A Survey of Performance Optimization for Mobile Applications

Max Hort, Maria Kechagia, Federica Sarro, Mark Harman

Abstract—To ensure user satisfaction and success of mobile applications, it is important to provide highly performant applications. This is particularly important for resource-constrained systems such as mobile devices. Thereby, non-functional performance characteristics, such as energy and memory consumption, play an important role for user satisfaction. This paper provides a comprehensive survey of non-functional performance optimization for Android applications. We collected 156 unique publications, published between 2008 and 2020, that focus on the optimization of performance of mobile applications. We target our search at four performance characteristics: responsiveness, launch time, memory and energy consumption. For each performance characteristic, we categorize optimization approaches based on the method used in the corresponding publications. Furthermore, we identify research gaps in the literature for future work.

Index Terms—mobile applications, android, non-functional performance optimization, software optimization, literature survey.

1 INTRODUCTION

The relevance of mobile (handheld) devices, such as the so-called smartphones, has been ever growing for the past ten years, reaching an estimate of 3.2 billion smartphone users in 2019.1 Smartphones can nowadays be considered as the main information processing devices for users. With smartphones, users cannot only receive and make phone calls, but also execute similar tasks as those performed on personal computers (e.g., surf the internet, perform calculations, pay bills).

Even though mobile devices are powerful, they represent resource-constrained devices making the development of applications that can run on them (mobile applications) challenging. This means that the functionality and performance of mobile applications depend on the characteristics of mobile phones (e.g., their physical memory, processors, battery) and on the current execution context (e.g., how many applications run at the same time on a mobile phone).

To ensure the success of an application (e.g., whether it will be used, updated, or uninstalled), developers aim to maximize user experience quality, and, consequently, user satisfaction. User satisfaction is mainly influenced by functional (Does the application operate as the user expects?) and non-functional (How does the application perform?) application characteristics. Examples of functional issues can include missing or buggy features (e.g., a game application that functions in a different way than presented in its description). An example of a non-functional characteristic is the energy consumption of an application. Regardless of an application’s functionality, users will be dissatisfied if the application drains the battery of their mobile devices within minutes.

Finkelstein et al. 6 found that the success of mobile applications in terms of downloads is correlated to the rating that the application attracts. These ratings are recorded by App Stores (e.g., Google Play, Apple Store, BlackBerry World). In 2018, the number of total applications downloaded amounted to 194 billion 1 with every user having a multitude of different applications installed on their phone 2, 3. With such a high number of applications, 75% of mobile device usage is filled by mobile applications 3. While several studies show the importance of fixing software bugs that hinder applications’ smooth function, non-functional performance characteristics have shown to have a strong impact on user satisfaction as well. This impact can be seen in the user reviews of real-world mobile applications:

• “This app is destroying my battery. I will have to uninstall it if there isn’t a fix soon.” 13
• “It lags and doesn’t respond to my touch which almost always causes me to run into stuff.” 19
• “Bring back the old version. Scrolling lags.” 3
• “Makes GPS stay on all the time. Kills my battery.” 3
• “Too much memory usage for a glorified web portal ad machine.” 18

Furthermore, Banerjee and Roychoudhury 20 conducted a study on 170,000 user reviews, and showed that poor performance and energy consumption lead to application downvotes from users. Among all causes of downvotes, energy consumption caused the highest ratio of uninstallations.

Given the importance of non-functional performance characteristics on user satisfaction and the consequent success of mobile applications, as well as new optimization
approaches that are developed each year, we provide a comprehensive overview of existing approaches for non-functional performance optimization of mobile applications. This can be used by practitioners, developers, and researchers to search for approaches appropriate to their needs (e.g., “How can I reduce energy consumption only by changing application source code?” or “Can I improve responsiveness by applying changes to the device hardware?”). Furthermore, we provide information on dependencies among non-functional properties, which reside in mobile applications. We focus our review on the Android platform, since it is open-source software and has the highest market share among mobile platforms at the time of writing.

Initially, we gathered and analyzed existing work to detect non-functional performance characteristics (Section 3). Based on this, we identify four non-functional characteristics, which describe user-perceived performance of mobile applications, and thereby their success, i.e.: responsiveness, launch time, memory consumption, and energy consumption. For each of these characteristics, we have categorized previous work based on the optimization level (e.g., optimization applied to application, platform, or hardware level) and proposed optimization type (e.g., prefetching, preloading, display), as shown in Figure 1.

We found that the majority of approaches to optimize responsiveness applied changes to the application’s source code, while launch time was improvement by changes to the Android platform. Approaches that optimize memory apply changes to both the application and Android platform’s source code. The majority of work was concerned with optimizing energy consumption. Moreover, we were able to detect relationships among the four non-functional performance characteristics (e.g., energy consumption can increase with an improved responsiveness of applications).

To the best of our knowledge, this is the first survey to investigate multiple non-functional performance characteristics and their relationships. To summarize, our work:

1) provides a comprehensive literature review of the state-of-the-art research on the optimization of non-functional characteristics for mobile applications;
2) provides a categorization of existing optimization approaches based on their level and type;
3) identifies challenges and opportunities for future research in this area.

We have made publicly available some additional resources [21] and an online version of the work reviewed in this survey, which we will keep up-to-date by accepting external contributions [22].

The rest of this paper is structured as follows. Section 2 presents an overview of mobile devices and mobile-application ecosystems. The search methodology is described in Section 3. Sections 4-7 describe research on non-functional performance optimization. These refer to: responsiveness (Section 4), launch time (Section 5), memory consumption (Section 6), and energy consumption (Section 7). A discussion of results considering all non-functional performance optimization characteristics is given in Section 8. Section 9 presents related work and Section 10 outlines threats to validity. Section 11 concludes this survey.

2 BACKGROUND

This section presents an overview of the context of this survey. Initially, we present key terms and definitions regarding the architecture of mobile devices. Then, we focus on the characteristics of the Android platform that we take into account in this work. Finally, we explain how the function and performance of mobile applications can affect users.

2.1 Mobile Devices

Mobile devices are embedded systems that consist of hardware and software components. Figure 2 illustrates a representative architecture of a mobile device.
The foundation of mobile devices is their hardware. The capacity of hardware components, such as physical memory, processors, and battery, is constrained. Additionally, mobile devices come with a growing set of embedded sensors, including accelerometers, digital compasses, GPS, microphones, and cameras, which enable the emergence of personal, group and community-scale sensing applications [24]. However, the use of these sensors in applications requires a higher energy consumption [25].

The software of mobile devices comprises two basic layers: the mobile platform and the hosted mobile applications. The mobile platform (e.g., Android, iOS) consists of an embedded operating system (OS) that connects hardware with software components. It offers services such as memory management, networking, and power management. In mobile platforms, software libraries, which are used for the interaction with components, such as the database and media framework, are on the top of the OS. Mobile applications (e.g., calculator, photos, contacts) are either provided by a mobile framework (e.g., the Android platform) or third-party applications provided by online stores for mobile applications (e.g., Google Play Store, Apple iOS App Store, AppNokia, Samsung, BlackBerry World and Windows Phone Store).

2.2 Android

This survey focuses on the Android platform because it is currently the most used mobile platform[1] and open-source software, facilitating the analysis and evaluation of mobile systems. The following paragraphs present the main components of the Android platform.

Android is an embedded system based on the Linux OS. The Linux kernel links hardware and software components of a mobile device. It manages services such memory, processes, power, and networking access, and it offers drivers for flash memory, Bluetooth, WiFi, keyboard and audio. On top of the Linux kernel, lies the Android Runtime (ART), which is essential for running different applications. Each application runs as a separate process, having its own virtual machine instance.

The Android platform provides several methods and tools for improving the performance of mobile applications. For instance, memory is freed by the OS if the available memory on a device is low. To achieve that, Android uses the Low Memory Killer (LMK) to remove the Least Recently Used (LRU) cached application from the memory. Cached data is stored in the virtual memory, as long as memory is available [26]. In order to optimize the cache memory, and address problems such as duplicated pages in virtual memory, Kernel Same-page Merging (KSM) [27] and zRAM [28] are applied by Android [29]. Even though these methods optimize the cache memory, they consume power while they are being executed.

Furthermore, Android offers a variety of tools in its SDK to analyze system information and support application development. [4] Tools can be used for logging (LogCat), retrieving application and system information (APK Analyzer, Dumpsys, SysTrace), as well as for simulations and debugging (Android Debug Bridge, ADB Manager). Finally, Android provides developers with selected performance measures (Android Vitals) that use real user data, in case users have agreed on providing such information. If that happens, several metrics related to startup time, battery usage, and crash stack traces are recorded. Such metrics can assist developers to monitor memory and energy consumption, to identify synchronization issues, and to avoid application crashes [30].

2.3 User Experience

User experience and satisfaction are important factors that can ensure the success of mobile applications [31]. Application rating, or user satisfaction with an application, has been shown to correlate with the number of downloads [32]. After installing and using an application, users are able to make judgements regarding their satisfaction. Reviews regarding user satisfaction of mobile applications appear in App Stores and new users consider them in order to decide whether they will download an application or not.

To achieve a high level of user satisfaction, developers focus on improving both the functional and non-functional characteristics of mobile applications [1], [2], [3], [13], [15], [16]. Apart from fatal issues with functionality, such as application crashes [3], non-functional performance characteristics also shape users’ perception [1]. Non-functional performance characteristics are the first characteristics to, potentially adversely, affect users [15] and can lead to application uninstallations [2]. In the following, we describe functional and non-functional characteristics of mobile applications that can affect user experience.

