Developers. Android 8.0 Behavior Changes. (2018). [Online]. Available: https://developer.android.com/about/versions/oreo/android-8.0-changes.html

[2] ——. Background Location Limits. (2018). [Online]. Available: https://developer.android.com/about/versions/oreo/background-location-limits.html

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

[4] V. Etacheri, R. Marom, R. Elazari, G. Salitra, and D. Aurbach, "Challenges in the development of advanced Li-ion batteries: a review," *Energy & Environmental Science*, vol. 4, no. 9, pp. 3243–3262, 2011.

[5] M. A. Rumi, D. H. Hasan *et al.*, "CPU power consumption reduction in android smartphone," in *International conference on Green Energy and Technology (ICGET),*, 2015, pp. 1–6.

[6] S. He, Y. Liu, and H. Zhou, "Optimizing smartphone power consumption through dynamic resolution scaling," in *Proceedings of the Annual International Conference on Mobile Computing and Networking*, 2015, pp. 27–39.

[7] P. T. Bezerra, L. A. Araujo, G. B. Ribeiro, A. C. d. S. B. Neto, A. G. Silva-Filho, C. A. Siebra, F. QB 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 ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks*, 2013, pp. 189–196.

[8] A. C. de Melo, "The new linux 'perf' tools," in *Slides from Linux Kongress*, vol. 18, 2010.

[9] A. W. Min, R. Wang, J. Tsai, and T.-Y. C. Tai, "Joint optimization of DVFS and low-power sleep-state selection for mobile platforms," in *International conference on Communications (ICC)*, 2014, pp. 3541–3546.

[10] O. Sahin and A. K. Coskun, "Qscale: Thermally-efficient QoS management on heterogeneous mobile platforms," in *International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–8.

[11] X. Li and J. P. Gallagher, "An energy-aware programming approach for mobile application development guided by a fine-grained energy model," *CoRR*, vol. abs/1605.05234, 2016. [Online]. Available: http://arxiv.org/abs/1605.05234

[12] K. Rao, J. Wang, S. Yalamanchili, Y. Wardi, and Y. Handong, "Application-specific performance-aware energy optimization on android mobile devices," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 169–180.

[13] D. Shingari, A. Arunkumar, and C.-J. Wu, "Characterization and throttling-based mitigation of memory interference for heterogeneous smartphones," in *International Symposium on Workload Characterization (IISWC)*, 2015, pp. 22–33.

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

[15] Android Developers. (2013) Android SDK. [Online]. Available: http://developer.android.com/sdk/ndk/index, html

[16] N. Chaudhary, T. Pallavi *et al.*, "Bus bandwidth monitoring, prediction and control," in *International conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2015, pp. 1152–1158.

[17] A. Carroll and G. Heiser, "Unifying DVFS and offlining in mobile multicores," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 287–296.

[18] Y. G. Kim, M. Kim, and S. W. Chung, "Enhancing energy efficiency of multimedia applications in heterogeneous mobile multi-core processors," *IEEE Transactions on Computers*, vol. 66, no. 11, pp. 1878–1889, 2017.

[19] X. Chen, J. Mao, J. Gao, K. W. Nixon, and Y. Chen, "MORPh: mobile OLED-friendly recording and playback system for low power video streaming," in *Proceedings of the Annual Design Automation Conference*, 2016, p. 153.

[20] M. H. Memon, M. Hunain, A. Khan, R. A. Shaikh, and I. Khan, "Power management for android platform by set CPU," in *International conference on Computing for Sustainable Global Development (INDIACom)*, 2016, pp. 3953–3958.

[21] V. Spiliopoulos, A. Bagdia, A. Hansson, P. Aldworth, and S. Kaxiras, "Introducing DVFS-management in a full-system simulator," in *International Symposium on, Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2013, pp. 535–545.

[22] F. Ghanei, P. Tipnis, K. Marcus, K. Dantu, S. Ko, and L. Ziarek, "Os-based resource accounting for asynchronous resource use in mobile systems," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2016, pp. 296–301.

