An Integrated Energy Management Framework for Multiple Side-by-Side Applications on Smartphones

Anway Mukherjee and Thidapat Chantem
Department of Electrical and Computer Engineering, Virginia Tech, USA. Email: {anwaym, tchantem} @vt.edu

***Abstract*—While simultaneous side-by-side execution of multiple applications in split-screen mode on smartphones improves quality-of-service (QoS) for end-users, it also results in increased power consumption and reduced battery lifetime. Saving energy when multiple concurrent applications run side-by-side is extremely challenging since applications often utilize the same shared system resources, e.g., memory, at the same time, and resource needs change over time. We present an application-aware integrated system-level energy management framework that (1) leverages applications' offline profiles to detect dynamic changes in resource usage patterns of the workload of applications, and (2) opportunistically throttle (and later redeem) applications based on their instantaneous resource usage characteristics by dynamically adjusting both the voltage and frequency settings of the processor and memory bandwidth at runtime to achieve the most energy-efficient configuration subject to QoS constraints. Experiments on a Pixel 2 smartphone show that our approach achieves an average energy reduction of 13% (16%) and up to 16% (20%) compared to the most closely related work [1] (and default Android governor) for different combinations of real-world applications running side-by-side in split-screen mode. Our approach is also able to reduce the energy consumption of newly installed real-world applications for which there exists no prior resource usage data by up to 12% (14%) when compared to [1] (and default Android governor).**

I. Introduction

Modern smartphones now increasingly support more features and functionality, which lead to improved application performance and quality-of-service (QoS). A fairly new smartphone feature, thanks to larger dynamic random-access memory (DRAM), is that several applications can share the device screen simultaneously. That is, a user could split the screen, composing a document on one side while viewing a video on the other side. However, currently available battery technology in smartphones cannot adequately support such technological advances, resulting in poor battery life in smartphone [2], or even worse, device overheating [3], [4].

Existing energy saving solutions for smartphones often focus on enabling independent fine-grained dynamic voltage and frequency scaling (DVFS)-based power management of distributed components or subsystems at different layers of abstraction [5], [6]. However, minimizing the power consumption of each component independently does not always result in an optimized solution. For instance, the overall system load is often used to adjust core voltage and frequency settings to optimize application QoS. Such a technique ignores memory usage, which has been shown to be application-specific [7] and which cannot be easily captured by monitoring system load alone.

This work was supported in part by the National Science Foundation under grant numbers CNS-1618979.

978-1-5386-5541-2/18/$31.00 ©2020 IEEE

While DVFS can significantly reduce energy consumption, the time overhead associated with dynamic voltage and frequency (VF) scaling is not insignificant and often adversely affects the overall performance of the system [8]. DFVS can also lead to more energy dissipation since each VF level switching incurs additional power state transitions at the device hardware-level. Existing work on DVFS policies either assumes no latency in VF scaling [1], [9], or intersperses such VF transitions without prior knowledge of the workload requirements [10]. An effective system-level energy management solution, therefore, must judiciously make use of power saving opportunities, by considering the varying resource usage pattern of an application over time. However, a major challenge is to detect and classify the dynamic resource usage pattern of individual applications within an execution context where multiple concurrent applications run side-by-side on the same shared system resources.

In this paper, unlike many previous works, e.g., [11] which limit the energy saving technique to processor cores only, we propose a coordinated energy management solution that aims to reduce the energy consumption of side-by-side applications on smartphones. We leverage an offline profiling tool [1] to detect and classify application workloads based on their dynamic resource usage patterns. We then present a technique to group applications with similar resource usage patterns, which can greatly improve the execution predictability, while reducing the runtime overhead of the energy management solution, as reported later in this work. In contrast, for concurrent applications running side-by-side with different resource usage patterns, we present a dynamic policy that can judiciously throttle (and later redeem) such applications to achieve significant power saving opportunities without incurring runtime overhead or sacrificing QoS.

The most closely related work to our proposed dynamic energy management framework is presented in [1]. Said work implemented an integrated system-level DVFS technique that detects instantaneous phase changes of the target application, and uses that information to find the most energy-efficient CPU frequencies and memory bandwidth without sacrificing QoS. While the approach aims to reduce the energy consumption of side-by-side applications on smartphones, it makes several restrictive assumptions. Firstly, it states that each application running side-by-side must utilize *separate* cores. Secondly, the work simulated system noise, and spatial and temporal contention of shared resources, by running a fixed lightweight application side-by-side with a real-world application, as opposed to capturing the actual interference

due to synchronous execution of multiple side-by-side real-world applications. Finally, it fails to dynamically detect phase changes of the overall system workload when *multiple* applications are running concurrently on the same processor core. We address these limitations in our proposed DVFS-based energy management framework by

1) Leveraging a lightweight phase detection tool [1] to record the instantaneous phase changes of concurrent applications, running side-by-side, and categorize application workloads based on their dynamic phase changes. Our approach also assists in the energy management of newly installed applications for which no prior resource usage data exists.
2) We group applications with similar resource usage patterns, and leverage an online controller [1] at runtime to select the most energy-efficient configuration of the system without sacrificing performance. For applications with different resource usage patterns, we propose a novel runtime overhead-aware energy saving policy which judiciously throttles the application performance, and later opportunistically compensates for the performance loss, to save system-wide energy consumption without sacrificing QoS.
3) We validate our proposed approach and assess its performance by comparing it with the most closely related work [1], 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 on our target smartphone, Pixel 2, running Android Oreo. Experimental results show that our approach is able to achieve an improvement of up to 16% in energy savings over the most closely related work [1].