Functional characteristics describe whether an application is doing what it is supposed to do (i.e., its behavior). Frequent complaints about functional aspects of applications include freezes or crashes [1], functional errors, such as not getting push notifications, and the removal of features [3].

Non-functional characteristics determine how an application carries out (performs) its behavior. Even though it is difficult to measure and judge non-functional characteristics [33], they represent a vital part of user satisfaction for mobile applications. Related work analyzes a range of different non-functional characteristics regarding applications’ performance [1], [2], [3], [13], [15], [16].

Different schemes exist to classify functional and non-functional characteristics of applications [34], [35], [36]. Among these, the FURPS model [35], [37] clearly distinguishes performance characteristics from other functional and non-functional characteristics, as follows:

- Functionality: feature set, capabilities, generality, security;
- Usability: human factors, aesthetics, consistency, documentation;
- Reliability: frequency/severity of failure, recoverability, predictability, accuracy, mean time to failure;
- Performance: speed, efficiency, resource consumption, throughput, response time;

we distinguish between four different non-functional perfor-
work on Android performance optimization techniques to
A detailed description of each of the four non-functional
Among these characteristics, we are interested in the
The purpose of this survey is to gather and categorize
prior to systematically searching online repositories, we
3.1.1 Preliminary Search
Our literature review on performance optimization includes
keywords used to guide our repository search. Keywords are divided into five categories. Firstly, the
Furthermore, keywords that belong to the
3.1.3 Selection
To ensure that the publications found during our search
To assesses whether the publications satisfy our inclusion
3) Memory consumption describes the amount of oc-
4) Energy consumption is associated with the battery life.
A detailed description of each of the four non-functional
3 Survey Methodology
The purpose of this survey is to gather and categorize research work published in the mobile computing and software engineering literature that refers to the optimization of non-functional performance of Android applications.
As this is an emergent topic and there is limited related work on Android performance optimization techniques to perform a systematic literature review (according to the guidelines of Kitchenham [41]), we conduct a comprehensive literature review. In the following, we present our search methodology in detail, starting with a preliminary and venue search, followed by a repository search and snowballing.
3.1 Search Methodology
Our literature review on performance optimization includes publications that refer to optimization techniques on mobile applications and measurement of application performance.
3.1.1 Preliminary Search
Prior to systematically searching online repositories, we conducted a preliminary search. The goal of the preliminary search is to gain a deeper understanding of the field and assess whether there is a sufficient amount of publications that allows for subsequent analysis. Based on these results, we distinguish between four different non-functional performance characteristics: responsiveness, launch time, memory and energy consumption.
Other than Sadeghi et al. [42], who refined keywords during their search, we perform a preliminary search to guide our repository search. Additionally, we use the results of the preliminary search to define keywords (Table 2) and venues (listed in Section 3.1.2).
3.1.2 Repository Search
Proceeding the preliminary search, we conduct a search of six established online repositories (IEEE, ACM, ScienceDirect, Scopus, arXiv, and Google Scholar). We have gathered publications from 2008 to February 2020, since the first version of Android was released in 2008.
To ensure that we provide an exhaustive literature search, we manually examine relevant venues from the field of software engineering and mobile computing, which we encountered during the preliminary search. We search venues with at least five publications in our preliminary search.
• Conferences: ICSE, ASE, MSR, MobiSys, MobileHCI, MobileSoft, UbirComp, CHI, ESEC/FSE.
• Journals: IEEE TSE.
Table 2 lists keywords used to guide our repository search. Keywords are divided into five categories. Firstly, the keywords that belong to the Platform category ensure that the selected publications deal with mobile platforms, particularly Android. Furthermore, keywords that belong to the Responsiveness, Launch time, Memory, and Energy categories filter publications referring to non-functional performance characteristics. We restrict search results to publications that contain at least one platform keyword and one non-functional keyword in their title.
To ensure that the publications found during our search are relevant to the context of non-functional performance optimization of mobile applications, we consider the following inclusion criteria:
• The publication should refer to at least one of the non-functional performance characteristics investigated in this survey (e.g., responsiveness, memory, energy, and launch time), or to an approach that profiles at least one of the mentioned performance characteristics.
• The publication investigates the proposed methods on smartphones with an Android OS.
To assesses whether the publications satisfy our inclusion criteria, we manually examined every publication using the process adopted by Martin et al. [43], as follows:
1) Title: First, all those publications whose title clearly does not match our inclusion criteria are excluded;
2) Abstract: Second, the abstract of every remaining publications is checked. Publications whose abstract does not meet our inclusion criteria are excluded at this step;
3) Body: Publications that passed the previous two steps are then read in full, and excluded if their content does neither satisfy the inclusion criteria nor contribute to this survey.
Based on the above three-stage process and inclusion criteria, we iteratively reduce the amount of publications
TABLE 1
Results of the Repository Search. The number of papers retained at each stage of the search (e.g., Hits, Title, Abstract, Body) is given for each online repository (e.g., Google Scholar, IEEE, Scopus, ACM, Science Direct, arXiv) and non-functional performance characteristic (Responsiveness, Launch Time, Memory, Energy). Google Scholar is abbreviated with GS; Science Direct is abbreviated with SD. For example, searching for “responsiveness” in GS retrieves 835 publications, among those 31 have a relevant title, 24 of those have an abstract satisfying our inclusion criteria, and a total of 11 (of those 24) publications are included in our survey after reading them entirely.

<table>
<thead>
<tr>
<th>Repository</th>
<th>Responsiveness Hits</th>
<th>Responsiveness Title</th>
<th>Responsiveness Abstract</th>
<th>Responsiveness Body</th>
<th>Launch Time Hits</th>
<th>Launch Time Title</th>
<th>Launch Time Abstract</th>
<th>Launch Time Body</th>
<th>Memory Hits</th>
<th>Memory Title</th>
<th>Memory Abstract</th>
<th>Memory Body</th>
<th>Energy Hits</th>
<th>Energy Title</th>
<th>Energy Abstract</th>
<th>Energy Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Scholar (GS)</td>
<td>835</td>
<td>31</td>
<td>24</td>
<td>11</td>
<td>90</td>
<td>10</td>
<td>9</td>
<td>1</td>
<td>392</td>
<td>16</td>
<td>11</td>
<td>4</td>
<td>1129</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IEEE</td>
<td>180</td>
<td>21</td>
<td>17</td>
<td>9</td>
<td>1</td>
<td>9</td>
<td>0</td>
<td>7</td>
<td>76</td>
<td>5</td>
<td>9</td>
<td>3</td>
<td>52</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scopus</td>
<td>146</td>
<td>20</td>
<td>15</td>
<td>7</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>57</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>269</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACM</td>
<td>73</td>
<td>17</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arXiv</td>
<td>17</td>
<td>3</td>
<td>15</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE 2
Keywords Used for the Repository Search.

<table>
<thead>
<tr>
<th>Category</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>android, smartphone, app, apps</td>
</tr>
<tr>
<td>Responsiveness</td>
<td>responsiveness, performance</td>
</tr>
<tr>
<td>Launch time</td>
<td>launch, start</td>
</tr>
<tr>
<td>Memory</td>
<td>memory</td>
</tr>
<tr>
<td>Energy</td>
<td>energy, battery, power</td>
</tr>
</tbody>
</table>

3.1.4 Snowballing
After a collection of publications is obtained from the repository search, we proceed to inspect the related work of the publications selected in the previous search to gather cited publications using snowballing [44]. We apply one level of backwards snowballing.

3.2 Selected Publications
Table 1 shows the results of the repository search. The amount of publications found during each step of the search is listed.

In the following, we give the number of unique publications after each stage of the search procedure in addition to the number of newly added publications:

1) Preliminary search: 96
2) Repository search: 174 (+80)
3) Venue search: 180 (+4)
4) Snowballing: 252 (+72)
5) Author feedback: 297 (+45)

In addition to the discussed stages of the search procedure (1-4), we added 45 publications based on the feedback from the authors cited. Among all 294 publications, 156 unique publications optimize at least one non-functional performance characteristic. These 156 publications were published in 97 different venues. We further classify top publication venues (A, A* according the CORE ranking Portal) regardiing their category based on the ACM’s Computing Classification System (CCS). Among these, a majority of publications is obtained from Software Engineering (32%), and Computer Systems Organisation (29.33%) venues. The remaining publications are retrieved from Mobile Computing (20%), Networks (17.33%), and Security and Privacy (1.33%) venues. A full list of conferences and journals is available online [21].

The publication distribution over the entire search period is illustrated in Figure 3. Note, a publication can contribute to more than one subtotal if it explicitly optimizes more than one non-functional performance characteristic. During our search, we found ten publications that optimize more than one non-functional performance aspect [38], [45], [46], [47], [48], [49], [15], [50], [51], [52]. Among these, there is one publication that optimizes three characteristics (responsiveness, energy consumption, memory consumption) [45], while the others optimize two. Section 8.3 provides further details on the relationships between performance characteristics.

Based on our search results, we devise the categorization of approaches shown in Figure 1. These categories consist of approaches (e.g., offloading, code optimization) and elements of the Android platform (e.g., Low Memory Killer, API). In the remainder of the survey, we discuss our search results in detail.

4 Responsiveness
Responsiveness refers to the ability of mobile applications to respond to user interactions fast and smoothly. An application is highly responsive when the time it takes to respond to user requests is minimal. A highly responsive application offers high user satisfaction as users prefer not to wait when interacting with an application. On the other hand, applications that are not highly responsive can cause users to lose interest and switch to other applications.
hand, an application with poor responsiveness can have a negative impact on user perception, and on its success [53]. Specifically, Tolia et al. [54] argued that response times lower than 150ms do not negatively affect user satisfaction. In fact, delays that last almost one second do not significantly affect users, but start making them aware of these delays, whereas delays that last more than one second indeed make users “unhappy” [55]. Willocx et al. [55] stated that response times under 100ms appear as instantaneous to users. Furthermore, users would accept response times up to a few seconds, if delays were to occur rarely [55].

