Differential Energy Profiling: Energy Optimization via Diffing Similar Apps

Abhilash Jindal and Y. Charlie Hu, *Purdue University and Mobile Enerlytics, LLC*

https://www.usenix.org/conference/osdi18/presentation/jindal

This paper is included in the Proceedings of the
13th USENIX Symposium on Operating Systems Design
and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Differential Energy Profiling: Energy Optimization via Diffing Similar Apps

Abhilash Jindal and Y. Charlie Hu
Purdue University and Mobile Enerlytics, LLC

Abstract

Mobile app energy profilers provide a foundational energy diagnostic tool by identifying energy hotspots in the app source code. However, they only tackle the first challenge faced by developers, as, after presented with the energy hotspots, developers typically do not have any guidance on how to proceed with the remaining optimization process: (1) Is there a more energy-efficient implementation for the same app task? (2) How to come up with the more efficient implementation?

To help developers tackle these challenges, we developed a new energy profiling methodology called *differential energy profiling* that automatically uncovers more efficient implementations of common app tasks by leveraging existing implementations of similar apps which are bountiful in the app marketplace. To demonstrate its effectiveness, we implemented such a differential energy profiler, DIFFPROF, for Android apps and used it to profile 8 groups (from 6 popular app categories) of 5 similar apps each. Our extensive case studies show that DIFFPROF provides developers with actionable diagnosis beyond a traditional energy profiler: it identifies non-essential (unmatched or extra) and known-to-be inefficient (matched) tasks, and the call trees of tasks it extracts further allow developers to quickly understand the reasons and develop fixes for the energy difference with minor manual debugging efforts.

1 Introduction

Despite the prevalence of smartphones, the user experience has remained severely limited by their battery life. As such, major mobile platform vendors such as Apple and Google have taken initiatives encouraging app developers to take effort optimizing their apps [7, 26].

The typical development cycle for optimizing the energy drain of mobile apps is similar to that for optimizing the running time of traditional software – iterating the process of (1) finding hotspots in the app source code that contribute to a significant portion of the total app energy drain, and then (2) determining whether and how the energy hotspots can be restructured to drain less energy.

However, modern mobile apps are highly complex, easily consisting of millions of lines of source code and third-party software, and interacting with the OS-provided frameworks in complex ways. Without the help of automatic tools, even finding energy hotspots in the app source code by developers would be very hard.

To this end, mobile app energy profilers (*e.g.,* [32, 31]) made a major step forward by providing a foundational energy diagnostic tool that automatically identifies energy hotspots in the app source code. However, these profilers only help with the first step of the app energy optimization process, because after presented with the energy hotspots, developers typically do not have any guidance on *whether* and *how* the energy hotspots can be restructured to drain less energy.

To help app developers with this remaining challenge in the energy optimization process, in this paper, we develop a new energy profiling methodology called *differential energy profiling* (or *energy diffing* for short) that can automatically uncover more efficient implementations of common app tasks, and in doing so, not only determines whether an energy-hotspot code segment can be optimized, but also gives hints on how to optimize it.

The basic idea behind differential energy profiling is intuitive: if we can find a set of similar apps by different developers that implement many identical app tasks, chances are the implementations differ and will have different energy footprint. Directly comparing their source-code energy profile generated by an energy profiler should expose more efficient implementation from the less one for the same app tasks.

In this work, we first make three key observations about the uniqueness of the mobile app marketplace and common mobile app development practice: (1) Because of the low barriers to entry of app development, for every popular app in the app market, there are typically a few dozen competing apps that implement similar or identical app functions or app features. (2) Using a traditional energy profiler, we profiled 8 selected app groups from 6 popular app categories from Google Play, each consisting of 5 similar apps and 5 different versions of one of them, and we found similar apps can differ significantly in energy drain in performing similar app functions. (3) We further observe that mobile apps make heavy use of the common framework services provided by modern mobile OSes such as the Android framework, and our profiling analysis of the above 8 app groups has shown similar apps in each group share 38.6% to 81.9% of the same framework method calls and spent 44.0% to 96.7% of their app energy drain in calling framework services.

Observations (1) and (2) suggest it is possible to learn more efficient implementation of the same app task by comparing the energy profiles of similar apps, but if apps have very different source code structures, such comparison may not be effective. Observation (3) affirms such comparison of similar apps is actually meaningful and potentially effective.

We then present the design and implementation of such a differential energy profiler, DIFFPROF. Developing DIFFPROF faces three challenges: (1) What should be the diffing granularity? (2) How to identify the diffing units in the source-code energy profiler output of each app? (3) How to actually diff the energy profiles of similar apps? We address these challenges as follows:

(1) Using app tasks as the diffing granularity. We argue following the widely adopted modular programming principle, an app is typically structured to implement a number of app features or tasks. Since the ultimate goal of energy diffing is to uncover more efficient implementations of app tasks, the ideal diffing granularity that most directly helps the developers should be an app task.

(2) Characterizing how app tasks manifest in call trees. Diffing at the app task granularity requires identifying app tasks in the call tree output by a source-code energy profiler. To address this challenge, we examine the call trees for top 100 non-game apps and find that app tasks manifest themselves as Erlenmeyer flask-shaped slices (denoted as EFLASKS) represented in (call path, framework-method, subtree) tuples where the call path identifies the context of the task, the framework-method is used to invoke the framework service to accomplish the task, and the subtree captures the particular execution of the framework service.

(3) An efficient EFLASK matching algorithm. We give insights on how and why different implementations (EFLASKS) of the same app task differ which motivates the need for approximate EFLASK matching. We develop to our knowledge the first EFLASK-shaped tree slice matching algorithm that accurately finds similar EFLASKS corresponding to the same app task.

To demonstrate its effectiveness, we implemented DIFFPROF on top of a state-of-the-art energy profiler EPROF [32] for Android, and compared it to EPROF in profiling 8 groups (from 6 popular app categories in Google Play) of 5 similar apps each. We show DIFFPROF accurately identifies matched tasks that account for 79% of the app total energy drain on average as well as unique tasks (21% of total energy on average), in similar apps.

Further, we conducted 12 case studies to show that DIFFPROF provides developers with actionable diagnosis beyond a traditional energy profiler: (1) When EPROF identifies energy bottlenecks, they may be necessary or not inefficient; DIFFPROF identifies non-essential (unmatched or extra) and known-to-be inefficient (matched) tasks; (2) The EFLASK of tasks extracted by DIFFPROF further shows the details of the more efficient implementation, which allows the developer to quickly understand the reasons for the energy difference with minor manual debugging efforts (*e.g.,* setting breakpoints) since the developer did not author the similar app. Out of the 12 inefficient or buggy implementations in 9 apps, 3 of which have already been confirmed by developers, and removing them reduces app energy drain by 5.2%–27.4%.

This work makes the following contributions:

- It presents differential energy profiling, which tackles a key challenge faced by app developers in optimizing app energy drain - determining whether and how energy hotspots in app source code can be optimized, by identifying and comparing different implementations of the same tasks in similar apps.
- It presents DIFFPROF, an energy diffing tool for Android mobile apps. It describes DIFFPROF's implementation and the core algorithm that finds approximate matching of Erlenmeyer flask-shaped slices in calling context trees of similar apps, and demonstrates its benefits over traditional energy profilers.

2 Key Insights

The DIFFPROF design is motivated by three key insights we make about the mobile app market.

2.1 Competing/similar apps are abundant

Our first observation is about a unique phenomenon of the mobile app marketplace: *(O1) for every popular app, there are typically a few dozen competing apps that implement similar or identical app functions or app features.* The top 100 non-game apps in Google Play belong to 34 functionally similar app groups and each of these categories consists of many competing popular apps. Table 1 lists 8 such similar app groups with apps in the top 100 as well as outside the top 100 apps; the majority of them have 50M+ downloads.[1] We see that many groups include over a dozen similar apps each. Moreover, similar apps, *e.g.,* competing apps such as Pandora and Spotify, or a popular app (Candy Crush Saga) and its dozens of clones, typically have similar user interactions. For example, the music playback screens of all music streaming apps have an album cover image, the song and the album title, a progress bar, elapsed and remaining time text, and buttons to control music playback, and every app performs music playback.

Table 1: Eight groups of similar apps from top 100 non-game apps, their competitors, and energy drain measurement. "*": Popular but not a top 100 app, "+": Pre-installed app.

App Category	App Group	Similar/Competing Apps	Max/min energy ratio	Perc. energy in framework
Communication	Messaging & calling	Whatsapp, Google Hangouts+, Facebook Messenger, BBM, Line, Wechat, Viber, Skype, Tango, Whatscall, Telegram, TextNow, imo	8.0	60.2% - 90.3%
	Email	Yahoo Mail, Gmail, Outlook, Android mail+, Aqua Mail*, Email For Any*, MailRU*, myMail*	4.6	56.6% - 90.9%
Music & Audio	Music streaming	Spotify, Pandora, Soundcloud, iHeartRadio, Youtube Music, Free music, Napster, Google Play Music+, Apple Music*	4.2	49.6% - 93.8%
Personalization	Launcher	GO Launcher, CM Launcher 3D*, APUS Launcher*, Solo Launcher*, Hola Launcher*	3.1	44.0% - 93.8%
Productivity	File explorer	ES*, FX*, Solid*, File explorer*, File manager*	5.3	89.3% - 94.9%
Shopping	Shopping	Wish, eBay, Amazon, Walmart, AliExpress, Kohl*, letgo*	3.2	83.1% - 96.7%
Tools	Antivirus	Supo Security, CM Security AppLock AntiVirus, 360 Security, AVG AntiVirus, DU antivirus, Mobile Security & Antivirus*, Kaspersky Antivirus Security*	2.8	53.5% - 91.3%
	Cleaning	Clean Master, DFNDR, Fast Cleaner - Speed Booster, Turbo cleaner, Power clean Lionmobi, OK clean lite, DU speed booster & cleaner*, Ccleaner*	3.6	78.5% - 93.9%

2.2 Similar apps differ in energy drain

Given the abundance of similar apps for every popular app, we next ask the question: how do they stack against each other in energy drain, in performing similar app functions? To answer this question, we profiled the similar apps in the 8 popular app categories on a Nexus 6 phone running Android 6.0.1 while connected to WiFi.