II. Preliminaries

A. Platform

We chose Android OS [12] in this work since it is the most widely used open-source mobile OS. Android implements an abstraction of device-level DVFS framework, called the Android governors, to reactively manage the energy consumption of the device. The Pixel 2's governors can choose from a list of 14 CPU frequencies a processing core can operate at (for e.g., 0.3000, 0.4224, 0.6528, 0.7296, ..., 2.5728, 2.6496 in GHz). Similarly, a governor, known as *devfreq*, which manages memory bandwidth, has 13 levels to choose from (for e.g., 762, 1144, 1525, 2288, ..., 12145, 16250 in MBps). A *system configuration* is defined as any combination of the tuple (*CPU Frequency*, *Memory Bandwidth*) from the corresponding available choices of frequencies as shown in Table I.

B. Characterizing an Android Application

Android applications are governed by their *activity* list. An activity is the subroutine with an application that contains a list of *tasks* to be performed whenever the activity is invoked. An activity can be invoked through a trigger when the user interacts with the application. A typical lifecycle of an activity spans over three parts; (1) an actual trigger to start the activity, followed by (2) a set of functions performed in the background to set up the activity context, and finally (3) the activity context being rendered onto the screen. For instance, launching an application by tapping the application button triggers the launch-related activity that ultimately renders the application on the smartphone screen. All application-related activity is defined in its `AndroidManifest.xml` file. An activity, and its set of events, can be coarsely classified as either computation intensive or memory intensive. For instance, an activity triggered by tapping on the screen results in a series of events that contribute to the creation or deletion of multiple new execution contexts. Such events contribute to computation intensive operations [13]. Similarly, an application *window resizing* activity may result in memory intensive (or less computation intensive) events, partially due to related data reloading and image rendering tasks offloaded to a dedicated co-processor.

Previously, an application in Android was categorized as either being in the *foreground* context (i.e., an application currently visible on screen), or in the *background* context (i.e., an application with limited functionality). With split-screen mode of side-by-side execution, however, multiple concurrent applications can demand access to the available system resources at the same time, and run at peak functionality. This leads to an increased energy consumption, and poor battery life in smartphones [3].

III. Proposed Framework

There are multiple challenges associated with the implementation of an integrated energy management framework on Androids with split-screen mode of execution. For instance, existing work [1], [7], [11] fail to detect, segregate, and subsequently predict the dynamic resource usage pattern of individual applications within an execution context. A segregation of resource needs would help to isolate the resource utilization of each application from the overall resource usage among concurrent side-by-side applications while running on shared system resources. Only with such knowledge can we effectively reduce the energy consumption of all concurrent side-by-side applications without incurring any unnecessary overhead that can unfairly impact QoS.

In this work, we overcome the aforementioned challenge by proposing (1) a technique to categorize application *activity* contexts (defined in Section II-B) based on its dynamic resource usage pattern, and (2) a novel DVFS-based throttle-and-redeem policy for concurrent side-by-side applications to be run within an integrated energy management framework for minimum system-wide energy consumption with negligible effect on QoS. An overview of our proposed framework is shown in Figure 1.

We split our work into a two-step procedure; an offline profiling step followed by online execution. Firstly, we leverage the technique shown in [1] to run our target application (say τ_a) through an offline power-performance profiling step (Section IV). However, instead of simulating system noise through a lightweight application, we run τ_a side-by-side with another real-world application (say τ_b) to capture the expected variable system loads, and contention on shared resources due to concurrent side-by-side execution during actual runtime. While our approach can readily be extended to

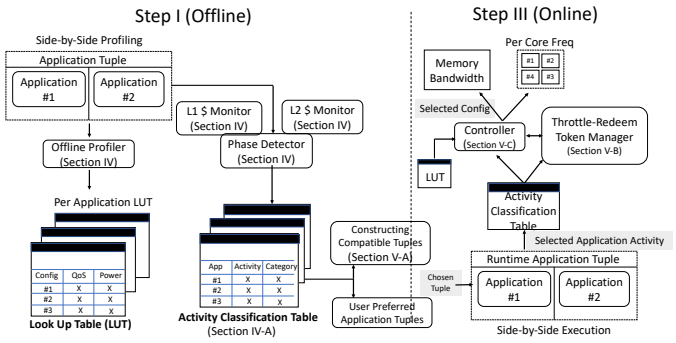

Fig. 1. The proposed framework is decoupled into (1) running applications side-by-side for offline profiling (Step I), (2) analyzing application-specific activity's resource usage pattern (Step II), and (3) an online framework to select the best energy-performance configuration (Step III).

multiple applications on split-screen, we restrict the number of applications sharing the split-screen to two in this work. This is due to the limitation of our hardware testbed's RAM capacity since all processes sharing the device screen run with equal activity context, resulting in the split-screen mode of execution using extremely high memory in order to keep all the application contexts alive. We also catalog said offline profile data into categories of applications for the runtime energy management of applications which do not have offline profile data (Section VII-B). Note, that we assign each application to separate core(s) to help isolate the resource usage (CPU load and memory traffic) of our target application (τ_a) when running simultaneously with another real-world application (τ_b) in split-screen mode during offline analysis.