To detect and fix hot spots in mobile applications that may cause response-time delays, developers use several techniques, including profiling and optimization approaches. Section 4.1 presents methods and approaches for profiling, whereas Section 4.2 presents approaches to optimize responsiveness. Section 4.3 summarizes our findings.

4.1 Profiling
There are several profiling approaches that developers use to measure the responsiveness of applications and locate hot spots for improvement. Popular profiling techniques include the measurement of page loading time [56], the measurement of the overall frame time and the calculation of the number of delayed frames [51], as well as the estimation of Central Processing Unit (CPU) time [57]. Furthermore, responsiveness can be measured at different levels of the Android platform, considering the UI [58] and hardware components [59].

Several tools have been developed to measure the responsiveness of mobile applications. Specifically, Ravindranath et al. [60] introduced APPINSIGHT to detect critical paths in applications, which represent bottlenecks for user transactions. Hong et al. [61] proposed PERFPROBE, a profiling approach to diagnose hardware and software causes for slowdowns with runtime information. Kim et al. [62] conducted performance testing, using unit tests, at early development stages of the applications to identify response-time delays. Kang et al. [63], [64] presented a technique that analyzes application performance focusing on particular asynchronous executions. Wang and Rountev [65] introduced a novel approach that profiles responsiveness by tracking the usage of mobile resources such as bitmap or SQLite databases. Kwon et al. [66] proposed MANTS, a framework that predicts the execution time of an application while using particular inputs.

4.2 Optimization Approaches
For a high responsiveness of mobile applications, developers apply several categories of optimization approaches. In the following, we describe techniques found in literature. Table 3 lists our findings.

**Table 3**

<table>
<thead>
<tr>
<th>Category</th>
<th>Authors/Ref</th>
<th>Year</th>
<th>Venue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offloading</td>
<td>Kemp et al. [67]</td>
<td>2010</td>
<td>MobICASE</td>
</tr>
<tr>
<td></td>
<td>Chun et al. [68]</td>
<td>2011</td>
<td>EuroSys</td>
</tr>
<tr>
<td></td>
<td>Ra et al. [69]</td>
<td>2011</td>
<td>MobIoSy</td>
</tr>
<tr>
<td></td>
<td>Kosta et al. [70]</td>
<td>2012</td>
<td>INFOCOM</td>
</tr>
<tr>
<td></td>
<td>Gordon et al. [71]</td>
<td>2012</td>
<td>OSDI</td>
</tr>
<tr>
<td></td>
<td>Gordon et al. [72]</td>
<td>2015</td>
<td>MobIoSy</td>
</tr>
<tr>
<td></td>
<td>Das et al. [73]</td>
<td>2016</td>
<td>IACC</td>
</tr>
<tr>
<td></td>
<td>Montella et al. [74]</td>
<td>2017</td>
<td>CCPE</td>
</tr>
<tr>
<td></td>
<td>Shen and Haas [75]</td>
<td>2018</td>
<td>J-Sac</td>
</tr>
<tr>
<td>Antipatterns</td>
<td>Jin et al. [76]</td>
<td>2012</td>
<td>SIGPLAN</td>
</tr>
<tr>
<td></td>
<td>Yang et al. [77]</td>
<td>2013</td>
<td>MOBS</td>
</tr>
<tr>
<td></td>
<td>Nistor et al. [78]</td>
<td>2013</td>
<td>ICSE</td>
</tr>
<tr>
<td></td>
<td>Liu et al. [79]</td>
<td>2014</td>
<td>ICSE</td>
</tr>
<tr>
<td></td>
<td>Onogkis and Takada [77]</td>
<td>2014</td>
<td>DeMobile</td>
</tr>
<tr>
<td></td>
<td>Hecht et al. [80]</td>
<td>2015</td>
<td>ASE</td>
</tr>
<tr>
<td></td>
<td>Habchi et al. [81]</td>
<td>2016</td>
<td>ASE</td>
</tr>
<tr>
<td></td>
<td>Hecht et al. [82]</td>
<td>2016</td>
<td>MobileSoft</td>
</tr>
<tr>
<td></td>
<td>Li et al. [83]</td>
<td>2019</td>
<td>SANER</td>
</tr>
<tr>
<td>Refactoring</td>
<td>Lin et al. [84]</td>
<td>2014</td>
<td>FSE</td>
</tr>
<tr>
<td></td>
<td>Okur et al. [85]</td>
<td>2014</td>
<td>ICSE</td>
</tr>
<tr>
<td></td>
<td>Lin et al. [86]</td>
<td>2015</td>
<td>ASE</td>
</tr>
<tr>
<td></td>
<td>Lyu et al. [87]</td>
<td>2018</td>
<td>ISSTA</td>
</tr>
<tr>
<td></td>
<td>Fong et al. [88]</td>
<td>2018</td>
<td>ICSE-NIER</td>
</tr>
<tr>
<td>Prefetching</td>
<td>Higgins et al. [89]</td>
<td>2012</td>
<td>MobIoSy</td>
</tr>
<tr>
<td></td>
<td>Zhai et al. [90]</td>
<td>2018</td>
<td>ICSE</td>
</tr>
<tr>
<td></td>
<td>Cho et al. [91]</td>
<td>2018</td>
<td>CoNEXT</td>
</tr>
<tr>
<td></td>
<td>Malavolta et al. [92]</td>
<td>2019</td>
<td>ICSE-NIER</td>
</tr>
<tr>
<td>Programming languages</td>
<td>Battyak et al. [93]</td>
<td>2009</td>
<td>MobileWare</td>
</tr>
<tr>
<td></td>
<td>Lee and Jeon [94]</td>
<td>2010</td>
<td>ICCAS</td>
</tr>
<tr>
<td></td>
<td>Lee and Lee [95]</td>
<td>2011</td>
<td>iCAST</td>
</tr>
<tr>
<td></td>
<td>Lin et al. [96]</td>
<td>2011</td>
<td>BRCA</td>
</tr>
<tr>
<td></td>
<td>Saborido et al. [97]</td>
<td>2018</td>
<td>EMSE</td>
</tr>
<tr>
<td>CPU &amp; GPU</td>
<td>Wang et al. [98]</td>
<td>2013</td>
<td>CGO</td>
</tr>
<tr>
<td></td>
<td>Cheng et al. [99]</td>
<td>2013</td>
<td>IWSHIS</td>
</tr>
<tr>
<td></td>
<td>Thongkaew et al. [100]</td>
<td>2015</td>
<td>JIP</td>
</tr>
<tr>
<td>I/O operations</td>
<td>Nguyen et al. [101]</td>
<td>2015</td>
<td>MobIoSy</td>
</tr>
<tr>
<td></td>
<td>Mao et al. [102]</td>
<td>2018</td>
<td>ITCSJDI</td>
</tr>
<tr>
<td>Hardware components</td>
<td>Kim and Shin [103]</td>
<td>2015</td>
<td>ICUMC</td>
</tr>
</tbody>
</table>

**Fig. 3.** Number of publications on non-functional performance optimization per year.
in mobile applications. PERFChecker is built on top of the SOOT [38] Java optimization framework and analyzes applications at a bytecode level. PERFChecker applies static code analysis to detect antipatterns. Other tools focus on detecting particular types of bugs related to application responsiveness [51], [76]. For instance, Nistor et al. [76] searched for repetitive computations in code loops, following the intuition that repetitive behavior is likely to be optimizable. To detect repetitive behavior, they created TODDLER, an automated oracle to analyze memory access patterns. Hecht et al. [51] used a static analysis tool called PAPRIKA [78] to detect three types of code smells (Internal Getter/Setter, Member Ignoring Method, and HashMap Usage). They investigated the removal of code smells in an empirical study, obtaining responsiveness improvements of up to 12.4%.

Furthermore, Inefficient Image Displaying (IID) can cause performance degradation (e.g., repeated and redundant image decoding). Li et al. [80] developed the static analysis tool TAPIR to detect IID issues, which can be strongly correlated with antipatterns.

Finally, responsiveness-related bugs can be detected using predefined rule sets (e.g., efficiency rules) [75], [77] and test amplification (insertion of artificial delays in application source code) [83].

Refactoring can be performed to utilize efficient programming practices. Lin et al. [81], [83] provided an analysis showing that even though applications include concurrent code, they often contain bugs or end up with executing the source code sequentially. For concurrent code execution, and higher responsiveness, the authors located and refactored long-running operations, using the two tools ASYNCHRONIZER and ASYNCDROID. Okur et al. [82] developed two tools to refactor asynchronous code in Windows Phone applications. Asynchronous code is converted by ASYNCFINDER and common misuses in asynchronous code are corrected by CORRECTOR. Their empirical study showed that developers accept the proposed changes to asynchronous code. Lyu et al. [47] applied static analysis to change inefficient database operations that are placed in within a loop. Database operations called in loops can cause Repetitive Autocommit Transaction (RAT), which creates a new transaction in each iteration of the loop. Furthermore, Feng et al. [84] mined optimization patterns from GITHUB projects, considering performance-aware APIs, which can be manually injected into the source code of mobile applications and improve their performance.