[23] X. Chen, C. Xu, and R. P. Dick, "Memory access aware on-line voltage control for performance and energy optimization," in *Proceedings of the International Conference on Computer-Aided Design*, 2010, pp. 365–372.

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

[25] E. Ahmad and B. Shihada, "Green smartphone GPUs: Optimizing energy consumption using GPUFreq scaling governors," in *International conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, 2015, pp. 740–747.

[26] J.-G. Park, C.-Y. Hsieh, N. Dutt, and S.-S. Lim, "Quality-aware mobile graphics workload characterization for energy-efficient DVFS design," in *Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, 2014, pp. 70–79.

[27] A. Prakash, H. Amrouch, M. Shafique, T. Mitra, and J. Henkel, "Improving mobile gaming performance through cooperative CPU-GPU thermal management," in *Design Automation Conference (DAC)*, 2016, pp. 1–6.

[28] J. Kong and K. Lee, "A DVFS-aware cache bypassing technique for multiple clock domain mobile socs," *IEICE Electronics Express*, vol. 14, no. 11, pp. 20 170 324–20 170 324, 2017.

[29] P. S. Patil, J. Doshi, and D. Ambawade, "Reducing power consumption of smart device by proper management of wakelocks," in *International, Advance Computing Conference (IACC)*, 2015, pp. 883–887.

[30] B. K. Reddy, A. K. Singh, D. Biswas, G. V. Merrett, and B. M. Al-Hashimi, "Inter-cluster thread-to-core mapping and DVFS on heterogeneous multi-cores," *IEEE Transactions on Multi-Scale Computing Systems*, no. 1, pp. 1–1, 2017.

[31] P.-H. Tseng, P.-C. Hsiu, C.-C. Pan, and T.-W. Kuo, "User-centric energy-efficient scheduling on multi-core mobile devices," in *Design Automation Conference (DAC)*, 2014, pp. 1–6.

[32] W. Jung, K. Kim, and H. Cha, "Userscope: A fine-grained framework for collecting energy-related smartphone user contexts," in *International Conference on Parallel and Distributed Systems (ICPADS)*, 2013, pp. 158–165.

[33] P. Mercati, F. Paterna, A. Bartolini, L. Benini, and T. Rosing, "WARM: Workload-aware reliability management in linux/android," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 9, pp. 1557–1570, 2016.

[34] H. Yang and S. Ha, "Power optimization of multimode mobile embedded systems with workload-delay dependency," *Mobile Information Systems*, vol. 2016, 2016.

[35] Y. Huang, Z. Zha, M. Chen, and L. Zhang, "Moby: A mobile benchmark suite for architectural simulators," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 45–54.

[36] X. Dong, S. Dwarkadas, and A. L. Cox, "Characterization of shared library access patterns of android applications," in *International Symposium on Workload Characterization (IISWC)*, 2015, pp. 112–113.

[37] Android Developers. Monkey Command-Line Emulator. (2018). [Online]. Available: https://developer.android.com/studio/test/monkey

[38] L.-T. Duan, M. Lawo, I. Rügge, and X. Yu, "Power management of smartphones based on device usage patterns," in *Dynamics in Logistics*. Springer, 2017, pp. 197–207.

[39] I. Abubakar, S. Khalid, M. Mustafa, H. Shareef, and M. Mustapha, "Application of load monitoring in appliances energy management–a review," *Renewable and Sustainable Energy Reviews*, vol. 67, pp. 235–245, 2017.

Anway Mukherjee (S'18) received his B.S. degree in electronics and communications from West Bengal University of Technology, India, in 2011. He is currently working towards his Ph.D. degree at the Department of Electrical and Computer Engineering, Virginia Tech, VA, USA. His research focuses on energy-aware and resource-aware hardware-software co-design of real-time embedded systems.

Thidapat Chantem (S'05-M'11-SM'18) received her Ph.D. and Master's degrees from the University of Notre Dame in 2011 and her Bachelor's degrees from Iowa State University in 2005. She is an Assistant Professor in Electrical and Computer Engineering at Virginia Tech. Her research interests include real-time embedded systems, energy-aware and thermal-aware system-level design, cyber-physical system design, and intelligent transportation systems.