Second, we propose a technique to categorize application activity contexts into either computation intensive contexts or memory intensive contexts (defined later in Section IV-A). This is accomplished by running the same application tuple (τ_a, τ_b) through an offline phase analysis tool (Section IV-A) similar to the technique shown in [1]. Said tool captures, over time, the application-specific offline memory traffic for each target application τ_a in the application tuple (τ_a, τ_b). Third, we propose an online token-based throttle-and-redeem DVFS policy at runtime where we opportunistically throttle the VF levels of an application's activity (acknowledged by issuing a token) during side-by-side execution, and compensate the throttled application by redeeming said token at a later time to ensure fairness. We also use a runtime controller framework (Section V-C), which relies on a throttle-redeem token manager (Section V), along with the offline profiles (Section IV-IV-A), to periodically select the voltage and frequency levels of associated cores and memory subsystem that minimizes the energy consumption without sacrificing target QoS.

IV. Offline Profiling and Analysis

In the first step, we leverage an offline power-performance profiler presented in [1] to construct a lookup table (LUT), an example of which is shown in Table I, to record the correlation between a given system configuration, performance, and power consumption for the target application (τ_a) in an application tuple (τ_a, τ_b) competing for resource demands by

TABLE I
An Example LUT for Target Application (τ_a) in the Application Tuple (τ_a, τ_b) running on Pixel 2

Config	CPU Freq (GHz)	Mem Bandwidth (MBps)	Normalized IPC	Power (mW)
1	0.3000	762	1	2258.9
2	0.6528	1525	1.3	2133.1
3	0.7296	2288	1.5	2375.0

concurrently running side-by-side. Note, that we limit the concurrent execution of application tuples (τ_a, τ_b) by assigning each application to separate core(s) for the ease of offline profiling and phase analyses [1]. Also, a combination tuple *(CPU Frequency, Memory Bandwidth)* constitutes a *system configuration* [1]. The normalized instructions per cycle (IPC) is given by [1],

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

where IPC_1, is the system configuration with the lowest CPU frequency and memory bandwidth, and IPC_i is the average IPC of the i^{th} system configuration.

In conjunction, we use a lightweight cache monitor and phase detector tool from [1], that runs simultaneously with the power-performance profiler, to monitor the memory traffic for a target application (τ_a) when executing side-by-side with another concurrent application (τ_b). This helps us predict a target application's resource usage patterns. We categorize a *memory intensive* phase of the application if $\frac{U_j}{S} \leq 1$, or a *computation intensive* phase when $\frac{U_j}{S} > 1$ such that [1],

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

where S is the *sensitivity level* of the target application, and defined as the minimum change in cache traffic that would cause any change in the memory bandwidth under the default Android governor [1]. Similarly, $out\_L1\_traffic$ and $out\_L2\_traffic$ are the dynamic L1 and LLC memory traffic data for the target application respectively. Next, we leverage the phase prediction tool to propose a technique to categorize our target application based on its instantaneous phase change pattern.

A. Classifying Activity Contexts

An Android application is divided into a series of *activity* contexts (Section II-B). A typical application-specific activity list consists of separate subroutine(s) to react to different user interactions. Therefore, we propose an offline analysis technique whereby the different phase changes during each activity context can be used to ascertain the resource usage pattern of said activity, and in turn the overall application. For a given target application, (1) we determine the start and finish times of each activity context, during which (2) we predict and record the number of phase changes, and finally (3) categorize each activity to be in computation intensive or a memory intensive execution context based on its resource usage data. We classify an application activity to be *computation intensive* if the total number of detected computation

TABLE II
APPLICATION-SPECIFIC ACTIVITY CLASSIFICATION TABLE FOR A LIST OF REAL-WORLD APPLICATIONS

Target App	Activity List	Category
Facebook	Touch_Launcher	Computation Intensive
	Touch_Back	Computation Intensive
	Touch_Down	Computation Intensive
	Touch_Video	Memory Intensive
	Touch_Resize	Memory Intensive
	...	...
MobileBench	Touch_Down	Computation Intensive
	Touch_Move	Computation Intensive
	Touch_Back	Computation Intensive
	Touch_Resize	Memory Intensive
	...	...

intensive phases are more than the total number of detected memory intensive phases within the period of execution for said activity. Similarly, for *memory intensive* activity, the total number of detected memory intensive phases is more than the total number of detected computation intensive phases during the activity context. Such activity-specific classifications are stored in the form of a table where each entry contains the following parameters: *Target App*, *Activity List*, and *Category*. Table II shows an example activity classification table (ACT) for a list of real-world applications.

In our example ACT (Table II), the Facebook application shows two computation intensive activities, namely, Touch_Launcher (i.e., app launch context) and Touch_Down (i.e., scroll down context), while the other two activities, Touch_Video (i.e., launch video context) and Touch_Resize (i.e., resize window context) are classified as memory intensive execution contexts. Similarly, another example application, MobileBench, contains three computation intensive activities, and one memory intensive activity. When an application tuple runs for the *first* time, we construct the activity classification profile for a target application, and store it for future runs. The data collected in this step later helps in the runtime controller framework for online energy management.