Prefetching refers to a technique that caches data in advance, so that it can timely provide required data when it is needed. Higgins et al. [85] provided a library called Informed Mobile Prefetching (IMP) that assigns the task of determining when to prefetch data to the mobile system, rather than leaving the choice to developers. Developers solely specify which items could benefit from prefetching, while IMP determines whether and how prefetching is handled, based on responsiveness, battery lifetime, and mobile data usage. Zhao et al. [86] proposed a technique named PALOMA, that prefetches HTTP requests to reduce responsiveness latency. PALOMA uses string analysis to detect prefetchable content in the application source code. While users navigate in an application, PALOMA uses short pauses (“user think time”) for prefetching. Choi et al. [87] identified resource dependencies with static analysis to automatically generate acceleration proxies for dynamic prefetching. Application binary files are analyzed to detect HTTP(s) messages, which are later used for prefetching. As static analysis lacks certain information, missing information of HTTP(s) requests is added at runtime. Lastly, Malavolta et al. [88] proposed a technique called NAPPA that prefetches network requests based on user navigation patterns.

Programming languages can impact the processing speed of applications and therefore responsiveness. Android provides the Native Development Kit (NDK) that allows developers to write native C/C++ code. Native instructions are directly executed by the CPU and, therefore, they provide a better performance over non-native ones [93]. Several empirical studies compared the performance of programming languages, and found that native C code reduces the running time of the same algorithms written in Dalvik Java code [92], [90], [91], [89]. Furthermore, efficient implementation choices, such as which map variant to use (e.g., HashMap, ArrayMap, and SparseArray) can improve responsiveness [45].

CPU and Graphics Processing Unit (GPU) adaptations can accelerate the execution of time-consuming programming tasks and increase application responsiveness. Wang et al. [93] proposed ACCELDROID to accelerate the execution of bytecode on the HW/SW co-designed processor of Android. Therefore, instead of translating bytecode twice, this is only translated once. Cheng et al. [94] provided guidelines to map applications to the Android platform (e.g., whether to use CPU or GPU and how many cores are used). This mapping is platform as well as task-dependent. An optimal performance choice can avoid performance degradation. Additionally, Thongkaew et al. [95] developed architectural hardware extensions that can fetch and decode Dalvik bytecode directly.

I/O operations have an impact on responsiveness and can enable optimization [100]. For instance, Nguyen et al. [50] proposed an approach that adapts the prioritization of read and write operations for avoiding slowdowns. Mao et al. [96] introduced a trace collection tool to identify redundant I/O requests in mobile applications and eliminate them to reduce response times. As redundancy is minimally shared among applications, they performed an application-aware optimization.

Hardware components, such as the use of embedded Multimedia Cards (eMMC), can be investigated for responsiveness improvements. Kim and Shin [97] studied whether additional features of eMMCs are utilized by Android smartphones, and reduced the I/O latency.

4.3 Summary
Responsiveness is a non-functional performance characteristic concerned with the time an applications needs to respond to user requests. In practice, this is either measured in time (ms) [56], [57] or in frames [51]. Several tools have been proposed to measure responsiveness and detect responsiveness issues [50], [61], [83], [84]. Since responsiveness measures the duration required to complete computations, a naive approach to improve response times is to move the computations from the smartphone to devices with less restrictions. This approach is called
“offloading” and requires additional infrastructure (e.g., external servers for computation). If such an infrastructure is not available, other approaches can be followed, which in majority are applicable to a source-code level (23/38).

Furthermore, changes that have been applied to mobile applications’ source code include the removal of bad programming patterns (antipatterns), and the usage of good programming practices (e.g., concurrency with the help of refactoring [31], [33], [52], and prefetching of content [35], [36], [37], [38]). Other than changing source code after responsiveness issues have been detected, a carefully considered choice of the right programming language can lead to improvements (e.g., native C/C++ is faster when executed on CPUs [39], [40], [41], [42]). Lastly, changes to the hardware have achieved responsiveness improvements. This can directly happen at CPU or GPU level [33], [44], which can carry out computations, or on other hardware components (e.g., memory [47]).

5 Launch Time

During the launch of a mobile application, operations and data are loaded to make the application available to the user. Therefore, launch time is the first performance characteristic of a mobile application that users have the opportunity to notice. Launch time directly influences user experience and satisfaction. Nagata et al. [101], Song et al. [38] and Kim et al. [15] defined the launch time of an application as the required time until user input is accepted. Yan et al. [102] described the Total Launch Time (TLT) of an application as the needed time until the entire content, including asynchronously loaded content, can be displayed to the user.

Furthermore, Song et al. [38] found that cold start time (when an application is started from scratch) has a significant impact on the application launching experience of users. Developers can address this issue by analyzing their applications’ source code to identify and fix bottlenecks that possibly increase launch time. The following sections discuss profiling methods (Section 5.1) and approaches to optimize launch time and cold start issues (Section 5.2). A summary is given in Section 5.3.

5.1 Profiling

Profiling approaches have been proposed for locating issues in mobile applications that may increase the application launch time. Using monitoring functions in the source code of Android applications, in the Android platform, and in third-party libraries used by Android applications, developers can pinpoint performance issues causing launch-time delays [101], [103]. Also, developers can profile the usage of system resources to pinpoint the application launch completion [104]. Additionally, Nguyen et al. [50] studied launch delays of an application as the time taken in kernel mode and the time spent waiting for disk network operations.

To understand how launch time can affect user behavior, Song et al. [38] investigated logs from application usages. Other approaches for assessing the launch behavior of applications include monitoring of: handling of I/O requests [105], system memory usage [15], restart ratio of applications (the number of cold starts over all application launches) [38] and user satisfaction with regards to launch-time delays [106].

5.2 Optimization Approaches

In order to reduce launch time, developers apply optimization techniques. Table 4 summarizes publications found in literature, which are described in the following.

<table>
<thead>
<tr>
<th>Category</th>
<th>Authors [Ref]</th>
<th>Year</th>
<th>Venue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preloading</td>
<td>Yan et al. [102]</td>
<td>2012</td>
<td>MobiSys</td>
</tr>
<tr>
<td>Low Memory Killer</td>
<td>Chung et al. [38]</td>
<td>2013</td>
<td>TECS</td>
</tr>
<tr>
<td>Preloading</td>
<td>Prodduturi and Phatak [110]</td>
<td>2013</td>
<td>BT</td>
</tr>
<tr>
<td>Preloading</td>
<td>Song et al. [38]</td>
<td>2014</td>
<td>TECS</td>
</tr>
<tr>
<td>Preloading</td>
<td>Baik and Huh [111]</td>
<td>2014</td>
<td>ICSE</td>
</tr>
<tr>
<td>Preloading</td>
<td>Vimal and Vreddy [112]</td>
<td>2015</td>
<td>RAICS</td>
</tr>
<tr>
<td>Memory</td>
<td>Singh et al. [113]</td>
<td>2016</td>
<td>IOTA</td>
</tr>
<tr>
<td>Preloading</td>
<td>Kim et al. [15]</td>
<td>2016</td>
<td>TECS</td>
</tr>
<tr>
<td>Preloading</td>
<td>Lee et al. [129]</td>
<td>2017</td>
<td>I-SAC</td>
</tr>
<tr>
<td>Preloading</td>
<td>Li et al. [114]</td>
<td>2017</td>
<td>IWCMC</td>
</tr>
<tr>
<td>Prefetching</td>
<td>Joo et al. [105]</td>
<td>2011</td>
<td>FAST</td>
</tr>
<tr>
<td>Prefetching</td>
<td>Nguyen et al. [50]</td>
<td>2015</td>
<td>MobiSys</td>
</tr>
</tbody>
</table>

The FALCON approach by Yan et al. [102], which refers to an OS extension, preloads applications and application-specific content based on the context (e.g., location) and usage patterns. For this purpose, spatial and temporal features are designed based on an extensive analysis. Usage patterns include the use of weather applications in the morning, or playing games at home.

Additionally, application predictions determine the applications to be launched next [38], [48], [108], [109] and when they are going to be used [107]. In particular, Nagata et al. [104] analyzed the relationship of application launch time regarding the number of preloaded classes. For this, they manually selected a number of preloaded classes, and showed for one application that the launch time is reduced when the number of preloaded classes is high. The prediction of the next application to be used has been also investigated for restructuring user interfaces [116], [117], [118].

The Low Memory Killer (LMK), which removes the LRU application from memory, may not lead to optimal results [15], [113], because users do not always rely on recently used applications. To address this issue, several studies introduce techniques that determine which data should be removed from memory. Specifically, Song et al. [38] and Li et al. [114] devised models to detect patterns in
application usages based on application cold start times. Decisions for the LMK are based on usage patterns to prioritize data of applications that are likely to be launched.

Instead of removing the LRU application from memory, choices can be made based on application cold start times [15], the required storage size [104], and the importance of an application to a user [110]. Furthermore, Baik and Huh [111] analyzed usage patterns and determined a threshold on how many processes to keep in memory before freeing them. If this limit is fixed to a high value, more applications can be kept in memory, leading to fewer restarts.

Memory can be adapted to suit application launches better. Accordingly, Joo et al. [105] proposed the use of SSDs instead of HHDs to speed up application launch time. This approach was not designed for mobile devices; however, mobile devices use NAND flash memory as secondary storage carrying almost identical performance characteristics as SSDs. One could therefore apply this approach to mobile devices as well. Furthermore, Kim et al. [15] proposed the use of Non-Volatile Memory (NVM) to store frequently used applications and shared libraries among applications. Shared data is stored on Phase-Change Memory (PCM). Therefore, less data needs to be loaded when launching applications.