We use automated tests to perform identical actions on the similar apps in each group and measure the energy drained by these actions using EPROF. In particular, we use `UI Automator`, the Android black-box UI testing framework, which does not require app source code.

For each group of similar apps, we first write a generic base test that interacts with common UI elements. Next, for each app in the cluster, we launch the app on the phone and find the unique ids of all the UI elements involved in the base test using Android's `uiautomatorview` tool. Finally, we run the base test with app-specific UI element ids, thus performing homogeneous interactions across similar apps. The specific tests for the 8 app groups are listed in the sub-captions of Figure 1.

Figure 1 contrasts the total energy drain of 5 similar apps and 5 versions of 1 app under the same user interactions in each of the selected 8 app groups from Table 1. We observe that the maximal to minimal energy drain across the 5 apps in each group range between 2.8x to 8.0x, as shown in Table 1. We thus draw our second observation that *(O2) similar apps easily differ significantly in energy drain in performing similar app functions.*

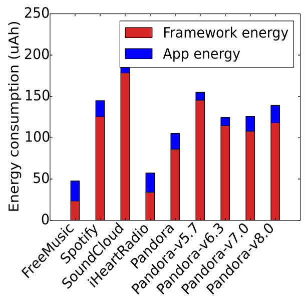
The above observation suggests that directly comparing the energy footprint of similar apps at the source-code level is promising to diagnose energy hotspots. However, such comparison will be fruitful only if their source code have significant overlap.

2.3 Framework services dominate app energy drain

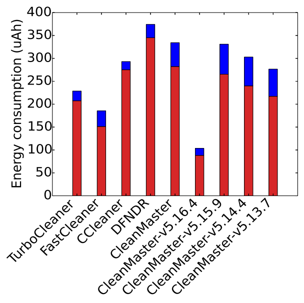
Our next observation is that mobile apps make heavy use of the common framework services provided by modern mobile OSes such as the Android framework. To simplify app programming, such frameworks implement and export to apps many services that implement commonly performed tasks, *e.g.,* the Android framework provides `LocationManager`, `DownloadManager`, `MediaPlayer`, and `WindowManager`, among others. Typically, an app presents requirements via configuration parameters to the services, and the services then perform the low-level work on the app's behalf. We hypothesize that the heavy usage of framework services leads to a high percentage of app energy drain occurring in these common services and the framework methods called in similar apps have a high overlap.

To confirm this hypothesis, we use EPROF to decouple the energy spent in app methods from those spent in framework services. First, we run `dexinfo` [5] on all the framework jar files located in `/system/framework/` on the phone to identify all the framework packages such as `android.view`, `dalvik.system` and `java.math`. Next, for each app, we identify all the framework methods in its energy profiling output belonging to these framework packages. Finally, we aggregate their energy drain to compute the total framework energy drain. The remaining energy drain is marked as app energy drain.

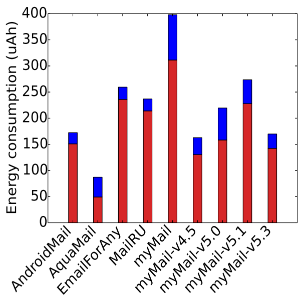
Our results (details for only 4 app groups are shown in Figure 2 due to page limit) show that the apps in the 8 app groups have significant pairwise overlap in the framework methods called during the profiling run, between 38.6% and 81.9% (61.7% on average). Further, Table 1 shows that a significant portion of the total energy of the apps in each group was spent in framework API calls, ranging between 44.0% to 93.8% for Launcher apps to



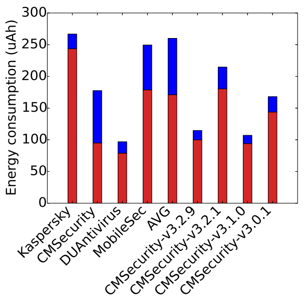
(a) Music streaming. Streaming for 30 seconds.



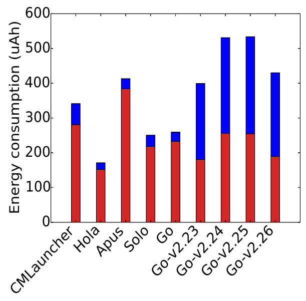
(b) Cleaning. Press clean and wait for operation to end.



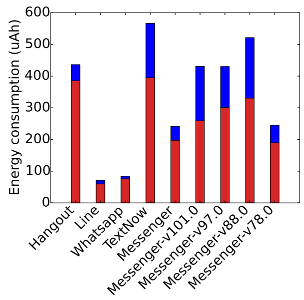
(c) Email. Compose an email and send.



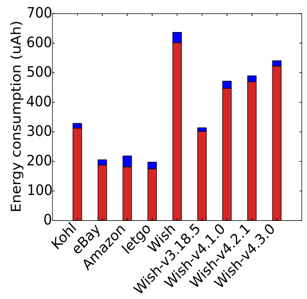
(d) Antivirus. Perform security scan.



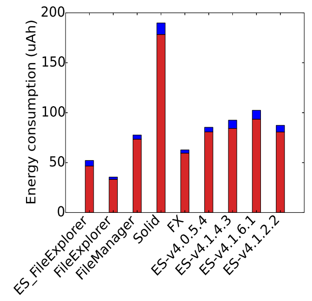
(e) Launcher. Slide from left to right every 2 seconds for 30 seconds.



(f) Instant messaging and calling. Send five messages to a friend.



(g) Shopping. Search for "socks", scroll the results and finally click on the first one.



(h) File explorer. Scroll the root directory and create a new directory.

Figure 1: Energy consumption of similar apps in 8 app groups. Energy drain numbers (in μAh) are direct output of EPROF, for the actual tests, which vary between 30 seconds to 1 minute long for different app groups.

between 89.3% to 94.9% for File explorer apps. We thus draw our third observation that *(O3) the heavy usage of framework services leads to a high percentage of app energy drain occurring in these shared services, up to over 90% of the app energy consumption.* This phenomenon suggests learning more efficient implementations of app functions by comparing their energy footprints not only is possible, but actually is a meaningful and practical approach.

3 How to Diff Energy Profiles?

The above three key insights suggest comparing the energy profiles of similar apps generated by a source-code energy profiler has the potential to automatically identify inefficiencies in implementing common app functions in similar apps. We call this approach *differential energy profiling*, or *energy diffing* for short.

Developing such a differential energy profiler has to address three challenges: (1) What is the diffing granularity? (2) How to identify the diffing units in the energy profiler output of each app? (3) How to actually diff the energy profiles of similar apps?

3.1 What diffing granularity?

A mobile app typically implements many features. We refer to the implementation of individual app features in the source code and their invocations at runtime as *app tasks*. Similar apps are expected to implement a common set of core tasks pertaining to the apps' common, main functionality, *e.g.,* music playback along with some basic UI features (*e.g.,* progress bar) for music streaming apps.

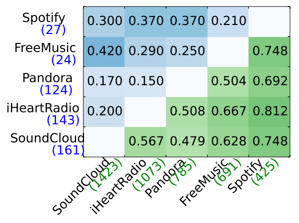
In addition, similar apps by different vendors often support some differentiating features which result in different tasks at runtime. For example, among the five streaming apps, SoundCloud uniquely depicts the audio track using a waveform animation during music playback.

Since there are two potential factors that contribute to the different energy drain of similar apps: (1) different implementation of common app tasks, and (2) app tasks unique to each of the similar apps, the natural granularity for energy diffing of similar apps should be an app task.

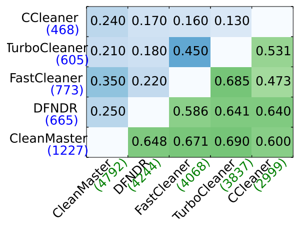
3.2 How do app tasks manifest in call trees?

Diffing at the task granularity, however, faces a fundamental challenge: *app tasks are not explicitly labeled by developers*. To overcome the above challenge, we examine how app tasks manifest in the call trees of Android apps.

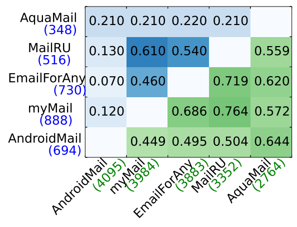
Android app programming is event-driven where the Android framework implements frequently used tasks as services. These Android framework services provide sev-



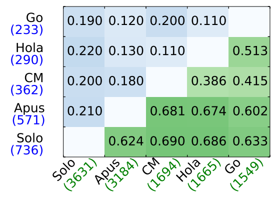
(a) Music streaming. Streaming music for 30 seconds.



(b) Cleaning. Press clean and wait for operation to end.



(c) Email. Compose an email and send.



(d) Launcher. Slide from left to right every 2sec for 30 seconds.

Figure 2: Pairwise overlap of similar apps. Lower triangle boxes show the percentage of overlapping framework method calls (0.30 means 30%). Upper triangle boxes show the matched app tasks in percentage of all tasks.