While our offline power-performance LUTs (Section IV), and application-specific ACTs may incur a storage cost as low as 5 KB, the overall offline data set may exponentially increase with scaling number of installed applications in resource constrained smartphones. We address this challenge by maintaining a catalogue of offline profiles, each broadly classified as either *computation intensive* or *memory intensive* on the basis of their resource usage pattern. Any *newly installed* application, that shares similar resource usage patterns can reuse these profiles (see next section for more details). Selecting which profile to use is performed using a nearest neighbor (NN)-classification algorithm [14].

V. ONLINE ENERGY MANAGEMENT

Our objective is to design a dynamic voltage and frequency scaling (DVFS) policy that can achieve system-wide energy reduction while maintaining the desired QoS. Heretofore, we have limited the concurrent execution of application tuples (τ_a, τ_b) by assigning each application to separate core(s) for the ease of offline profiling and phase analyses. However, in reality, two or more applications may need to share a core when the total number of active applications is large, and often lead to resource contention due to concurrent accesses to the other shared resources. Another key challenge is to manage the runtime overhead associated with each change in voltage and frequency levels. This is because the prescribed modification to the system configuration due to such a change at the hardware level incurs a significant overhead (up to 13 ms for our testbed). For instance, an aggressive energy saving policy would not only fail to achieve expected performance efficiency, but also, at worst, lead to more energy dissipation [15].

We, therefore, propose an online DVFS-based energy management solution that (1) utilizes the offline classification of application-specific activity contexts (Table II) to group side-by-side applications with similar resource usage patterns into compatible application tuple(s) (Section V-A), (2) uses a token-based dynamic energy management policy that judiciously throttles application performance (and later compensates for the performance loss) during the side-by-side execution of applications that do not have compatible resource usage patterns (Section V-B), and finally (3) design a controller framework to determine the voltage frequency level(s) that optimizes for both power reduction and QoS requirements (Section V-C).

A. Constructing Compatible Application Tuples

The motivation behind our approach is based on the observation that multiple activities of different applications not only share the same set of events but also show similar resource usage patterns. For instance, in our example activity classification table (Table II), the two applications Facebook and MobileBench have similar activities, i.e., Touch_Down, Touch_Back etc., with compatible computation intensive phase patterns. Grouping applications with similar phase patterns maximizes the duration of usage of shared resource (e.g., running on the same core) under similar system configurations. This has a two-fold advantage. Firstly, running application processes which are *in-phase* present a highly predictable and almost constant system load, thereby allowing shared resources (processor cores, peripherals) to remain in the same system configuration (CPU frequency, memory bandwidth) for longer duration(s). Secondly, reducing the frequent changes in system configuration leads to reduced runtime overhead, along with improved system-wide power consumption.

We implement a learning-based classification approach, called the nearest neighbor (NN) algorithm [14], for constructing the best application tuples (τ_a, τ_b) from a list of real-world applications. For each candidate application (say τ_a), we use its activity classification table data collected offline (Section IV-A) to choose its nearest neighbor application (say τ_b) from the training data set. There are multiple advantages of using the NN-based classifier in our work. The NN-algorithm is highly flexible, i.e., new data entry can be classified and adapted in real-time into the training data set within a single run. Furthermore, it is resistant to noisy training data [14]. However, NN-algorithm is inherently slow and does not scale well with increasing size of training data set. We overcome these challenges by (1) bounding the size of the training set by limiting activity-specific offline profile data (Section IV-A)

and, (2) storing the data points in two-dimensional tree for faster computation. For an application τ_a with offline profiles, we use its activity classification table entry as input to the NN-classifier to find another application τ_b with activities that are compatible to phase change patterns of τ_a. The tuple thus constructed forms a resource-aware application tuple (τ_a, τ_b) for side-by-side execution. Applications which do not have an offline profile can be classified and included in the training data set with minimum overhead in real-time (Section VII-B).

B. Token-Based Throttle-Redeem Policy

The user can choose any combination of concurrent applications to run side-by-side. In reality, such an application tuple may not be classified as a compatible application tuple. These applications, containing highly dissimilar resource usage patterns (including pareto-optimal cases), when run concurrently on the same core may result in frequent activity-specific phase changes. Therefore, they too need to be judiciously managed so that the voltage frequency scaling results in an optimal trade-off between the system-wide energy reduction and maintaining the target QoS. We propose a runtime overhead-aware token-based DVFS policy, which avoids unnecessary voltage frequency level transitions while reacting to frequent changes in activity contexts.

Throttling technique: Each application tuple (τ_a, τ_b) must contain a *throttled* application τ_a, and a *normal* application τ_b. The selection is based on our observation (during offline profiling) that certain applications present a higher probability of energy saving with negligible effect on its target QoS. A normal application is targeted for optimum energy reduction, while a throttled application is opportunistically assigned lower than the desired VF levels during their activity runtime. This may have an unfair effect on their desired QoS. For instance, a throttled application in computation intensive (or memory intensive) activity/phase may be assigned a lower than desired CPU frequency (or memory bandwidth) directly resulting in runtime overhead and performance degradation. However, at a later point in execution, the throttled application gets to redeem such unfair treatment by being allotted extra CPU cycles and/or memory bandwidth to compensate for lost performance, thus eventually obtaining the desired QoS.