I/O operations impact the application launch time, as their speed can be seen as a performance bottleneck during application launch [15]. In fact, Nguyen et al. [50] analyzed the impact of read and write operations on launch time, as mobile devices wait for I/O operations to complete. As application launches are dominated by read operations (five times as many read operations as write operations [50]), this can have a high impact on overall application launch time. A prioritization of read and write operations avoids slowdowns and reduces launch time.

5.3 Summary

Launch time describes the time required until a user input is received [15, 38, 101] or the entire application content is displayed [102], after an application has been started by the user. The application launch completion has been profiled according to the system’s resource-management usage [104].

Due to the high negative impact that cold starts can have on user satisfaction [38], the majority of launch time optimization methods (15/18) prevent cold starts, and reduce application launch time. On one hand, preloading of application data can be applied to spend loading times before the application launch, and reduce the actual launch itself. For this purpose, predictions are used to determine applications that are likely to be used next, based on usage patterns [16, 38, 49, 102]. On the other hand, changes to memory management (LMK) have achieved similar results. In contrast to preloading, which loads desired application data, changes to the LMK to keep important data in memory for a longer time. Both of these approaches require access to the Android OS to implement the required adaptations.

An increased speed of memory operations (e.g., usage of SSDs over HHDs [103]), shared libraries among applications [15], and I/O prioritization of read over write operations [50] have been also applied to reduce launch time.

6 Memory

Memory is a critical resource for embedded systems [26], such as mobile devices. Specifically, in Android devices, data can be loaded either by the Android platform (to be shared across multiple running applications) or by each application, separately. Application data is stored in separate heaps, per application [39]. The main memory is typically shared between the CPU and GPU [52]. Therefore, a considerable amount of the main memory is occupied by graphic processing operations [26]. Kim et al. [15] classified applications in two categories, based on their memory consumption, stable and unstable. The memory consumption of stable applications increases within the first ten seconds of the applications’ launch, and it stabilizes afterwards. The memory consumption of unstable applications increases steadily, and it does not stabilize.

The following sections present approaches that are used to measure the consumed memory of mobile applications (Section 6.1), optimize application memory consumption (Section 6.2). At the end, we provide a summary (Section 6.3).

6.1 Profiling

Different tools and approaches have been proposed to measure the memory usage of applications. For instance, memory consumed by mobile applications can be measured by kernel memory footprints [119], garbage collection calls [51], physical memory dumps, and logging information [29]. Vimal and Trivedi [112] used the Dalvik Debug Monitor Server (DDMS) to analyze memory footprints of Android components and measure memory consumption. Tools such as ANDROSCOPE by Cho et al. [59] have been also used to analyze the performance (including memory) of all the layers of the Android platform. Furthermore, ANDROBENCH [120] and ANDROSTEP [121] are benchmark tools that assess the storage performance of Android devices by analyzing logs from read and write I/O operations.

6.2 Optimization Approaches

To reduce the memory usage of mobile applications developers use different categories of optimization approaches. The following paragraphs summarize the relevant approaches found in literature. Table [3] lists the representative studies.

Antipattern coding practices can be used to identify code that is likely to lead to memory leaks. Memory leaks occur when applications constantly request memory while running [122], or when unused objects are being kept in memory longer than required [123].

Hecht et al. [51] showed in an empirical study that memory consumption can be reduced by correcting code smells. In particular, memory can be improved in terms of memory usage and number of garbage collection calls. Shahriar et al. [123] developed memory leak patterns for Android applications and used fuzz testing to emulate and detect memory leaks. A total of three fuzzing types (application, resource, and API) are used in their experiments.

10. DDMS is deprecated and was removed from Android Studio 3.2. Android offers other tools to carry out the functions of DDMS [https://developer.android.com/studio/profile/monitor].
which discovered crashes due to memory leaks in real-world applications. Furthermore, memory leaks can be identified by analyzing memory dumps [52], [139], the activity lifecycle [125], source code patterns [124] or memory execution information by applying process control block hooking [122].

**Garbage collection** is used in Android to manage memory and identify unused objects that can be removed [52]. Since version 2.2., Android uses a stop-the-world (STW) garbage collector [29], [126]. This stops other operations to free the memory and resume them afterwards, resulting in pauses that can negatively affect user experience [59].

Different garbage collector designs have been evaluated for improvements: reference counting garbage collection [52], [126], concurrent garbage collection [39] and generational garbage collection [52], [128]. Lim et al. [127] proposed a memory partitioning scheme, which partitions available memory into two nodes (for critical and uncritical applications). If one node runs out of memory, only the memory of this node is freed.

**Deduplication** is a technique to remove redundant pages from memory. While duplicated memory reduces available memory for other applications, Android is prone to have page-level duplication in memory [29]. Lee et al. [29] developed a system (MEMSCope) to analyze memory duplication in Android OS. MEMSCope identifies memory segments that contain duplicated memory pages. One of the disadvantages of deduplication is the additional computation needed to detect and merge redundant pages. Therefore, Kim et al. [129] proposed a computationally efficient deduplication scheme, considering background applications that do not update memory contents and need to be scanned only once.

**Memory management** changes can be applied to achieve further improvements in memory usage by mobile applications. For example, *swapping* is a technique that reclaims memory by writing inactive memory pages to secondary storage (e.g., eMMC). Kim et al. [135] proposed a swapping scheme (Application-Aware Swapping) that considers OS processes in the swapping decision. For example, swapping an application to secondary memory is not useful if the LMK is about to remove it from memory, freeing the used memory pages. Other approaches utilized NVM for swapping [132], [137].

Journaling in Android applies a write-twice behavior, to ensure reliability, which reduces system performance by additional write operations. Kim et al. [130] proposed an architecture to reduce storage accesses for journaling. They use non-volatile memory for this purpose. Among others, Jeong et al. [131] eliminated the journaling of unnecessary metadata. Nguyen et al. [134] proposed iRAM, a system that cleans low-priority processes to maintain a high level of free memory. Kim and Bahn [135] evicted write-only-once data from the buffer cache to improve the utilization of cache space. Kim et al. [133] proposed an approach to group memory pages with the same lifetime to alleviate fragmentation of I/O buffers.

**GPU buffers** have been analyzed by Kwon et al. [26], who introduced a compressing scheme. Once an application goes to the background, its GPU buffers are treated as inactive and compressed. If the application is launched in the foreground, GPU buffers are decompressed.

**Programming languages** influence the choice of language constructs that further impact storage requirements. Escobar De La Torre and Cheon [138] analyzed the impact of the Java language constructs on the allocated memory. For instance, `for`-`each` loops require more memory than equivalent code snippets using regular loops. Removing those constructs (`for`-`each` loops, lambda expressions and the Stream API) reduces memory requirements [138]. Saborido et al. [45] showed that map implementations consume different amounts of memory. Specifically, `ArrayMap` uses less memory than `HashMap`.

### 6.3 Summary
Memory describes the occupation of device memory by applications, and is critical for resource-constraint systems [26]. For Android applications, data is either shared between multiple applications, or loaded separately by each application.

Memory has been measured according to kernel memory footprints [119], garbage collection calls [51], physical memory dumps, and logging information [29]. Memory can be analyzed by tools provided by the Android OS [112], and external tools provided by researchers [59], [120], [121].

Memory consumption has been reduced by removing code smells from application source code [51]. In particular, memory leaks (e.g., constantly requesting memory [122], or keeping unused objects in memory [123]) have a negative impact on memory consumption.

Memory consumption has been further improved by changes in the Android OS. For example, garbage collection, which is used to free memory in Android, can use different strategies for freeing memory [110]. Another approach is the removal of redundant data from memory (deduplication) [29], [129]. Improvements can furthermore be achieved by changes in swapping [134], [136], [137] and journaling strategies [130], [131].

## 7 Energy
Embedded systems include several components that consume battery. CPU, LCD, GPS, audio and WiFi services are
power-intensive components \[25, 40\]. Due to the limitation in battery size and stored energy \[140, 141, 142\], reducing energy consumption is gaining more and more relevance \[143\]. In general, optimizing energy consumption depends on individual usage \[7, 144\].

The following sections outline methods and tools to profile energy consumption (Section 7.1) and reduce the energy consumption of mobile applications (Section 7.2). A summary is given in Section 7.3.

### 7.1 Profiling

Several measurements and prediction approaches have been used to profile energy consumption on mobile devices. Hoque et al. \[145\] discussed two ways to measure energy consumption: with external instruments and self-metering. This section gives an overview of respective profiling techniques.

A common approach to determine energy consumption is to investigate hardware components. Zhang et al. \[25\] measured power consumption using battery voltage sensors and knowledge of battery discharge behavior. Additionally, fuel gauge chips \[146\] and Battery Monitoring Unit \[147\] can be used to measure energy consumption. Other approaches use physical power meters to measure energy consumption \[99, 148, 149, 150, 151\]. Morales et al. \[143\] used a digital oscilloscope for high frequency energy measurements. Ferrari et al. \[152\] designed a Portable Open Source Energy Monitor \[POEM\] to measure energy consumption of applications at a control flow level. Bokhari et al. \[153\] built energy models based on CPU utilization and lines of code, as external meters can be expensive and not easy for developers to set these meters up.