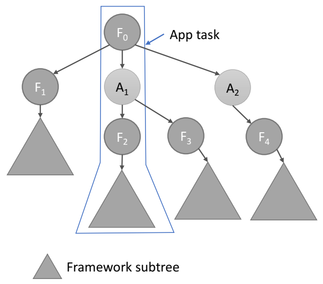


Figure 3: A typical call tree.

eral Java interfaces and classes with callback methods that apps can override. Apps then use the associated registration-callback mechanism to register the overridden callback app methods with the framework. Upon an event, the Android framework calls these overridden methods registered for the event.

We refer to Android framework methods as *F-methods* and app methods as *A-methods*. The above asynchronous programming suggests (1) an app's energy profiling output typically consists of many call trees [9], one for each thread; (2) as shown in Figure 3, each call tree typically starts with some framework method (F_0) that receives call-back related messages and makes a callback into the app (A_1). The app callback method (A_1) may call various other app methods (folded in A_1) which later call another framework method (F_2 or F_3) to register more callbacks (A_2) or for general processing that implements the task.

Using a script, we examined the call tree output by EPROF for all the apps in Figure 1 and confirmed that their call trees all follow the above structure, with one minor variation: a path may contain only F-methods (*e.g.,* (F_0, F_1)). This happens when an app task calls some framework method X that in turn registers an asynchronous callback of some other framework method Y. When framework method Y is invoked, it starts a new path off the root of the call tree consisting entirely of framework methods. Typical general-purpose framework methods that serve as the roots of the call trees include `Handler.dispatchMessage` and `Binder.execTransact`.

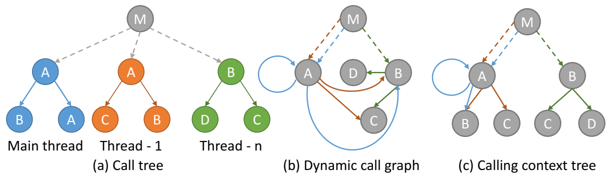


Figure 4: Call trees, dynamic call graphs, and calling context trees.

What constitutes a task in the call tree? The above call tree structure suggests an app task typically manifests in a call tree in an Erlenmeyer flask-shaped [19] slice with three components [2], as shown in Figure 3:

- *Call path:* The call path from the root of the call tree consisting of some F-methods followed by some A-methods that lead to the F-method uniquely captures the context of the task, *i.e.,* under which the F-method was called;
- *F-method:* The specific F-method invoked by the app method that is the entry to the invoked framework service that accomplishes the app task;
- *Subtree:* The actual execution of the F-method, given the context and the parameters passed to the entry method.

We denote the three-component structure as an EFLASK, which is a (path, F-method, subtree) task tuple.

In practice, it is often not obvious to isolate all the EFLASKS in a given call tree that correspond to app tasks, due to the possibly many layers of interleaving of A-methods and F-methods. Our EFLASK matching algorithm described in §3.4 takes the call trees of two similar apps and simultaneously identifies EFLASKS corresponding to app tasks and finds matching tasks.

3.3 What tree structures to diff?

Before discussing the diffing algorithm, we first explore different options of tree structures to perform diffing, as shown in Figure 4.

Call tree Since EPROF outputs a call tree for each execution profile, the baseline approach would be to directly diff the two call trees (CT). However, this is not practical, since an app task may be invoked many times during a profiling run and thus its task tuple may appear many times in the call tree output. Further, the call tree becomes hopelessly large, up to several million call tree nodes in just a few minutes of a typical profiling run.

Dynamic call graph An alternative approach is to convert call trees to dynamic call graphs (DCG) [9] and diff DCGs instead, where every method executed has just one corresponding method node in a DCG. However, using DCG faces a fundamental challenge that a DCG is not path preserving, *i.e.,* it may contain code paths that never occurred during the profile run. For example, the DCG in Figure 4(b) contains path $M \rightarrow A \rightarrow B \rightarrow D$ which never occurred in the CT in Figure 4(a). Paths need to be preserved for matching the EFLASKS of the same app task.

Calling context tree DIFFPROF overcomes the above shortcomings of CT and DCG by building and using calling context trees (CCT) [9], a middle ground between call trees and dynamic call graphs. In a nutshell, two method call nodes in the call tree are merged in the CCT whenever both nodes have an identical path from the root. In addition, recursive calls are merged to the non-recursive ancestor to keep the tree bounded in size, for example node A in Figure 4(c)[3]. Thus, using a CCT preserves the valuable path information while significantly reducing the number of nodes in the tree. In practice, we found CCT to contain only tens of thousands of nodes in a few minutes of profiling run, allowing our sophisticated matching algorithm to run in less than 30 seconds (§5).

3.4 How to perform EFLASK matching?

We first discuss the need for approximate matching to find EFLASKS corresponding to the same app tasks. We then review prior tree matching algorithms, discuss their drawbacks when applied to our problem, followed by our EFLASK matching algorithm.

3.4.1 Need for approximate DIFFPROF matching

The above understanding of how app tasks manifest in call trees in §3.2 suggests that different implementations and hence their EFLASK structures of the same task in two apps can differ in the following ways:

- *The corresponding call paths may differ slightly.* This can happen for two main reasons. First, apps may use slightly different mechanisms to achieve the same app callback. For example, an app can start its `Runnable.run` method directly from a new thread, or via `ExecutorService`; the two lead to different paths from root. Second, the app can use different app callbacks for receiving similar events. For example, the Turbocleaner app handles the "clean" button press using `.onClick` callback while the DFNDR app uses `.onItemClick` callback after which both apps call `Activity.startActivity` to perform a common task.
- *The entry F-methods may differ* due to two main reasons. First, the same task API can be provided by many different framework classes. For example, both `Https URLConnectionImpl.getInputStream` and `HttpURLConnectionImpl.getInputStream` get data from a server, one from an https and another from an http connection. Second, the same framework class may provide many alternate APIs to perform the same app task. For example, three different apps, Wish, Kohl and letgo, share 8 common nodes in the call path from the root call and finally call three different APIs, `ImageView.setImageDrawable`, `ImageView.setImageBitmap` and `ImageView.setImageResource`, respectively, for setting an image.
- *The subtrees that reflect the actual executions of the app task in similar apps can differ.* Even when the developers use the same framework API call to accomplish a task, the program state and the call parameters passed in can differ which lead the framework service to take different paths resulting in different subtrees.

3.4.2 Prior tree matching algorithms

How to match two trees to find similar components has been previously studied with a diverse set of applications such as matching RNA structures, structured text databases and image analysis [12]. However, prior matching algorithms are not suitable for matching EFLASKS.

Exact path matching Let T_1 and T_2 be two CCTs rooted at r_1 and r_2, with the set of nodes denoted by $V(T_1)$ and $V(T_2)$. Formally, exact path matching produces a maximal one-to-one node matching[4] $M \subseteq V(T_1) \times V(T_2)$, where for any pair $(v,w) \in \{M - (r_1,r_2)\}$:

$$(r_1,r_2) \in M \text{ and } (P(v),P(w)) \in M \qquad \text{(Path Condition)} \qquad (1)$$

where $P(v)$ and $P(w)$ are parents of nodes v and w respectively. However, exact path matching cannot match paths (*e.g.,* of EFLASK) with minor variations.

Prior approximate tree matching algorithms Tai *et al.* [38] gave the first approximate tree matching algorithm. This algorithm produces a maximal one-to-one matching M where for any pair $(v_1,w_1),(v_2,w_2) \in M$:

$$v_1 \text{ is ancestor of } v_2 \text{ iff } w_1 \text{ is ancestor of } w_2 \qquad \text{(Ancestor Condition)} \qquad (2)$$

The output matching replaces the Path Condition in Eqn. 1 with a significantly weaker *Ancestor Condition* (*i.e.,* Path Condition implies Ancestor Condition). However, the algorithm is Max-SNP hard.

To reduce the running time, Zhang *et al.* [44] added a *Structure Respecting Condition* to output matching. This algorithm produce a matching M, such that for any pairs (v_1, w_1), (v_2, w_2), $(v_3, w_3) \in M$:

$$nca(v_1, v_2) = nca(v_1, v_3) \text{ iff } nca(w_1, w_2) = nca(w_1, w_3)$$
$$\text{(Structure Respecting Condition)} \qquad (3)$$

where $nca(x, y)$ is the nearest common ancestor of nodes x and y. Due to the additional constraint, fewer matching possibilities need to be considered, making the algorithm's running time polynomial.

However, these algorithms may match EFLASKS with very different call paths. In contrast to the exact path matching algorithm which focuses on matching the path (component of EFLASKS) without considering the subtrees underneath, the above approximate matching algorithms match two nodes only based on similarity of subtrees (another component of EFLASKS) underneath them disregarding the call paths. The EFLASK matching algorithm we propose below leverages both the path and subtree information in matching two nodes, and in doing so, matches two EFLASKS.

3.4.3 The EFLASK matching algorithm

The EFLASK matching algorithm relaxes the Path Condition incrementally, *i.e.,* the paths from root to matched nodes in two trees can differ by *at most α nodes*, and maximizes the subtree overlap. We replace the Path Condition in the exact matching algorithm with a *Relaxed Path Condition* while retaining the Structure Respecting Condition (Eqn. 3) and Ancestor Condition (Eqn. 2) to find such matching. Formally, we wish to produce a maximal one-to-one matching M, that satisfies Eqn. 2 and Eqn. 3 and for any pair $(v, w) \in M$:

$$w \in C_\alpha(v)$$
$$\text{(Relaxed Path Condition)} \qquad (4)$$

where $C_\alpha(v) \subseteq V(T_2)$ is the *candidate set*, where the path from T_2's root to each node in $C_\alpha(v)$ differs from the path from T_1's root to v by less than or equal to α nodes. For example, Figure 5 highlights the nodes in the candidate set $C_\alpha(b)$ for α equal to 0, 1 and 2. $C_0(b)$ contains just 1 node that has the same path from its root as the b in T_1. $C_1(b)$ includes 3 additional nodes a, b and c whose path from root becomes identical to b's from T_1's root, $r \to a \to b$ by doing exactly one operation – deleting b, deleting a and replacing b by c, respectively.