Candidate for throttling: Our proposed solution operates in the granularity of application-specific activity list. Each activity can be classified as computation intensive (or memory intensive) based on the *majority* of computation intensive (or memory intensive) phases that are registered during the execution context of an activity (Section IV-A). Therefore, the majority of phases within an activity context can range anywhere between 50% to 100%. The choice of application to be throttled is based on the percentage of phases that constitute the majority within the activity context. For instance, an application (τ_a) with a computation intensive activity where 60% of its phases are computation intensive (and the rest being memory intensive) will be chosen as a candidate for throttling over an application (τ_b) with a memory intensive activity where 90% of its phases are memory intensive. The throttled application experiences reduced CPU frequency and higher memory bandwidth. However, the unfair CPU throttling, in this scenario, only affects τ_a's 60% of the phases (computation intensive) while the rest 40% of the phases, which are memory intensive, will benefit from the throttling. Such an informed mis-prediction limits the performance loss for a throttled application.

Compensation technique: The decision to redeem a token is dependent on multiple factors. First, a throttled application can opportunistically redeem its token(s) when current core-specific system configuration matches with the resource usage pattern requirements of an activity ready for execution. This allows the DVFS controller to estimate a new system configuration to compensate for the lost performance. For instance, let us consider that the core is currently set for computation intensive execution (i.e., high CPU and low memory bandwidth frequency), and the activity to be run is also computation intensive. In such a case, the controller can decide to estimate a new system configuration that can select a higher CPU frequency (through *redeemQoS*) to save computation time and thus, improve the QoS. Second, the number of tokens per-application is limited by the number of available tokens for distribution. The maximum tokens that can be generated by the token manager can be pre-determined by the system administrator. This limits the amount of throttling that can be unfairly imposed on an application with negligible QoS degradation.

The overview of our proposed solution is described in Algorithm 1. We start with an application tuple (τ_a, τ_b), with dissimilar resource usage patterns, chosen for concurrent side-by-side execution. First, we select an application (say τ_a) for *throttling*, detect its activity ready for execution (Lines $26-29$), and obtain the information about its resource usage pattern from the activity classification table (Table II). Second, we use this information as input to a DVFS controller framework (described later in Section V-C) whose job is to judiciously select the best system configuration (CPU frequency, memory bandwidth) to minimize the system-wide energy consumption with negligible effect on application QoS (Lines $2-17$). Once the selected activity is assigned a core, we obtain the current system configuration of the core, a necessary requirement for an informed runtime overhead-aware DVFS policy (Line 5). For instance, let us consider that the core is currently set for memory intensive execution (i.e., low CPU and high memory bandwidth frequency). However, the activity to be run is computation intensive, i.e., it requires high CPU and low memory bandwidth frequency for optimal QoS. If the controller decides not to modify the current system configuration (thus choosing to save time overhead over maintaining the desired QoS), it may unfairly throttle the current activity. To remedy this, we utilize a *token manager* to generate a token for the throttled application that can be redeemed later to compensate for the performance loss (Lines $18-24$).

C. Controller

Heretofore, we have (1) constructed application tuples with similar dynamic phase change patterns (Section V-A), and (2) designed a token-based DVFS policy to manage application

Algorithm 1 Throttle-Redeem Policy

Input: application tuple (τ_a, τ_b) to be executed side-by-side

1: ▷ High-level overview of throttle-redeem policy implementation
2: **function** DVFS_CONTROLLER(τ_a, τ_b, detectedPhase, #cores)
3: **for** $\tau_i \in (\tau_a, \tau_b)$ **do**
4: **for** $m_i \in \#cores$ **do** ▷ If previous config was different
5: **if** PREV_PHASE(τ_i, m_i) != detectedPhase **then**
6: **if** TOKEN_MANAGER(τ_i, throttle) **then**
7: Do not change VF configuration
8: **else** ▷ If throttle tokens are not available
9: selectedConfig ← SEARCH_LUT(τ_i, targetQoS)
10: Assign selectedConfig to m_i and memory BW
11: **else** ▷ If prev config matches curr requirements
12: **if** TOKEN_MANAGER(τ_i, redeem) **then**
13: selectedConfig ← SEARCH_LUT(τ_i, redeemQoS)
14: Assign selectedConfig to m_i and memory BW
15: **else** ▷ If compensation is not allowed
16: selectedConfig ← SEARCH_LUT(τ_i, targetQoS)
17: Assign selectedConfig to m_i and memory BW
18: **function** TOKEN_MANAGER (τ_i, actionToPerform)
19: **if** actionToPerform == throttle **then**
20: **if** tokenAvailable **then**
21: TokenValue ++ ▷ Assign new token **return** true
22: **else** ▷ Redeem prev assigned token
23: **if** redeemAvailable **then**
24: TokenValue - - , RedeemValue ++ **return** true
25: ▷ Find phase of current activity from Activity Classification Table (ACT)
26: **function** DETECT_ACTIVITY(τ_a, τ_b)
27: **for** $\tau_i \in (\tau_a, \tau_b)$ **do**
28: **if** FIND_ACTIVITY(τ_i) **then** ▷ When an activity is ready to run
29: detectedPhase ← SEARCH_ACT(τ_i)
return detectedPhase
30: **function** MAIN(τ_a, τ_b)
31: detectedPhase ← DETECT_ACTIVITY(τ_a, τ_b)
32: DVFS_CONTROLLER(τ_a, τ_b, detectedPhase, #cores)
33: **return**