Other than measuring energy consumption with physical devices, several studies make energy consumption estimates. Energy consumption estimates can be performed based on hardware utilization and system-calls \[154\]. Android kernel monitoring \[155\], pixel information \[156, 157\], user behavior \[7\], data transmission-flow characteristics \[158\], code level \[159, 160\] and source-code line level \[161\]. Jabbarvand et al. \[162\] proposed COBWEB, a search-based technique to generate test suites for energy testing. These tests are able to execute energy-greedy parts of the code. The computational cost of such a testing technique can be reduced by test-suite minimization \[163\]. Mittal et al. \[164\] presented an emulation tool WATTSON to estimate energy consumption during application development. CPU time has been used as a proxy for energy consumption. However, it is not as accurate as other techniques, because voltage is scaled dynamically and multiple hardware components are used \[161\]. One should consider that errors during the measurement and estimation of energy consumption, as noise, can be introduced by various hardware components, such as a rising temperature of the battery \[165\]. This impacts the number of samples required for ensuring statistical significance when comparing the energy consumption of applications \[166\]. Validation approaches should consider the level of noise to compare solutions fairly \[167\].

### 7.2 Optimization Approaches

Several techniques have been applied to reduce the energy consumption of mobile applications, which are discussed in the following and summarized in Table 6.

#### Offloading

e.g., transferring computationally expensive tasks to external devices, can be used to reduce energy consumption \[149\]. Cuervo et al. \[149\] developed MAUI, a system that supports automatic and developer-specified code offload. MAUI determines which method to execute remotely based on the current state of the device at runtime. Offloading decisions can be motivated by device status \[173\].
execution times [171], network conditions [45, 168, 169], or developer decisions [172]. Bolla et al. [174] proposed the concept of Application State Proxy (ASP) to offload entire applications. ASP transfers internet-based applications to other network devices, when they are kept in the background. As long as no new events occur (e.g., messages), applications are kept in the proxy, which reduces the resource load on the smartphone. Corral et al. [175] applied offloading to matrix multiplication and image processing tasks to reduce energy consumption.

**Prefetching**, e.g., the caching of data transmissions and advertisements in advance, can be used to reduce energy consumption. Balasubramanian et al. [176] distinguished applications in delay-tolerant and applications that can benefit from prefetching, to decide which networking technology (3G, GSM, WiFi) to use. Mohan et al. [178] and Chen et al. [179] prefetched multiple ads to reduce energy consumption induced by downloads. Dutta and Vandermeer [180] achieved energy reductions with caching of up to 45%, even with small cache sizes (e.g., 250MB). Yang and Cao [179] formalized the prefetching for energy reductions as an optimization problem. Two approaches (greedy and discrete) are investigated to minimize energy consumption with regard to the network condition (LTE).

**Antipatterns** can be defects such as energy bugs that will likely drain energy. Pathak et al. [181] defined energy bugs as errors that cause the system to unexpectedly consume energy. Banerjee et al. [182] categorized energy inefficiencies into two categories: energy hotspots and energy bugs. Energy hotspots cause high battery consumption even though the hardware utilization is low. Energy bugs prevent the idle state of smartphones causing undesired battery consumption without user activity. Pathak et al. [181] categorized energy bugs caused by hardware (faulty battery, hardware damage) and software (OS, configurations, applications) and proposed a framework to detect the causes of energy bugs. Pathak et al. [182] focused on detecting a particular type of energy bug (no-sleep bug) via static analysis. A no-sleep bug occurs when application components are being kept active when a smartphone is in an idle state, without the necessity of being kept active. Banerjee et al. [183] created a framework that automatically generates tests to detect energy bugs. Each test contains a sequence of user interactions that are aimed at revealing energy bugs. As system calls are a primary source for energy bugs, a directed search is used to generate test cases containing system calls. Zhang et al. [181] developed ADEL (Automatic Detector of Energy Leaks), to identify energy leaks caused by network operations. Liu et al. [183] created GREENDROID, a tool that extends Java PathFinder (JPF) to automatically detect energy problems and report actionable information to combat these problems. Jabbarvand and Malek [184] proposed μDROID, a mutation testing framework, that can be used to detect energy inefficiencies. This framework uses 50 different mutation operators and the similarity of power traces between original application and mutants is used as the test oracle. The detection and removal of energy bugs is not simple, as high energy consumption in applications is not necessarily a sign for wasted computations [141].

**Refactoring**, for example, by using energy-efficient algorithms, can be used to reduce energy consumption. Pathak et al. [185] manually restructured the source code of applications to make efficient use of high power states of components. They observed that applications consume I/O energy in distinct lumps. Bundling these lumps can reduce energy consumption. Similarly, Alam et al. [187] optimized the placement of waitlock calls. Lyu et al. [187] refactored database operations to avoid inefficiencies and reduce energy consumption. Another approach is to change the choice of colors used in an application, as the power consumption of displays is effected by the displayed color [237]. This goes as far as some applications consume double the energy as they would do if colors were optimized for energy consumption [157]. Li et al. [188] proposed an approach to automatically change the colors used in web applications. Linares et al. [155, 238] used multi-objective optimization to reduce the energy consumption of GUIs, while offering visually similar colors to the original design. Bruce et al. [189] applied Genetic Improvement (GI) to find a more energy efficient version of applications. Mutation operations were applied to the source code of a Boolean satisfiability solver, to reduce energy consumption as a measure of fitness. Bokhari et al. [163] applied approximate computing on Rebound [11], a Java Physics library, to achieve a trade-off between accuracy and energy consumption.

Another approach to automatically refactor applications is to follow energy efficiency guidelines [190, 191, 194]. Cito et al. [146] adapted application binaries to adjust the frequency of network requests to advertisements and analytics based on the battery status. Anwer et al. [186] adapted permissions and corresponding source code of applications based on user requirements, which can for example prevent the unconscious sending of an SMS.

Morales et al. [143] showed that there is a correlation between anti-patterns and energy consumption of mobile applications, and proposed the use of multi-objective search to find a set of refactoring sequences able to simultaneously improve code design quality (including the removal of energy smells) and reduce energy consumption. Banerjee et al. [192] performed an automatic repair of energy bugs with static and dynamic analysis. Cruz and Abreu [193] manually fixed antipatterns based on Android performance-based guidelines.

**Power states** determine the operating modes of hardware components, which require different amount of energy [154]. Power state transitions can be initiated by hardware components, but are usually performed by the OS [40]. As idle power consumption accounts for approximately 50% of the total energy consumption in a smartphone, it is suggested that using different power modes to shut down components is useful to reduce energy consumption [140]. Metri et al. [142] developed BATTERYEXTENDER, a tool that enables users to reconfigure device resources to reduce battery consumption. For this purpose, battery consumption of components is predicted with little computational overhead, by using energy profiling. Users are able to pick a period of time for which they want to reduce energy consumption and then choose which components to put in an idle power state to save energy. Bokhari and Wagner [192] proposed a framework to optimize default settings of smartphone

components to reduce energy consumption. This problem is formulated as an optimization problem, to minimize energy consumption by changing settings of components based on user behavior. Rao et al. [198] dynamically selected system configurations (CPU frequency and memory bandwidth) that reduce energy consumption while maintaining a user-specified level of responsiveness. Ding et al. [196] determined power modes based on wireless signal strength, as a poor signal strength drains energy. Based on this, network traffic can be delayed under poor signal strength and continued when the network strength improves. Kim et al. [195] limited power modes based on battery status. By minimizing the amount of transferred data in Social Networking Services, energy consumption can be reduced. For instance, Pyles et al. [150] switched WiFi to a low power or sleep mode during periods where it is not being used.

Displays are under constant energy consumption while mobile devices are used. Dong et al. [199] were the first to study the transformation of GUI colors of OLED displays to reduce energy consumption. Their automatic transformation can be applied on GUI elements (structured) or on pixel information (unstructured). Anand et al. [200] adjusted the brightness of screens to reduce the backlight level of the display. Lin et al. [201] reduced backlight energy consumption for mobile streaming applications, while other work dimmed areas of the screen [188], [203], [205], [208]. Other approaches include the adaptation of pixels [202], reduction of frame refreshes [203], [206], [209], [210], pixel density [206], [207] and resolution [211].

CPU clock frequency impacts energy consumption [140]. Nagata et al. [99] proposed a method that adjusts CPU clock frequency based on application requirements. Hsiu et al. [215] allocated computing resources based on the sensitivity of different applications. Application sensitivity states can be HIGH (interactive), MEDIUM (foreground) or LOW (background). Tseng et al. [214] adapted the allocation of CPU resources to applications based on their delay-sensitivity. Further approaches adjust the frequency and voltage of devices (Dynamic Voltage and Frequency Scaling) [212], [213]. CPU frequency [218] or the number of cores [216]. Muhuri et al. [217] considered linguistic feedback from users to adapt CPU frequency accordingly. They proposed the approach Per-C for Personalized Power Management Approach (Per-C PPMA), which collects user feedback about their degree of satisfaction when using an application. This can be applied to not only reduce energy consumption, but also to improve user satisfaction.

APIs impact energy consumption, as Li et al. [239] showed that 91.4% of applications consume more than 60% of their energy with APIs. Linares et al. [224] analyzed usage patterns of “energy-greedy” APIs and give recipes to reduce energy consumption. To support their quantitative and qualitative exploration of API usage pattern, they mined thousands of method calls and API usage patterns. Among those, there are usage patterns that have an unavoidable, high energy consumption, and others which can be improved. An example for energy-greedy APIs is GPS. Paek et al. [219] adapted the rate of GPS, and only turns on GPS, when the current location estimate is uncertain. Turning on GPS indoors is also avoided. Other approaches reduce the sampling rate of GPS [220], [221], [222], [223].