Notations Before presenting the algorithm we define a few notations. Let T_1, T_2 denote two unordered labeled tree with maximum degrees D_1 and D_2, respectively. We denote the set of children nodes of node v by $child(v)$ and its label by $label(v)$. The path from the root to node v thus forms a string of labels and is represented by $s(v)$.

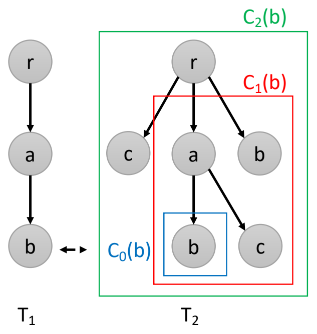


Figure 5: Candidate set $C_\alpha(b)$ for α = 0, 1 and 2.

Let θ denotes an empty tree and let $T(v)$ denote the subtree of T rooted at a node $v \in V(T)$ and $F(v)$ denote the forest under node v, $F(v) = T(v) - \{v\}$.

While matching the nodes in two trees, we can perform three types of edit operations to the tree nodes – (1) a relabeling operation to change the node label, (2) a deletion operation to delete node v and make all the children of v the children of $P(v)$, and (3) an insertion operation, the complement of deletion.

Let λ denote a special blank symbol. The cost of each edit operation can be specified using a cost function, γ. Thus, $\gamma(l_1, l_2)$ is the cost of replacing l_1 by l_2, $\gamma(l_1, \lambda)$ is the cost of deleting l_1 and $\gamma(\lambda, l_1)$ is the cost of inserting l_1. γ is generally assumed to be a distance metric, *i.e.,* γ is non-negative, symmetric and follows triangular inequality. We extend the notation such that $\gamma(v, w)$ for nodes v and w denotes $\gamma(label(v), label(w))$. We assume unit cost distance in the design of algorithm, *i.e.,* $\gamma(l_1, l_2) = 1$ when $l_1 \neq l_2$.

Now we are ready to define a few functions and their properties which form the basis of our algorithm.

Path edit distance function We first find $C_\alpha(v)$ by computing a *path edit distance function* ρ. For some $v \in V(T_1)$ and $w \in V(T_2)$, $\rho(s(v), s(w))$ is the total cost of edit operations required for v and w to have identical paths from the root. Thus $C_\alpha(v) = \{w \in V(T_2) | \rho(s(v), s(w)) \leq \alpha\}$.

Since paths $s(v)$ and $s(w)$ are strings, path edit distance function $\rho(s(v), s(w))$ is thus equal to the string edit distance [41] between $s(v)$ and $s(w)$ and hence can be calculated in a similar manner.

Since we only care about path edit distance when it is less than or equal to α, we prune some computation as soon as the distance exceeds α. We can show the runtime for computing C_α is $O(min(N_1 D_2^{\alpha+2}, N_1 N_2))$.

Subtree match function Next, we define a *subtree match function* μ_α between two trees. For $v \in V(T_1)$ and $w \in V(T_2), \mu_\alpha(T_1(v), T_2(w))$ is the size of maximal matching of subtrees $T_1(v)$ and $T_2(w)$ where the matching nodes' paths differ by at most α.

Before providing the next lemma, we need the following definition. A restricted matching $RM(v,w)$ is a matching between nodes of $F_1(v)$ and $F_2(w)$ and is defined as follows: (1) $RM(v,w)$ follows all the matching conditions – Relaxed Path Condition (Eqn. 4), Structure Respecting Condition (Eqn. 3), Ancestor Condition (Eqn. 2), and (2) if (p,q) is in $RM(v,w)$, p is in $T_1(v_i)$ and q is in $T_2(w_j)$, then for any (p',q') in $RM(v,w)$, p' is in $T_1(v_i)$ iff q' is in $T_2(w_j)$ where $v_i \in child(v)$ and $w_j \in child(w)$. In other words, node from a subtree $T_1(v_i)$ must only map to nodes of one subtree $T_2(w_j)$ and vice versa.

Motivated by the constrained edit distance algorithm [44], we derive the recurrence relationship for μ_α.

Lemma 3.1. *For all $v \in V(T_1)$ and $w \in V(T_2)$,*

$$\mu_\alpha(T_1(v),\theta) = 0$$
$$\mu_\alpha(\theta,T_2(w)) = 0$$
$$\mu_\alpha(T_1(v),T_2(w)) = 0 \qquad \text{if } w \notin C_\alpha(v)$$
$$\mu_\alpha(T_1(v),T_2(w)) = max\begin{pmatrix} \max\limits_{w_j \in child(w)} \mu_\alpha(T_1(v),T_2(w_j)) \\ \max\limits_{v_i \in child(v)} \mu_\alpha(T_1(v_i),T_2(w)) \\ \max\limits_{RM(v,w)} \mu_\alpha(RM(v,w)) \\ \qquad +(1-\gamma(v,w)) \end{pmatrix}; \quad otherwise$$

Proof. Proof is similar to [44], we skip the details here. □

Again, for any $v \in V(T_1)$, we need to compute the $\mu_\alpha(T_1(v),T_2(w))$ function described above for all $w \in C_\alpha(v)$. The runtime for computing μ_α is $O(N_1 \cdot min(D_2^{\alpha+1},N_2) \cdot (D_1+D_2) \cdot log(D_1+D_2))^2$.

The EFLASK **matching algorithm** Putting things together, the flexible tree matching algorithm makes two passes. First, it makes a *top-down pass* to compute $C_\alpha(v)$ for all $v \in V(T_1)$, *i.e.,* find nodes with call paths different by at most α nodes. Next, it makes a *bottom-up pass* to compute $\mu_\alpha(T_1,T_2)$. Third, it uses a simple backtracking mechanism to find for each node $v \in T_1$ the matching node $w \in T_2$ that maximizes the T_1-T_2 tree match. Finally, it finds the matching EFLASKS based on these maximally matched nodes.

The two passes together simultaneously accomplish matching of both the call path and the subtree components of similar EFLASKS.

3.5 Preprocessing CCTs to facilitate effective matching

The α value affects the tradeoff between finding more matching tasks (that vary in their call paths) and false positive matches. To make the algorithm more effective, we identified several factors that may increase the path distance between the paths for the same app task, and preprocess the CCTs to remove such factors so that more matchings can be found with smaller α values.

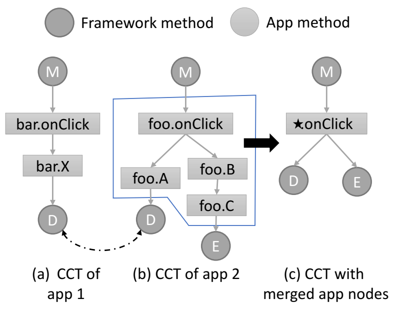


Figure 6: App namespace problem.

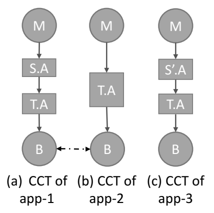


Figure 7: Class hierarchy problem.

App namespace problem The call paths for the same task in two CCTs can contain many app methods that are unique to either app as different developers are likely to structure and name the app methods differently. Such app-specific app methods can easily blow up the path edit distance of the call paths of a matching task. Figure 6(a,b) show an example of two paths with differing app methods.

We observe that all the callback app methods must override some predefined framework methods, and the remaining internal app methods called from other app methods have arbitrary names and are also often obfuscated. We thus merge all the internal app method calls into the app callback method root node as shown in Figure 6(c), and drop the app specific class names from app's callback node to allow matching callback methods.

We note that like using DCGs, merging app methods to address the app namespace problem conceptually also reduces path sensitivity, but it actually improves the effectiveness of task matching. This is because the internal methods of different apps tend to be named very differently and thus path sensitivity to app method names actually harms path similarity matching.

Class hierarchy problem A similar issue arises due to the object-oriented nature of Java, as shown in the following example. The two apps in Figure 7(a,b) share a same task pointed by the dashed arrow, but the first app uses method `S.A` which extends and calls method `T.A` and the second app directly uses `T.A`. Each such occurrence in the path increases the path edit distance by one, and more occurrences will quickly inflate the path edit distance.

We solve this problem in two steps. First, we merge `T.A` into the caller node `S.A` (`S'.A`). Second, we tweak the distance function γ to allow matching `S.A` with `T.A`, *i.e.,* γ(`S.A`,`T.A`) = 0. This allows matching the common task in Figure 7(a,b) with a path distance of zero while retaining the same path edit distance for sibling classes in Figure 7(a,c).

F-method only paths A third situation happens when a path off the root consists entirely of framework methods as discussed in §3.2 (path (F_0, F_1) in Figure 3). When this happens, in energy profiling, the energy consumption of the call path is not propagated to its asynchronous caller, *i.e.,* the app task, which leaves the developer clueless as to what app task caused the energy drain.