tuples with different resource usage patterns (Section V-B). However, at runtime, we still need to vote for the best system configuration based on the instantaneous phase of an application activity and/or throttle-redeem decision(s) to optimize system-wide energy consumption while maintaining the desired QoS. We leverage the controller framework in [1] to determine said trade-off between application performance (i.e., QoS) and its system-wide energy dissipation as shown in Algorithm 1 (Lines $2-17$). The inputs to the controller are (1) the LUT table(s) for a given application tuple (Table I), (2) independent target QoS value(s) to maintain the performance of each application in the tuple executing side-by-side, and (3) the decision from the throttle-and-redeem policy (Section V-B), along with the relevant activity-context and its predicted resource usage pattern information from Table II. The controller framework, with token manager (Section V-B), decides when, and by how much, to throttle or compensate an application. As will be shown in Section VII-D, the overhead of our proposed runtime controller is negligible.

VI. Experimental Setup

A. Experimental Platform

We implemented and tested our energy management framework on a real-world smartphone, Pixel 2. It supports a default energy management framework through several DVFS policies. Pixel 2 has a Qualcomm Snapdragon 835 chipset (with big.LITTLE architecture switcher disabled) that extends the necessary hardware and driver functionality to perform CPU usage and cache-level traffic monitoring. Pixel 2 runs Android Oreo. Starting from Android Nougat, all higher versions of Android (including Android Oreo) supports the split-screen mode of side-by-side execution for multiple applications on the same device screen.

B. Applications with Offline Profiles

We select a list of five of real-world open-source applications assuming that these applications already have their offline profiles stored in the system. The list of applications is as follows; (1) *VidCon* is a memory-intensive video converter, (2) *MobileBench* simulates a web browser and has both computation-intensive and memory-intensive phases (3) *Pokemon Go* is a memory-intensive gaming application, (4) *Facebook* is a reactive online application with multiple processes, each of which uses different system resources, and, (5) *Spotify* is another online audio and video streaming application.

C. Applications without Offline Profiles

We also consider another list of five newly installed real-world open-source applications for which we assume that no prior offline profile exists. The list of applications is as follows; (1) *Media Converter* is a video converter application with similar resource usage patterns as MobileBench, (2) *Android Browser* is the default web browser which comes with the device, (3) *Fruit Ninja* a popular gaming application with similar resource requirement to Pokemon Go, (4) *YouTube* is an online audio and video streaming application and, (5) *Amazon Music*, which is another online audio and video streaming application.

D. Evaluation Metrics

Our objective is to evaluate the effectiveness of our proposed framework with respect to the application performance and system-wide energy consumption. We use IPC and application makespan as metrics to define application performance. Both of these evaluation parameters have been shown to be good indicators of both CPU and memory performance, and are highly sensitive to system workload [1], [7].

We compared our approach against three most closely related works; the first work is the most closely related energy management framework on side-by-side execution model on Androids by Mukherjee et.al. [1] (henceforth denoted by MU). Moreover, since the authors in [1] compare their work against (1) the default Android governors (denoted as DG), and (2) an energy management policy for concurrent workload in embedded systems (henceforth referred to as WC) by Reddy et.al. [11], it is only fair for us to evaluate our proposed policy against these two as well.

VII. Results

A. Applications with Offline Profiles

Given a set of real-world applications that have been profiled offline *a priori*, our benchmark suite consists of application tuples consisting of two real-world applications from our

TABLE III
SUMMARY OF IPC REDUCTIONS, ENERGY SAVINGS, AND MAKESPAN INCREASE OF PROFILED APPLICATIONS USING PROPOSED APPROACH COMPARED TO THE DEFAULT ANDROID GOVERNORS (DG), CONCURRENT WORKLOAD CLASSIFICATION TECHNIQUE (WC) [11], AND THE CLOSEST EXISTING WORK (MU) [1] ON PIXEL 2

Application Tuple	IPC reductions (%)			System-wide energy savings (%)			Makespan increase (%)		
	DG	WC	MU	DG	WC	MU	DG	WC	MU
VidCon/ Spotify	-1.4/ -2.1	+2.8/ +1.9	+3.7/ +3.8	+14.1	+11.5	+13.2	+2.3/ +3.6	-4.1/ -9.3	-3.6/ -11.2
Facebook/ MobileBench	-3.1/ -1.8	+2.2/ +2.5	+6.9/ +4.4	+19.6	+12.5	+16.1	+5.2/ +2.3	-3.5/ -1.6	-7.3/ -4.9
Spotify/ Pokemon Go	-1.7/ -0.5	+1.3/ +2.6	+2.4/ +3.1	+16.5	+13.1	+11.8	+2.2/ +1.9	-6.5/ -0.9	-8.2/ -1.2

TABLE IV
SUMMARY OF IPC REDUCTIONS, ENERGY SAVINGS, AND MAKESPAN INCREASE OF APPLICATIONS WITHOUT OFFLINE PROFILES USING PROPOSED APPROACH COMPARED TO THE DEFAULT ANDROID GOVERNORS (DG), CONCURRENT WORKLOAD CLASSIFICATION TECHNIQUE (WC) [11], AND THE CLOSEST EXISTING WORK (MU) [1] ON PIXEL 2