Protocols can be used by mobile devices to optimize the energy consumption of networking technologies. Ra et al. [225] designed an algorithm to optimize the energy-delay trade-off of delay-tolerant applications that can benefit from low-energy WiFi connections. Energy can be reduced if mobile traffic is delayed to a situation where WiFi is available [228]. Pyles et al. [227] saved energy by prioritizing WiFi traffic based on application priority. Li et al. [231] bundled HTTP requests to reduce energy consumption. Cheng and Hsiu [229] considered signal strength to reduce energy consumption when fetching location-based information. Nurminen [226] showed that parallel TCP downloading can be used to reduce energy consumption. Siekkinen et al. [230] reduced energy consumption of streaming applications by shaping LTE traffic into bursts. Hoque et al. [240] surveyed other approaches for optimizing the energy efficiency of streaming.

System strategies can be used to manage background processes. Martins et al. [233] introduced TAMER, an OS mechanism that allows rate-limiting of background processes to reduce energy consumption. TAMER imposes on events and signals that cause background applications to wakeup and thereby consume a higher amount of energy. Among others, TAMER can limit the frequency of notifications an application sends while it runs in the background. Chen et al. [232] avoided running applications in the background when they are not beneficial for user experience.

Memory management strategies can be used to change or increase memory to cope with higher requirements of applications. However, a larger main memory size leads to higher energy consumption [40], [132]. Energy reductions can be achieved by using non-volatile memory [236], Phase Change Memory [40] or adaptations to the garbage collection [233] and scheduling algorithms [234].

Programming Languages impact the responsiveness and energy consumption of applications. Nagata et al. [99] compared applications developed in different programming languages (Java, JNI and C) and showed that the energy consumption for JNI and C is smaller than for Java. A programming language construct that impacts energy consumption refers to maps (e.g., using HashMap over ArrayMap can reduce energy consumption by 16% [45]).

7.3 Summary

Energy consumption is a crucial characteristic of embedded systems since these devices have a limited battery size. Energy is consumed by applications (often by multiple applications at the same time), which use several components (e.g., GPS, audio, WiFi, display) [25], [40].

To profile energy consumption, two techniques have been pursued. On one hand, energy has been measured with either internal or external instruments [145]. On the other hand, energy consumption has been empirically estimated. For this purpose, various indicators have been investigated [154], [156], [161]. However, when profiling energy consumption, one should consider noise, which impacts the validity of measurements [165], [168], [167].

Many approaches have been proposed to optimize energy consumption. The majority of these approaches address...
the application source code (38/85). Changes to source code included the removal of bad coding practices (antipatterns) [181, 182, 192], or refactoring to include best practices [190, 191, 199]. Source code adaptation approaches can also focus on particular elements of devices, such as the display. For instance, energy consumption of displays has been reduced by changing colors used in applications [156, 237, 238]. Adaptions have been also applied to displays directly. Thereby, display energy consumption has been reduced by changing brightness, colors and dimming [188, 199, 200, 203, 205, 208].

Similar to responsiveness optimization, energy inefficient computations have been offloaded to external devices with fewer computational restrictions [149]. Adapting the CPU clock frequency, to adjust computational speed, have also reduced energy consumption [99]. An adjustment of components can be performed with different power states (e.g., setting components in an idle state when they are not required).

8 DISCUSSION

This section provides an overview and discussion of the optimization approaches presented in previous sections (sections 4 to 7). In the following, we present a timeline of important methods and techniques proposed for advancing mobile applications’ performance. Additionally, we discuss current challenges and opportunities in this field.

8.1 Timeline

Important changes to the Android Platform and publications concerned with optimization of non-functional performance characteristics are shown in the timeline in Figure 8. The first commercial Android version was introduced in 2008. In 2009, Dong et al. [199] studied the transformation of the GUI to reduce the energy consumption of mobile device displays.

In 2010, Kemp et al. [67] were the first that proposed an offloading framework for Android applications. This framework can be used to offload computationally expensive parts of mobile applications and improve both responsiveness and energy consumption.

In 2011, Pathak et al. [151] extended energy profiling of hardware utilization based on system-calls to provide fine-grained energy estimates. In 2012, Yan et al. [102] proposed an OS extension that preloads applications and application-specific content to improve the launch time of applications. In 2013, Android designers applied a big change to the Android platform, introducing ART over the previously used DVM. With this change, optimizations such as Ahead-Of-Time (AOT) compilation are applied by the Android platform to improve application performance.

In 2014, Liu et al. [13] characterized several types of performance bugs, which addressed different non-functional characteristics of Android applications and caused excessive resource consumption of memory and battery. Examples of performance bugs include GUI lagging, energy leaks, and memory bloat. Based on the knowledge of bugs and antipatterns or guidelines, Banerjee and Roychoudhury [190] proposed the automated refactoring of application source code in 2016.

In 2016, Android 7.0-7.1 introduced a new JIT Compiler. This allowed faster application installations and it reduced the size of compiled code. The 10th major Android version was published in 2019.

8.2 Optimization Approaches per Android Layer

Reading the related work, we observed that approaches are applied to different layers of mobile devices. The distribution of these approaches differs between the four non-functional performance characteristics we investigated. In Figure 8 we organize optimization approaches based on layers (application, platform, hardware) to non-functional performance characteristics (i.e., responsiveness, launch time, memory and energy).

The majority of approaches to optimize responsiveness are application-based techniques and apply changes to the source code (e.g., antipatterns, refactoring). Hardware and I/O optimization approaches are concerned with increasing the speed of reads and writes, faster fetching of bytecode as well as correct usages of CPU and GPU.

Launch time has not been improved itself by changes to applications, but, in majority, by changes in the Android platform. In particular, the preloading of applications [16, 40, 41, 42, 50, 104, 105, 106, 107, 108, 109, 241] and better choices for the LMK [151, 40, 41, 42, 50, 104, 105, 111, 112, 113, 114], to prevent cold starts from happening, are investigated frequently. The overall research direction is concerned with reducing the amount of cold starts rather than reducing the launch time itself.

Memory is mostly optimized by removing antipatterns [51, 52, 122, 124, 125] and applying different strategies for the garbage collection [39, 52, 96, 110, 126, 127, 128].
A great deal of research work optimizes energy consumption and almost every optimization category has been addressed for energy consumption. Unlike the optimization of responsiveness, launch time and memory usage, a large portion of approaches apply changes to the hardware, especially for screens. The power mode of components plays an important role as well, which is governed by the Android OS. Furthermore, API usage has a huge impact on energy consumption. In particular, the use of GPS. Alike responsiveness, changes to application source code contribute to energy savings. Approaches like offloading optimize both, responsiveness and energy consumption.

8.3 Relationship of Optimization Approaches

A major trade-off that can be seen is between energy consumption and almost every optimization category has been addressed for energy consumption. Unlike the optimization of responsiveness, launch time and memory usage, a large portion of approaches apply changes to the hardware, especially for screens. The power mode of components plays an important role as well, which is governed by the Android OS. Furthermore, API usage has a huge impact on energy consumption. In particular, the use of GPS. Alike responsiveness, changes to application source code contribute to energy savings. Approaches like offloading optimize both, responsiveness and energy consumption.

8.4 Challenges

Reflecting on the different optimization approaches and the relationship between non-functional performance characteristics, we have identified challenges and opportunities for future work. We begin by outlining three challenges that developers face when optimizing non-functional performance characteristics.
characteristics, followed by an overview of future work targeted by the surveyed publications, and by opportunities we detected.

**Cross-characteristic dependencies.** Section 8.3 shows that a challenge that developers face while optimizing applications’ performance refers to the handling of the dependencies among different performance characteristics. This means that while improving one characteristic, they may decrease the performance of another [40], [59], [132]. For example, optimization approaches that extend the OS or invoke additional computation cause additional energy consumption [29], [85], [95], [102], [104], [109], [129]. Therefore, there should be a balance that developers need to achieve among performance characteristics. While we pointed out works that achieved improvements in more than one performance characteristic (Section 8.3), it would be interesting if adverse relationships and trade-offs between non-functional performance characteristics receive more attention. One example for this is the energy consumption and responsiveness trade-off when adjusting CPU clock frequency [99].

**Testing cost.** There are different definitions of the four non-functional characteristics to determine how performance is measured (e.g., responsiveness measured in ms [56], [57] or frames [51], or different definitions of application launch completion [101], [102]). Another problem arises with noisy measurements (e.g., due to hardware components [165]), as we have seen in Section 7. Therefore, testing should consider variance in measurements to ensure statistical significance [165], [167]. An attempt to reduce the cost of testing is the usage of emulators [123], [164] or prediction of non-functional performance without executing an application. Prediction of performance has been applied for responsiveness [57], [69] and energy consumption [7], [154], [155], [156], [157], [158], [159], [160], [161].

**User satisfaction.** While improvements in non-functional characteristics without performance deterioration in other characteristics, can always be seen as something positive, quantifying the impact of improvements on user satisfaction remains challenging.

Two of the surveyed publications attempted to tackle this issues. For example, Muhuri et al. [217] collected linguistic feedback about user satisfaction (e.g., ranging from “very low” to “extremely high”) to examine the impact of performance on user satisfaction. This information has been used to adapt the CPU frequency. Zhao et al. [106] stated that collecting user feedback can be costly and inconvenient for developers. To overcome this issue, they mapped a user-perceived satisfaction score about launch times to the actual launch time delay, which is easily measurable. Such an incorporation of user satisfaction in the optimization procedure, could prove to be an interesting consideration for future work on other performance characteristics.

In addition to these challenges, the reviewed publications distinguished several fields of future work, including:

- Improvement and extension of prediction methods (e.g., for offloading, prefetching, and preloading) [38], [68], [69], [79], [119], [116], [132], [221],
- Investigation of antipatterns [13], [51], [58], [123], [224];
- Automation of the optimization process [51], [148];
- Extension of testing (e.g., usage of more devices and applications) [51], [183], [207];
- Improvement of measurements (profiling) [132], [189].