DIFFPROF patches such asynchronous framework only subtrees to its parent app task by adding additional logging in the Android framework. In particular, it logs the callback object's `.hashCode()` along with the current timestamp and thread id, when an asynchronous callback is enqueued in framework and when the callback is later dispatched. During post-processing, for each dispatch method call, the nearest preceding enqueue method call with matching object `.hashCode` log is patched as the dispatch method call's asynchronous caller. [5]

4 Implementation and Usage

We implemented DIFFPROF on top of EPROF [32] with 5.7K lines of Java code. DIFFPROF is packaged as an IDE plugin that can be installed on a laptop, with a GUI front-end, for interacting with the developer and computing and showing the energy diffing result. EPROF traces are collected on a phone running a modified Android 6.0.1 framework version that adds 95 lines to capture hidden causal relationships due to asynchronous programming (§3.5). [6]

After collecting EPROF traces of two similar apps, the developer specifies these traces to DIFFPROF, and DIFFPROF performs energy diffing in the following steps. (1) First, DIFFPROF patches the call tree dumped by EPROF using the call timing and the log timestamp as described in §3.5. (2) Next, DIFFPROF converts EPROF's CT output into CCT and dumps the CCT along with the inclusive and exclusive energy consumption by and the number of recursive and non-recursive invocations of each CCT node. (3) Next, the developer is presented with a list of Java package names that appeared in either app trace to determine app packages used for merging app methods as described in §3.5. By default, all packages not belonging to the Android framework are marked as app packages. For comparing two different apps, developers can skip this step, since packages not belonging to the Android framework are already marked as app packages. When comparing two versions of the same app, however, this presents an opportunity for the developer to unmark certain app packages to expose app-internal path information (Figure 6) during matching. (4) DIFFPROF performs the EFLASK matching algorithm on the pair of CCTs. (5) Finally, since the EFLASKS of multiple tasks may share a common path, DIFFPROF assigns the energy drain for each task as the inclusive energy of the F-method.

Table 2: Average running time and matched tasks when adjusting α. The results are averaged over all app pairs in each group. α=0 gives the exact matching algorithm.

α	0	1	2	3	4	5
Avg. time (sec)	0.20	1.12	4.89	7.79	16.4	25.7
Avg. % of matched tasks	10.8	15.6	18.0	19.5	21.3	22.9

DIFFPROF gives two outputs: (1) a merged list of matched (with the other app) and unmatched tasks in the app, sorted by the energy drain for unmatched tasks and the energy difference for matched ones, *i.e.,* based on the potential room for improvement; and (2) upon selection, a task's EFLASK in a graphical view.

5 Evaluation

Our evaluation answers the following questions: (1) Does DIFFPROF effectively identify matching and unique tasks among similar apps? (2) Does DIFFPROF offer added benefits over EPROF, in particular, how does it help developers with understanding and coming up with more efficient implementation?

5.1 Experimental setup

We use DIFFPROF to profile popular apps belonging to the 8 app groups in Table 1. For each group, we pick 5 different apps and 4 older versions of one of the 5 apps, same as in Figure 1. In running the tests, we ensure user interaction homogeneity using automated testing as described in §2.2. All app tests are less than 1 minute long and are run on a Nexus 6 phone running DIFFPROF's modified version of Android 6.0.1. The traces are post processed and task matching is performed on a Macbook pro laptop with a 2.5 GHz Intel i5 CPU and 8GB 1600 MHz DDR3 main memory.

Impact of α We first evaluate the impact of changing α on the EFLASK matching algorithm's running time and output. Table 2 summarizes the results. We see that as expected, the running time grows close to exponentially with the α value (from 0 to 2 and from 2 to 4). On average, the algorithm produces the energy diffing output within half a minute for all values of $\alpha \leq 5$.

Next, we observe that the average percentage of matching tasks grows steadily as we increase the value of α, starting 10.8% on average at α=0 up to 22.9% at α=5. The growth slows down at α = 5.

Based on the above result, when profiling the 8 app groups, for each app pair in a category, we run DIFFPROF to find the matching tasks using the lowest α that can match 20% of the tasks, up to $\alpha = 5$ (shown as dynamic α in Table 3).

5.2 Diffing results

The pairwise task overlap for 4 app groups (Music streaming, Cleaning, Email, Launcher) are shown in the upper triangles in Figure 2. We see that the task overlap between similar apps is significant, ranging between 7%–61%, with an average of 27%, 24%, 28%, and 17%, for the 4 groups, respectively.

Table 3 gives the details of diffing results for each app in the 8 app groups. For each app, we classify all its tasks into tasks that could not be matched with any of the 4 other apps in its category and tasks that were matched with 4, 3, 2 or 1 other app(s). The columns under "Dynamic α" show that the count of such tasks for each app varies for different categories, *e.g.,* Email apps have 17 5-way matching tasks while Music apps have only 2, suggesting the apps in different categories have different levels of overlapping tasks. We manually examined 20% of the matched tasks and did not find any false positives.

Table 3 also shows that the percentage of energy drained by matched tasks (*i.e.,* 1 minus that of unique tasks energy) is over 70% of the total energy drained by the app for 32 out of the 40 apps. This suggests that although it is hard to measure the coverage (false negative) of task matching produced by DIFFPROF, in practice, DIFFPROF produces matched tasks that already account for a majority of the app energy drain which gives app developers enough focus for optimization.

DIFFPROF also exposes app unique tasks that drain significant amounts of energy. Table 3 shows SoundCloud and CM launcher drain 53.7% and 43.7% of the total energy in performing unique app tasks/features, waveform animation and rotation animation, respectively.

To show the effectiveness of the EFLASK algorithm, Table 3 last column lists the number of tasks in each app that do not get matched using the exact path matching algorithm ($\alpha = 0$). We see that the EFLASK matching algorithm with dynamic α reduces the number of unmatched tasks by 13.5% on average (shown in second column).

5.3 Effectiveness

We discuss how DIFFPROF offers added benefits over a standard energy profiler through extensive case studies. Our case studies show that DIFFPROF provides developers with actionable diagnosis beyond a standard energy profiler in two ways: (1) DIFFPROF identifies non-essential (unmatched or extra) and known-to-be inefficient (matched) tasks; (2) the EFLASKS of tasks extracted by DIFFPROF further expose the reasons for the more efficient implementation. For convenience, in the following, we often refer to a task by the F-method in its EFLASK 3-tuple.

Methodology We ran DIFFPROF on the top 3 energy-draining apps in each of the 8 groups against the least

Table 3: Task overlap for all apps.

	Dynamic α						α=0
App	**0**	**1**	**2**	**3**	**4**	**Unique tasks' energy**	**0**
Antivirus							
AVG	424	191	45	5	5	7.27%	498
CMSecurity	433	169	36	6	5	20.36%	532
DU	252	68	27	8	5	14.10%	286
Kaspersky	126	48	28	7	5	26.01%	149
MobileSec	165	52	39	9	5	30.02%	227
Cleaner							
CCleaner	301	100	41	18	8	23.43%	395
CM	797	290	92	44	8	29.18%	863
DFNDR	402	138	77	46	8	26.48%	495
Fast	265	356	92	55	8	5.11%	286
Turbo	250	234	69	46	8	11.54%	259
Email							
Android Mail	581	67	25	11	17	17.26%	656
Aqua Mail	223	59	28	21	17	6.67%	308
Email For Any	331	154	193	40	17	3.79%	338
Mail RU	131	129	199	47	17	0.60%	145
myMail	434	200	202	40	17	3.10%	454
File Explorer							
ES	244	43	14	4	5	25.15%	272
FX	83	33	5	2	5	4.97%	97
File Exp.	110	42	13	1	5	10.92%	130
File Man.	332	51	9	4	5	24.76%	366
Solid	260	47	16	2	5	7.31%	295
Instant Messaging and Calling							
Hangout	780	160	44	7	8	36.50%	881
Line	291	88	35	21	8	29.14%	411
Messenger	928	256	59	13	8	28.00%	1167
TextNow	1405	194	40	4	8	38.47%	1542
Whatsapp	274	107	26	15	8	40.51%	391
Launcher							
Apus	430	111	23	6	8	32.77%	495
CM	252	65	30	11	8	43.65%	318
Go	161	50	11	8	8	26.21%	204
Hola	212	45	21	4	8	29.74%	252
Solo	560	132	31	7	8	26.23%	640
Music							
FreeMusic	11	6	6	1	2	1.38%	12
Pandora	97	18	5	3	2	17.34%	107
SoundCloud	123	24	10	3	2	53.72%	135
Spotify	14	5	3	3	2	6.23%	14
iHeartRadio	98	34	8	3	2	8.72%	104
Shopping							
Amazon	1030	135	54	17	10	32.03%	1118
Kohl	900	218	80	24	10	27.84%	1041
Wish	1321	264	94	30	10	23.15%	1473
eBay	715	172	86	32	10	26.69%	840
letgo	618	222	108	31	10	19.55%	729
Average	409	119	51	16	8	21.14%	473

Table 4: Buggy and inefficient tasks in case studies and their energy drain.

App	Task	Task energy drain (μAh)	% of total energy drain
Unmatched tasks			
Hangout	ContentResolver.query	44.3	10.1%
Kohl	ObjectInputStream.readObject	12.8	3.9%
Kohl	ObjectOutputStream.writeObject	10.5	3.2%
Kaspersky	Thread.getStackTrace	39.6	14.8%
Pandora 8.0	SharedPreferencesImpl $EditorImpl.apply	22.9	17.5%
DFNDR	Runtime.exec	19.5	5.2%
Matched tasks			
Wish letgo	Bitmap.compress	100.9 7.14	15.9% 3.6%
Wish letgo	BitmapFactory.decodeStream	126.3 5.01	19.9% 2.5%
Pandora5.7 Pandora8.3	TextView.setText	43.6 0.74	28.1% 0.7%
Spotify Pandora	ProgressBar.setProgress	29.2 1.74	20.2% 1.6%
TextNow Whatsapp	ViewRootImpl.performTraversal	230.5 24.0	40.6% 28.4%
Solid FX	Drawable.invalidateSelf	35.5 1.24	18.9% 2.0%

Table 5: Rank ordered EPROF's method energy output and DIFFPROF's task energy difference output for Google Hangout compared to Whatsapp. Energy in μAh. "*": unmatched tasks.