Application Tuple	IPC reductions (%)			System-wide energy savings (%)			Makespan increase (%)		
	DG	WC	MU	DG	WC	MU	DG	WC	MU
Media Converter/ Amazon Music	-2.1/ -1.8	+2.3/ +0.7	+5.9/ +3.4	+12.2	+8.5	+11.6	+1.4/ +2.6	-3.1/ -4.8	-4.3/ -5.5
YouTube/ Android Browser	-2.5/ -0.5	+1.6/ +1.1	+3.5/ +3.7	+13.6	+6.0	9.2	+3.3/ +3.4	-2.1/ -3.7	-3.3/ -5.6
Amazon Music/ Fruit Ninja	-1.6/ -2.8	+4.1/ +0.6	+6.3/ +4.1	+12.7	+8.5	+11.5	+2.8/ +7.2	-2.5/ -0.3	-4.2/ -2.2

given set of applications. The activity-specific resource usage patterns for each application in the corresponding application tuple may or may not be similar to each other. We report on the average value for performance and energy over 10 runs per benchmark tuple in Table III on Pixel 2.

When compared to the closest work (MU), our approach is able to save a significant amount of energy up to 20% for all applications considered, along with improvements in IPC up to 7% and an average reduction in makespan of up to 8%. Similarly, our approach can considerably save the system-wide energy up to 20% over the default governor (DG) for all applications considered, albeit with small hits in IPC of less than 3% and with an average increase in makespan of up to 5%. Next, we will briefly discuss our findings for a selected application tuple for brevity.

VidCon and Spotify form an application tuple with dissimilar resource usage patterns. Spotify, predominantly a computation intensive application, 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. On the other hand, VidCon is a highly memory intensive application. Therefore, we use the token-based throttle-redeem DVFS policy whenever activities of each application execute on the same core, thereby ensuring a resource-aware and latency-aware system-wide energy consumption. Across all governors, comparatively, our approach is able to save a significant amount of energy up to 14% for all applications considered, albeit with small reduction in IPC of less than 2% and with an average increase in makespan of up to 4%.

B. Application without Offline Profiles

In this set of experiments, we evaluate the robustness of our approach when used for a combination of applications for which no prior offline profiles exist. Assuming that the set of real-world applications listed in Section VI-B have their offline profile readily available, we present a set of newly installed applications (in Section VI-C), all of which have not previously been profiled offline. We tested our proposed framework on the new applications running on Pixel 2 smartphone, and using the offline profiles of the existing applications.

TABLE V
IPC REDUCTIONS, ENERGY SAVINGS, AND MAKESPAN INCREASE OF AMAZON MUSIC (RUNNING WITH FRUIT NINJA) WITH VARYING APPLICATION USAGE PATTERNS ON PIXEL 2

Application tuple	IPC reductions (%)		System-wide energy savings (%)		Makespan increase (%)	
	Monkey	User	Monkey	User	Monkey	User
Amazon Music/ Fruit Ninja	-1.6/ -2.8	-1.4/ -1.2	+12.7	+10.5	+2.8/ +7.2	+1.9/ +5.1

We compared our approach against the DG, WC, and MU, and reported the average value of application performance and energy consumption over 10 runs per benchmark tuple in Table IV. Overall, our approach is able to save a significant amount of energy up to 14% for all applications considered, albeit with small hits in IPC of less than 3% and with an average increase in makespan of up to 7%. Our results prove that the technique of reusing the catalogue of offline profile data for already installed applications can contribute to saving a significant amount of system wide-energy, albeit with small hits in IPC, and an average increase in makespan. We now briefly discuss our findings for a few of the selected application tuples for brevity.

Since Amazon Music work under a similar client-server module as Spotify, utilizing the existing offline profile of Spotify likely attributes to similar CPU usage and memory access patterns. Likewise, Media Converter utilizes the offline profile of VidCon that is present a priori. Since VidCon and Spotify form an application tuple with dissimilar resource usage patterns (as reported in Section VII-A), we utilize the token-based throttle-redeem DVFS policy for the application tuple Media Converter and Amazon Music. Our approach is able to save a significant amount of energy up to 13%, albeit with small reduction in IPC of less than 2% and with an average increase in makespan of up to 3%.

We now discuss a special pareto-optimal case of application tuple, Amazon Music running side-by-side with Fruit Ninja, where our proposed framework performs the worst on Pixel 2. Amazon Music uses the the already existing offline profile of Spotify, while Fruit Ninja leverages an already existing gaming application profile, Pokemon Go. Overall, our approach is able to save a significant amount of energy up to 13% for all applications considered. However, it incurs an IPC reduction of 3% and an average increase in makespan as high as 7%. The primary reason for such QoS degradation can be attributed to a very low degree of shared libraries between the two gaming applications, resulting in a erroneous classification of application resource usage profile. In such a case, we can either (1) increase the granularity of the offline profile to improve the phase detection mechanism, or (2) create an additional LUT at runtime for future use.