Rank	Method name (EPROF output)	Inclusive energy
1	(toplevel)	436.8
2	void Looper.loop()	220.6
3	void Handler.dispatchMessage(Message)	207.3
4	void Thread.run()	176.9
5	Object Method.invoke()	175.4
27	Cursor ContentResolver.query()	44.3
Rank	**Method name (EPROF output)**	**Exclusive energy**
1	boolean BinderProxy.transactNative()	50.8
2	void VMRuntime.runHeapTasks()	11.6
3	void MessageQueue.nativePollOnce()	9.86
4	Object Throwable.nativeFillInStackTrace()	9.39
5	void Trace.nativeTraceBegin()	7.81
1336	Cursor ContentResolver.query()	0.00
Rank	**Task name (DIFFPROF output)**	**Task energy**
1	Cursor ContentResolver.query()*	44.3
2	int TelephonyManager.getSimState()*	24.9
3	Cursor SQLiteQueryBuilder.query()*	17.2
4	void ObjectOutputStream.writeObject()	11.2
5	Spanned Html.fromHtml()	6.49

energy-draining app in the same group, and looked at the top energy-draining app tasks output by DIFFPROF. Out of these, we skip the cases where the app tasks are for supporting unique app features (*e.g.,* 47.2% of SoundCloud's total energy was by a task supporting the waveform animation feature). The remaining 12 tasks, summarized in Table 4, all belong to buggy or inefficient implementations, removing which reduces the app energy drain by 5.2%–27.4% (based on the energy difference).

5.3.1 Unmatched (extra) tasks

Instant Messaging Table 5 shows Google Hangout's energy output from EPROF and from DIFFPROF when compared with Whatsapp. When sorted by inclusive energy, EPROF shows really high-level Android methods such as `Looper.loop` on the top, and when sorted by exclusive energy, it shows really low-level Android methods such as `BinderProxy.transactNative` on the top. Such top energy drainers in both inclusive and exclusive energy lists are F-methods that do not directly call app methods and are not directly called by the app; the developers thus do not get useful guidance on what to focus on from the long list of EPROF output.

In contrast, DIFFPROF outputs tasks sorted by energy drain. It shows Hangout consumes more than 10% of its total energy in an unmatched task `ContentResolver.query`. Since tasks' F-methods are directly called by the app, the top task's name provides direct hints to developer on how to optimize the app. EPROF, however, does not highlight such methods; the top task method appeared at position 27 when sorted by inclusive energy and at 1336 when sorted by exclusive energy.

Finding the reasons and optimization for task `ContentResolver.query` would have been easy for its developer from the EFLASK output, *e.g.,* the `ContentResolver.query` method was called 116 times. But since we did not write the app, to understand this energy drain, we set a breakpoint at the `ContentResolver.query` method and reran the app to examine the parameters passed to the method. In one call to the method, the app queries multiple fields that are stored in a local database. We found that at one message send, the app queries for 81 unique database fields which often are repeated across two different queries. Moreover, 36 out of the 81 fields, such as `author_chat_id` and `author_first_name`, do not change across two send key presses, but keep on getting queried at each send. This suggests that there is ample room for optimization by keeping a staleness flag; only when the user navigates away from a chat window, the 36 fields can be declared stale and re-queried later.

Shopping Table 6 shows the Kohl's app's output from EPROF and from DIFFPROF when compared with letgo. DIFFPROF shows `ObjectInputStream.readObject` and `ObjectOutputStream.writeObject` are two top energy draining extra tasks, consuming 3.9% and 3.2% respectively of its total energy consumption. In contrast, EPROF outputs them at positions 90 and 133 when sorted by inclusive energy and at 1516 and 1547 when sorted by exclusive energy, respectively.

Table 6: Rank ordered EPROF's method energy output and DIFFPROF's task energy difference output for Kohl compared to letgo. Energy in μAh. "*": unmatched tasks.

Rank	Method name (EPROF output)	Inclusive energy
1	(toplevel)	329.71
2	Object Method.invoke()	149.74
3	void Looper.loop()	134.65
4	void ActivityThread.main()	132.89
5	void ZygoteInit$MethodAndArgsCaller.run()	132.89
90	Object ObjectInputStream.readObject()	12.82
133	void ObjectOutputStream.writeObject()	10.53
Rank	**Method name (EPROF output)**	**Exclusive energy**
1	void VMRuntime.runHeapTasks()	26.11
2	boolean BinderProxy.transactNative()	11.9
3	Bitmap BitmapFactory.nativeDecodeByteArray()	10.34
4	void DdmVmInternal.threadNotify()	10.02
5	String StringFactory.newStringFromChars()	7.96
1516	Object ObjectInputStream.readObject( )	0.0
1547	void ObjectOutputStream.writeObject()	0.0
Rank	**Task name (DIFFPROF output)**	**Task energy**
1	Object ObjectInputStream.readObject()*	12.82
2	Bitmap BitmapFactory.decodeByteArray()	11.19
3	void ObjectOutputStream.writeObject()*	10.53
4	boolean Class.isAnonymousClass()	9.55
5	String JSONObject.toString()	8.24

Since we did not write the app, we dug into the energy drain by setting breakpoints. We found that the app keeps the entire catalog and current discount campaigns on the SD card in `catalog.tmp` and `cms.tmp` files respectively which were 227 KB and 21 KB at the time of the experiment. Whenever a new catalog or a new campaign is synced with the server, the entire files are dumped again, rewriting the previous entries; using a database just to update new entries would have been more efficient.

Note that task `View.draw` consumes 12.16 μAh energy, more than the above extra tasks, but does not appear in the top task list. This is because DIFFPROF prioritizes the tasks with the most room for optimization: since the letgo app consumes 8.29 μAh for the same task, the difference is less than 4 μAh.

Antivirus DIFFPROF highlights `Thread.getStackTrace` as an extra task in the Kaspersky app which consumes 39.57 μAh, 14.8% of the app's total energy drain (position 1 in DIFFPROF output, but 22 in EPROF output). After decompiling the app apk using dex2jar [4], we inspected the caller of `Thread.getStackTrace` in the app source code and found that the app collects logs with unicode characters but in every such attempt, the code throws `UnsupportedEncodingException` which internally collects the thread stack trace thus unnecessarily wasting energy. This bug was confirmed by Kaspersky developers.

Music DIFFPROF highlights `SharedPreferencesImpl$Editor.apply` as an extra task in Pandora v8.0 that consumes 17.5% of its total energy drain (position 4 in DIFFPROF output but 42 in EPROF output). This method is used to change app preferences. The Android developer manual suggests that apps should call `SharedPreferencesImpl$Editor.edit` repeatedly to keep making changes in memory and then call `SharedPreferencesImpl$Editor.apply` once at the end to commit all the changes to the disk. However, the app mistakenly calls `SharedPreferencesImpl$Editor.apply` once every second. This bug was confirmed and fixed in the latest version of the Pandora app.

Cleaner DIFFPROF shows that the DFNDR app calls framework method `Runtime.exec`, consuming 19.52 μAh, 5.2% of the app's total energy consumption (position 3 in DIFFPROF output but 50 in EPROF output). We set a breakpoint at this method and examined its parameters and found that the app runs `ps | grep <app_pkg>` for each app installed on the phone. Since `ps` walks down the entire `/proc` directory, it would be more efficient to just obtain the `ps` output once and parse it to find the fields related to each app.

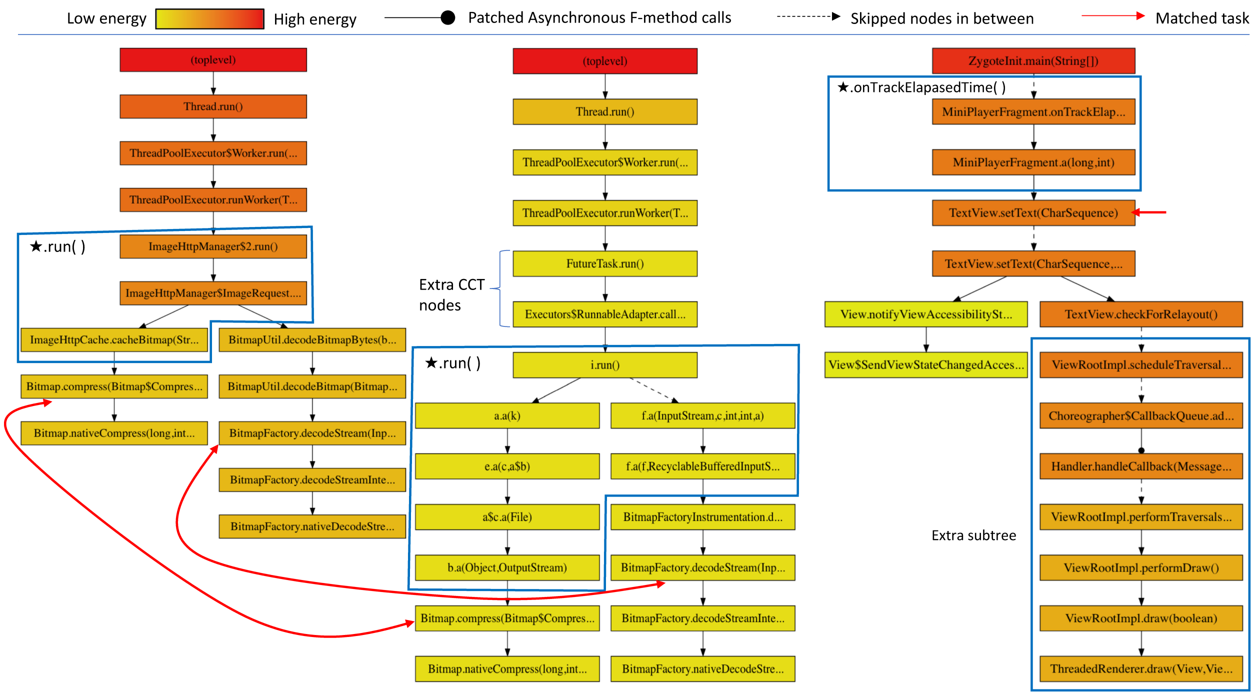
5.3.2 Matched tasks

Shopping In diffing Wish and letgo, although the CCTs of the two apps differ a lot structurally as shown in Figure 8(a), DIFFPROF is able to match two commons tasks, `Bitmap.compress` and `BitmapFactory.decodeStream`, by collapsing app methods to `*.run` and its flexible EFLASK matching algorithm.

For the `Bitmap.compress` task, DIFFPROF shows that Wish consumes 100.94 μAh, 15.9% of its total energy drain whereas letgo consumes only 7.14 μAh. To find the root cause of energy difference, we examined the parameters passed to the F-method by setting a breakpoint and rerunning both apps. We found that Wish compresses the image into a png image with quality set to 100 while letgo compresses into a jpg image with quality set to 90. This causes the large energy difference while the images shown by both apps are visually similar.

The above image format difference also explains the energy drain difference between the second common task `BitmapFactory.decodeStream` where Wish consumes 126.32 μAh, 19.9% of its total energy drain while letgo consumes only 5.01 μAh.

Music – Pandora In diffing two versions of Pandora, DIFFPROF matches the common task `TextView.setText` even though structurally their EFLASKS look different, as shown in Figure 8(b) (merged to save space). DIFFPROF shows that the common task consumes 43.63 μAh, 28.1% of its total energy consumption in Pandora v5.7 but only 0.74 μAh in the latest Pandora app, v8.3.


Low energy
High energy
Patched Asynchronous F-method calls
Skipped nodes in between
Matched task
(toplevel)
Thread.run()
ThreadPoolExecutor$Worker.run(...
ThreadPoolExecutor.runWorker(T...
★.run( )
ImageHttpManager$2.run()
ImageHttpManager$ImageRequest....
ImageHttpCache.cacheBitmap(Str...
BitmapUtil.decodeBitmapBytes(b...
Bitmap.compress(Bitmap$Compres...
BitmapUtil.decodeBitmap(Bitmap...
Bitmap.nativeCompress(long,int...
BitmapFactory.decodeStream(Inp...
BitmapFactory.decodeStreamInte...
BitmapFactory.nativeDecodeStre...
(toplevel)
Thread.run()
ThreadPoolExecutor$Worker.run(...
ThreadPoolExecutor.runWorker(T...
Extra CCT nodes
FutureTask.run()
Executors$RunnableAdapter.call...
★.run( )
i.run()
a.a(k)
f.a(InputStream,c,int,int,a)
e.a(c,a$b)
f.a(f,RecyclableBufferedInputS...
a$c.a(File)
BitmapFactoryInstrumentation.d...
b.a(Object,OutputStream)
BitmapFactory.decodeStream(Inp...
Bitmap.compress(Bitmap$Compres...
BitmapFactory.decodeStreamInte...
Bitmap.nativeCompress(long,int...
BitmapFactory.nativeDecodeStre...
ZygoteInit.main(String[])
★.onTrackElapsedTime( )
MiniPlayerFragment.onTrackElap...
MiniPlayerFragment.a(long,int)
TextView.setText(CharSequence)
TextView.setText(CharSequence,...
View.notifyViewAccessibilitySt...
TextView.checkForRelayout()
View$SendViewStateChangedAcces...
ViewRootImpl.scheduleTraversal...
Choreographer$CallbackQueue.ad...
Handler.handleCallback(Message...
Extra subtree
ViewRootImpl.performTraversals...
ViewRootImpl.performDraw()
ViewRootImpl.draw(boolean)
ThreadedRenderer.draw(View,Vie...

(a) Wish vs letgo. App methods get collapsed into ★.run to facilitate matching. Flexible EFLASK matching algorithm successfully matches tasks even in the presence of extra CCT nodes.

(b) Pandora 5.7 vs Pandora 8.3. Skipped Pandora 8.3 tree due to page limit. Pandora 5.7 consumes higher energy due to an extra subtree in the matched task's EFLASK.

Figure 8: Matched tasks between (a) Wish and letgo, (b) Pandora v5.7 and v8.3.

DIFFPROF further highlights the reason for the difference: in Pandora v5.7, the subtree additionally contains the `ViewRootImpl.scheduleTraversal` subtree that traverses and measures the entire view hierarchy. We used a premium account to disable ads and played the same radio station on both Pandora versions for two hours while leaving the phone on the playback screen. We found that Pandora v5.7 drained 9.2% battery per hour whereas Pandora v8.3 drained only 6.7% battery per hour. We reported this bug to Pandora engineers, who verified that Pandora v5.7's layout.xml file erroneously declared the width of elapsed time and remaining time text views to `wrap_content`. This flag signals Android's `ViewManager` that the text view must be just large enough to enclose its content. As a result, every second when the app updates the elapsed time and remaining time text views, Android `ViewManager` traverses the entire view hierarchy to recompute the size of the text boxes. The text boxes were set to a fixed size in later versions of Pandora.

Music – Spotify In diffing Pandora and Spotify apps, DIFFPROF shows that the common `ProgressBar.setProgress` task consumes 43.63 μAh, 28.1% of its total energy in Spotify, but just 1.74 μAh in Pandora. The EFLASK output further shows that Spotify calls this method from `App.doFrame` 596 times while Pandora calls it only 29 times from `App.onTrackElapsedTime` during the 30 second music playback, *i.e.,* while Pandora updates the progress bar once per second, Spotify updates it on every frame, which is unnecessarily frequent as many frame draws lead to no pixel change.

Instant Messaging In diffing TextNow and Whatsapp, DIFFPROF shows that TextNow consumes 230.46 μAh, 40.6% of its total energy drain, in calling a common task `ViewRootImpl.performTraversal`, almost 10 times that in Whatsapp. On inspecting the layout of the two apps with Android's `HierarchyViewer`, we found that TextNow contains 226 views compared to 76 in Whatsapp. Our closer inspection of view properties shows that 172 views in TextNow are in fact not even visible on the screen. The app statically loads all the possible UI interactions such as `pause_playing_voice_note_button` and `change_billing_details_button_icon`, keeping them all in the view hierarchy instead of dynamically loading views on demand as recommended by Android [3] and thus inflating the view hierarchy traversal energy. Moreover, the app contains several `LinearLayout` with just an `ImageView` and a `TextView` which are recommended to be compressed into one compound view [8] to reduce the size of the view hierarchy.

File Explorer DIFFPROF shows that Solid explorer consumes 35.52 μAh, 18.9% of its total energy in task `Drawable.invalidateSelf` whereas FX file explorer only consumes 1.24 μAh. DIFFPROF further shows that Solid calls `Drawable.invalidateSelf` 1002 more times than FX and that the EFLASK contains `ObjectAnimator.animateValue` followed by Solid's `CircularAnimatedDrawable$1.set`. Upon inspecting this class, we found that the app does the animation when a new folder is created. At each frame, it draws an arc and requests another frame. However, after the folder gets created, the app stops drawing the arc but keeps requesting new frames, unnecessarily wasting energy.

6 Discussions

DIFFPROF's effectiveness in finding energy optimizations stems from the large overlap of Android libraries used among competing Android apps and accurate source-level energy profiling. As such, its central idea of diffing source-code-level profiling of similar apps in principle can be extended to find optimization opportunities in other performance metrics of interests to developers, such as latency, scalability and memory efficiency.

One of the central principles of software engineering, DRY (Don't repeat yourself) [24], preaches the use of reusable code, by abstracting all common reusable code into standalone libraries. The principle improves modern software developers' productivity and has gained wide adoption in recent years; almost every major build tool today [1, 6, 2] allows developers to specify library dependencies which are downloaded from a central repository and packaged with their software. We envision that DIFFPROF's approach can be extended to effectively compare source-code level profiling measurements of software in broader domains beyond mobile such as games, web frontends and server backends.

7 Related work

Performance and energy profiling There is a large body of work on performance profiling of sequential programs [20, 15, 30] and concurrent programs [17, 39]. There are also several works on energy profiling for mobile apps [32, 31, 34, 18]. EPROF [32] performs source-code-level energy profiling and accounts the energy drained by each phone component to individual app method calls. ARO [34] performs cross-layer profiling for network usage to expose apps' inefficient interactions with lower layers. Wattson [31] estimates app energy consumption on the developer workstation by emulating different environments such as network conditions, CPU speed and display technologies. GfxDoctor [18] quantifies the energy drain spent in traversing the entire frame rendering stack due to each UI update. All such profilers stop at finding performance/energy hotspots. DIFFPROF builds on top of such traditional profilers and tackles the hard but critical question in the app energy optimization process: whether and how energy hotspots in app source code can be restructured to drain less energy.

Diffing programs and runtime behavior. (1) **Programs.** There has been a large body of research to find regressions introduced from code revisions [13, 36, 22, 23], and on data mining application source code to detect software bugs, *e.g.,* [40]. DIFFPROF allows app developers to catch and debug energy drain regressions by comparing source-code energy profiles after code revisions. (2) **Runtime behavior.** Execution indexing [43] aligns event logs of two executions of the same program under different input or perturbations and has been used in detecting and understanding security leaks [27], deadlocks [28] and failures [45, 21]. DIFFPROF aligns calling context trees of two executions that may be from apps written by different developers to find energy inefficiencies.