C. Variable Usage Patterns

In order to test the effect of end users' varying usage behavior pattern on application performance and energy savings, we experimented on different combinations of application tuples from a given set of real-world applications. Let us consider an example application which performs worst with our proposed framework, i.e., the tuple consisting of Amazon Music and Fruit Ninja running side-by-side in split-screen mode on Pixel 2. We run the application tuple for (1) 5 runs where we simulate pseudo-random usage patterns using `Monkey` [16]; and (2) 5 runs with an actual user. We report our findings in Table V. The results indicate negligible differences in performance metrics with varying usage patterns between `Monkey` and an actual user.

D. Overhead

While the time overhead associated with our offline phase is not a bottleneck, we need to assess the overhead of the online phase, which can be categorized into a setup phase and a runtime phase. The setup phase classifies application tuples based on their resource usage pattern. The tuple thus formed will execute side-by-side at runtime. A controller, assisted by a lightweight token-based throttle-redeem tool, changes CPU frequency and memory bandwidth on the fly. The controller, therefore, is the biggest contributor to the timing overhead of our proposed solution. We set the period of the controller to be 1 s (safe limit to perform all controller functions). The timing overhead can be attributed to (1) searching the activity classification table to attribute the current application activity to be executed (less than 6 ms), (2) instantaneous token generation and/or decision-making as to when to redeem the token (less than 8 ms), (3) LUT search (about 9 ms on average), and (4) actually setting new core frequencies and memory bandwidth. Thus, the computation overhead of our online phase is quite negligible. We choose an application tuple Amazon Music and Fruit Ninja to evaluate the timing overhead associated with our policy, as this tuple performs worst within our framework. Table VI shows the incurred overhead on Pixel 2 smartphone respectively, with varying duration of controller period. The higher performance loss with increasing controller period can be attributed to an inherent design limitation reported in [1]. Since we leverage that work in our implementation of the offline performance and phase profiler, we see a similar trend of increased timing overhead with scaling controller periods irrespective of our choice of smartphone devices.

TABLE VI
SUMMARY OF IPC REDUCTIONS, ENERGY SAVINGS, MAKESPAN INCREASE, AND OVERHEAD OF AMAZON MUSIC (WITH FRUIT NINJA) USING PROPOSED APPROACH COMPARED TO DEFAULT GOVERNOR WITH VARYING CONTROLLER PERIODS ON PIXEL 2

Period (in s)	IPC hits (%)	Energy savings (%)	Makespan increase (%)	Overhead
1	-1.6 (-2.8)	+12.7	+2.8 (+7.2)	4%
2	-1.7 (-2.8)	+12.6	+2.8 (+7.2)	5%
3	-2.9 (-3.2)	+9.4	+4.2 (+7.9)	6%
4	-5.6 (-8.2)	+7.0	+8.5 (+11.8)	6%
5	-6.1 (-9.2)	+4.4	+11.2 (+13.6)	7%

VIII. CONCLUSIONS

In this article, we presented a holistic system-level energy management solution for resource constrained embedded systems, especially smartphones. We augmented an existing fine-grained lightweight offline profile-based tool to capture application-specific performance and memory traffic data, and detect instantaneous phase changes of applications. An online controller leverages application-specific phase change profiles to find the most energy-efficient CPU frequencies and memory bandwidth without sacrificing QoS. Experiments on the Pixel 2 smartphone with real-world Android applications validate the feasibility of the proposed approach.

REFERENCES

[1] A. Mukherjee and T. Chantem, "Energy management of applications with varying resource usage on smartphones," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2416–2427, 2018.

[2] 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.

[3] Android Developers, "Android 8.0 Behavior Changes," https://developer.android.com/about/versions/oreo/android-8.0-changes.html, 2018.

[4] ——, "Background Location Limits," https://developer.android.com/about/versions/oreo/background-location-limits.html, 2018.

[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] 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

[7] 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.

[8] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom *et al.*, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *2010 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2010, pp. 108–109.

[9] K. Ma, X. Li, M. Chen, and X. Wang, "Scalable power control for many-core architectures running multi-threaded applications," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2011, pp. 449–460.

[10] N. Ioannou, M. Kauschke, M. Gries, and M. Cintra, "Phase-based application-driven hierarchical power management on the single-chip cloud computer," in *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2011, pp. 131–142.

[11] B. K. Reddy, G. V. Merrett, B. M. Al-Hashimi, and A. K. Singh, "Online concurrent workload classification for multi-core energy management," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 621–624.

[12] Android Developers, "Android SDK," http://developer.android.com/sdk/ndk/index, html, 2013.

[13] X. Li, G. Chen, and W. Wen, "Energy-efficient execution for repetitive app usages on big. little architectures," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[14] L. I. Kuncheva and L. C. Jain, "Nearest neighbor classifier: Simultaneous editing and feature selection," *Pattern recognition letters*, vol. 20, no. 11-13, pp. 1149–1156, 1999.

[15] Z. Lai, K. T. Lam, C.-L. Wang, J. Su, Y. Yan, and W. Zhu, "Latency-aware dynamic voltage and frequency scaling on many-core architectures for data-intensive applications," in *2013 International Conference on Cloud Computing and Big Data*. IEEE, 2013, pp. 78–83.

[16] Android Developers, "Monkey Command-Line Emulator," https://developer.android.com/studio/test/monkey, (2018).