Diffing beyond programs. More generally, diffing is a pervasive technique that celebrates and exploits diversity and has been applied to many other scenarios in computer systems and networking. Diffing data has been applied to storage data for data compression (*e.g.,* [29]), to network traffic for traffic reduction (*e.g.,* [11, 10]), to data structures in memory images for detecting polymorphic malware [16], and to frames for reducing graphics energy for mobile devices [25].

Beyond data, many systems, *e.g.,* PeerPressure [42], ClearView [33], Shen et al. [37], Encore [46], and Diff-Prov [14], apply diffing to learn or detect deviations from the correct or reference behavior, via statistical analysis or data mining, for detecting and diagnosing misconfigurations, performance anomalies or faulty events in the network and distributed systems.

8 Conclusion

This paper presents differential energy profiling which tackles the hard but critical question in the app energy optimization process faced by app developers: whether and how energy hotspots in app source code can be restructured to drain less energy. By performing approximate matching of energy profiles of similar apps by a traditional energy profiler, energy diffing automatically uncovers more efficient implementations of common app tasks and app-unique tasks among similar apps. We show how our prototype DIFFPROF tool provides developers with actionable diagnosis beyond a traditional energy profiler: it effortlessly reveals 12 inefficient or buggy implementations in 9 apps, and it further allows (non)developers to quickly understand the reasons and develop fixes for the energy difference.

Acknowledgement We thank our shepherd Andreas Haeberlen and the anonymous reviewers for their helpful comments which helped to improve this paper. This work was supported in part by NSF grant CSR-1718854.

References

[1] Apache maven project. http://maven.apache.org.

[2] Create .net apps faster with NuGeT. https://www.nuget.org.

[3] Delayed loading of views. https://developer.android.com/training/improving-layouts/loading-ondemand.html#ViewStub.

[4] dex2jar. https://sourceforge.net/projects/dex2jar/.

[5] dexinfo. https://github.com/poliva/dexinfo.

[6] Npm package manager. https://www.npmjs.com.

[7] Optimizing battery life. https://developer.android.com/training/monitoring-device-state/index.html.

[8] Using compound drawables. https://developer.android.com/training/improving-layouts/optimizing-layout.html#Lint.

[9] AMMONS, G., BALL, T., AND LARUS, J. R. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices 32*, 5 (1997), 85–96.

[10] ANAND, A., GUPTA, A., AKELLA, A., SESHAN, S., AND SHENKER, S. Packet caches on routers: the implications of universal redundant traffic elimination. In *Proc. of ACM SIGCOMM* (2008), pp. 219–230.

[11] ANAND, A., SEKAR, V., AND AKELLA, A. Smartre: an architecture for coordinated network-wide redundancy elimination. In *Proc. of ACM SIGCOMM* (2009), pp. 87–98.

[12] BILLE, P. A survey on tree edit distance and related problems. *Theoretical computer science 337*, 1 (2005), 217–239.

[13] CALCAGNO, C., DISTEFANO, D., DUBREIL, J., GABI, D., HOOIMEIJER, P., LUCA, M., OHEARN, P., PAPAKONSTANTINOU, I., PURBRICK, J., AND RODRIGUEZ, D. Moving fast with software verification. In *NASA Formal Methods Symposium* (2015), Springer, pp. 3–11.

[14] CHEN, A., WU, Y., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proc. of ACM SIGCOMM* (2016), pp. 115–128.

[15] COPPA, E., DEMETRESCU, C., AND FINOCCHI, I. Input-sensitive profiling. *ACM SIGPLAN Notices 47*, 6 (2012), 89–98.

[16] COZZIE, A., STRATTON, F., XUE, H., AND KING, S. T. Digging for data structures. In *Proc. of USENIX OSDI* (2008), pp. 255–266.

[17] CURTSINGER, C., AND BERGER, E. D. Coz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 184–197.

[18] DING, N., AND HU, Y. C. Gfxdoctor: A holistic graphics energy profiler for mobile devices. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 359–373.

[19] Erlenmeyer flask. https://en.wikipedia.org/wiki/Erlenmeyer_flask.

[20] GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K. gprof: A call graph execution profiler. In *Proc. of ACM PLDI* (1982).

[21] GUO, L., ROYCHOUDHURY, A., AND WANG, T. Accurately choosing execution runs for software fault localization. In *International Conference on Compiler Construction* (2006), Springer, pp. 80–95.

[22] GUPTA, R., HARROLD, M. J., AND SOFFA, M. L. An approach to regression testing using slicing. In *Software Maintenance, 1992. Proceedings., Conference on* (1992), IEEE, pp. 299–308.

[23] HASSAN, A. E. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering* (2009), IEEE Computer Society, pp. 78–88.

[24] HUNT, A., AND THOMAS, D. *The Pragmatic Programmer: From Journeyman to Master*. Addison Wesley Longman, Inc., ISBN-10: 020161622X., 1999.

[25] HWANG, C., PUSHP, S., KOH, C., YOON, J., LIU, Y., CHOI, S., AND SONG, J. Raven: Perception-aware optimization of power consumption for mobile games. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (2017), ACM, pp. 422–434.

[26] Energy efficiency and the user experience. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/EnergyandNetworking.html.

[27] JOHNSON, N. M., CABALLERO, J., CHEN, K. Z., MCCAMANT, S., POOSANKAM, P., REYNAUD, D., AND SONG, D. Differential slicing: Identifying causal execution differences for security applications. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), IEEE, pp. 347–362.

[28] JOSHI, P., PARK, C.-S., SEN, K., AND NAIK, M. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proc. of ACM PLDI* (2009), pp. 110–120.

[29] KULKARNI, P., DOUGLIS, F., LAVOIE, J. D., AND TRACEY, J. M. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference, General Track* (2004), pp. 59–72.

[30] KÜSTNER, T., WEIDENDORFER, J., AND WEINZIERL, T. Argument controlled profiling. In *European Conference on Parallel Processing* (2009), Springer, pp. 177–184.

[31] MITTAL, R., KANSAL, A., AND CHANDRA, R. Empowering developers to estimate app energy consumption. In *Proc. of ACM MobiCom* (2012).

[32] PATHAK, A., HU, Y. C., AND ZHANG, M. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *Proc. of EuroSys* (2012).

[33] PERKINSA, J. H., KIMB, S., LARSENG, S., AMARASINGHEA, S., BACHRACHA, J., CARBINA, M., PACHECOD, C., SHERWOOD, F., SIDIROGLOUA, S., SULLIVANE, G., WONGZ, W.-F., ERNSTQ, Y. Z. M. D., AND RINARD, M. Automatically patching errors in deployed software. In *SOSP* (2009), pp. 87–102.

[34] QIAN, F., WANG, Z., GERBER, A., MAO, Z., SEN, S., AND SPATSCHECK, O. Profiling resource usage for mobile applications: a cross-layer approach. In *Proc. of Mobisys* (2011).

[35] RAVINDRANATH, L., PADHYE, J., AGARWAL, S., MAHAJAN, R., OBERMILLER, I., AND SHAYANDEH, S. Appinsight: mobile app performance monitoring in the wild. In *Proc. of USENIX OSDI* (2012), pp. 107–120.

[36] ROTHERMEL, G., AND HARROLD, M. J. Selecting regression tests for object-oriented software. In *ICSM* (1994), vol. 94, pp. 14–25.

[37] SHEN, K., STEWART, C., LI, C., AND LI, X. Reference-driven performance anomaly identification. In *ACM SIGMETRICS* (2009), pp. 85–96.

[38] TAI, K.-C. The tree-to-tree correction problem. *Journal of the ACM (JACM) 26*, 3 (1979), 422–433.

[39] TALLENT, N. R., AND MELLOR-CRUMMEY, J. M. Effective performance measurement and analysis of multithreaded applications. In *PPoPP* (2009), ACM, pp. 229–240.

[40] TAN, L., YUAN, D., KRISHNA, G., AND ZHOU, Y. *icomment: bugs or bad comments?*. In *Proc. of ACM SOSP* (2007), pp. 145–158.

[41] WAGNER, R. A., AND FISCHER, M. J. The string-to-string correction problem. *Journal of the ACM (JACM) 21*, 1 (1974), 168–173.

[42] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with peerpressure. In *Proc. of USENIX OSDI* (2004), pp. 245–258.

[43] XIN, B., SUMNER, W. N., AND ZHANG, X. Efficient program execution indexing. In *Proc. of ACM PLDI* (2008), ACM, pp. 238–248.

[44] ZHANG, K. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern recognition 28*, 3 (1995), 463–474.

[45] ZHANG, X., TALLAM, S., GUPTA, N., AND GUPTA, R. Towards locating execution omission errors. In *Proc. of ACM PLDI* (2007), ACM, pp. 415–424.

[46] ZHANG, J., RENGANARAYANA, L., ZHANG, X., GE, N., BALA, V., XU, T., AND ZHOU, Y. Encore: exploiting system environment and correlation information for misconfiguration detection. In *ASPLOS* (2014), pp. 687–700.

Notes

[1]We did not include social networks because their main app functions appear to differ (*e.g.,* Facebook, twitter, snapchat).

[2]F-method-only paths will be patched to other tasks as discussed in §3.5.

[3]Refer to [9] for more details on calling context tree construction.

[4]A maximal one-to-one matching matches the most nodes in the two trees.

[5]Our approach to tracking events is similar to AppInsight [35], but instead of instrumenting app binary, we directly modify the Android framework to track asynchronous calls. Since we use timestamp and thread id in addition to hashCode to track objects, we did not see problems due to hashCode collisions in our experiments.

[6] Since EPROF does not break down app energy drain into native code methods - it simply folds native code's energy into JNI boundary method for Java, DIFFPROF would not be able to identify tasks in native code. In practice, tasks typically start from framework callback Java methods and hence most of the task structures are captured in the Java methods that invoke the native code.