In particular, extensions of prediction methods include the consideration of additional information, such as the context (e.g., location, time) [38], behavioral patterns [86] and information about remote resources for offloading (e.g., processor speed, available memory) [67]. Future work on antipatterns is concerned with investigating a broader range of antipatterns [51], [58] and discovering new antipatterns or categories [13], [223], [224]. Automation could be applied to time-consuming tasks, such as finding causes for energy wastage [148] or the correction of antipatterns [148].

Based on our results, including Figure 5 we identified further gaps in the literature. For once, changes to applications are not investigated for launch time improvements. It could be investigated whether antipatterns, that exist for responsiveness, memory and energy usage, exist for application launch as well.

Automatic refactoring has been applied for both responsiveness and energy, individually. It could therefore be interesting to apply refactoring in a multi-objective setting, to optimize both. Lastly, changes to the platform have scarcely been used to improve responsiveness. There could be the potential to apply ideas from other non-functional characteristics, such as APIs for energy consumption.

9 RELATED WORK

In the following, we give an overview of literature related to the non-functional performance optimization of Android applications. At first, we look into optimization approaches for software engineering. We furthermore describe the developer and user perspective on mobile-application optimization.

9.1 Optimization in Software Engineering

This survey outlined optimization approaches for mobile applications. In the following, we present studies and insights on optimization with regards to software engineering.

During the development and optimization of software, it is important to consider software requirements [248], which can contain functional, non-functional, business and user requirements. Different techniques for prioritizing requirements [249], [250] can be applied to determine the importance of non-functional over functional characteristics.

Nonetheless, the performance of software is difficult to be measured, as it is pervasive and affected by various different aspects (e.g., the platform used) [251]. Software Performance Engineering (SPE) is an approach to measure and improve system performance [251]. SPE subsumes software engineering activities that are applied to meet performance requirements and achieve improvements. Profiling tools can be used to measure performance (e.g., for energy consumption [252], [253], GPUs [254], responsiveness [255], and memory [256]).

Building upon performance profilers, optimization and improvement techniques can be applied to software. Petke et al. [33] conducted a survey on genetic improvement of software. They mention improvements for non-functional characteristics for energy and memory consumption as well
as functional improvements including repairs and addition of new functionalities.


9.2 Developer Perspective

In the following, we outline the developer perspective on optimization and development of mobile applications, including challenges during the development process. While certain characteristics of smartphones and PCs are similar [105], developers encounter differences to conventional software development as mobile applications are smaller than traditional software [266].

Developers use tools to support the development of applications [267], profiling, and debugging [268]. Static analysis can be used to support developers to find bugs and inspect code [269]. Other tools perform security assessment, automated test case generation and detection of non-functional issues such as energy consumption [270]. While fixing non-functional performance bugs, developers need to consider the threat of introducing functional bugs [271] and hindering code maintainability [272]. In this context, Linares et al. [273] suggested that developers rarely implement micro-optimizations (e.g., changes at statement level).

When developing an application, developers need to decide which and how many platforms to use (e.g., Android, iOS, Windows OS). Note that each platform faces non-functional issues alike (e.g., antipatterns can be found in iOS [274], Windows [275]). This impacts the development effort, as multiple codebases need to be maintained. Development of applications for multiple platforms can be supported by cross-platform tools (CPTs) [276]. By doing so, developers compromise between user experience and the ability to publish an application on multiple platforms. Willocx et al. [55] found that CPTs lead to an increased launch time of applications.

Furthermore, devices vary in their available memory, CPU, and display size, which has implications on application performance [277]. Therefore, developers need to test their applications on multiple devices.

Further changes and added functionality to the Android OS can be imposed by phone vendors [278]. This leads to varying behavior across different smartphone types [279]. Khalid et al. [280] analyzed the number of different phones that use applications in order to help developers to decide how many devices to use when testing applications, as the rating varies for different phone types. Often, applications run on more than hundred different phone types, implying a huge computational effort if all devices would be tested for. They found that around a third of the devices account for 80% of the reviews and thereby usage, which can be used to prioritize which devices to use during testing. Lu et al. [281] prioritized devices to test applications based on the amount of user activity rather than the number of devices.

In order to analyze the performance of applications, the Google Play Store provides developers with pre-launch reports after an application is published. Tests are carried out on different devices and for up to five languages [282]. Another tool that developers can use to judge the quality and performance of applications are Android Vitals. Statistics including battery consumption, and crashes are collected from real users and reported to developers [30]. Comparisons with regard to non-functional performance characteristics, such as energy consumption, can also be measured with applications of the same category [283].

Frequently, developers use user reviews and test applications manually to detect performance issues and bugs [284]. Developers can change applications based on user reviews. Those reviews contain among others information about performance, bugs, problems or new features [3]. There is a huge number of reviews that are written for applications, where some of them contain relevant information for developers [285]. For this purpose, Chen et al. [19] developed a framework to filter informative reviews by applying text mining and ranking methods. Furthermore, reviews can be analyzed for trends [286], [287] and emerging issues [288].

Even though developer identities do not often impact the choice of applications (only 11% of users choose an application based on who developed it [2]), they significantly impact the quality and success of applications [1]. Other factors that are correlated with the rating of applications include the apk size, minimum required SDK, and number of images on the application description page [289].

9.3 User Perspective

In the context of mobile applications, user shows a high degree of individualism [290]. Therefore, not every optimization approach can be applied in a general fashion. This is why understanding user behavior and differences among users and user groups is important when improving performance. Ultimately it is the user who decides the changes to an application result in a higher level of satisfaction.

The user-perceived quality of an application is not only determined by the application itself, but also by the device and attributes of its components [291]. Aspects that lead to the most negative complaints in reviews are related to privacy, hidden costs and features of the application [3].

Users’ behavior varies in terms of number of interactions, amount of data received, interaction length and number of applications used [7]. Some users even show addictive behavior [292]. Approaches that aim to improve user experience (e.g., by reducing energy consumption or improving the responsiveness), should therefore be adaptive to user behavior. Making matters more difficult to predict, usage patterns can change within a few days [117].
Additionally, user behavior across countries shows significant differences as well [2]. Users of different countries prioritize other aspects of applications and show variations in rating applications as well as writing reviews. For example, users in China are more likely to rate an application, while users in Brazil are more likely to abandon a slow or buggy application. While there exist differences among Android users based on countries, a study on application launch performed by Morrison et al. [293] showed that similarities between Android and iOS exist.

Application usage can be seen as a sequence, with multiple applications used consecutively in a short period of time, whereas 68.2% of sequences only contain a single application [115]. Application usage varies based on the context, such as location [294] and/or time [115], [295].

10 Threats to Validity

In this section, we discuss potential threats to our survey based on internal and external validity.

Internal validity refers to problems of our methods that could threat the validity of results and claims made in this survey [9]. A potential threat to internal validity is the completeness of the reviewed literature. Ideally, every relevant publication is included after the search process; however, the risk of missing a publication cannot be eliminated. A relevant publication can be missed if the respective data source containing the publication is not properly searched. Another cause for missing a publication is that corresponding keywords are not included in our searching criteria. To address both causes, we performed a preliminary search to gather relevant keywords and venues to guide the literature search. Moreover, two authors independently carried out the filtering process and their results were cross-checked, in order to ensure reliability and reduce researchers’ bias. A different threat to internal validity refers to the precision of our results (e.g., the inclusion of irrelevant publications). To mitigate the impact of irrelevant publications, we check the title, abstract, and body of the publications examined as outlined in Section 3.1.3. Furthermore, there is a risk of drawing incorrect conclusions or claims. For this purpose, every stage of the search and analysis of results (e.g., repository search and categorization of approaches) has been performed by one author and cross-checked by another.

External validity describes the generalizability of our results outside of the given scope [296]. A potential, external threat is that the chosen non-functional performance characteristics are insufficient to describe optimization techniques for embedded systems. Through our preliminary search and analysis of related work (Section 3) we found that the selected four non-functional characteristics (responsiveness, launch time, memory and energy consumption) are representative. Another threat is the applicability of our study to other mobile platforms (e.g., iOS). While there are differences between iOS and Android, the general organization and functionality within embedded systems remain the same. We therefore argue that our categorization can be applied to mobile platforms, other than Android.

11 Conclusions

In this paper, we have provided an overview of the existing research work on non-functional performance optimization for Android applications published between 2008 and 2020. Our survey presents optimization approaches for non-functional performance characteristics (e.g., responsiveness, launch time, memory, and energy). It also shows relationships among these characteristics, and identifies research gaps for potential future works. We hope that this survey will help researchers and developers to have a holistic perception on optimization approaches for mobile devices, the impact of these approaches, and the significance of different performance characteristics.

Acknowledgments

The work of Max Hort, Maria Kechagia, Federica Sarro and Mark Harman at UCL is supported by the ERC Advanced fellowship grant no. 741278 (EPIC). We would like to thank the members of the community who kindly provided comments and feedback on an earlier draft of this paper. Our living survey [22] has been inspired by Allamanis et al. [297].

References


---

Max Hort is a PhD student in Software Engineering at University College London, under the supervision of Prof. Federica Sarro and Prof. Mark Harman. His research research interests lie on software fairness, non-functional optimization of software, and search-based software engineering. Web page: [http://www0.cs.ucl.ac.uk/staff/M.Hort/](http://www0.cs.ucl.ac.uk/staff/M.Hort/)

Maria Kechagia is a Research Fellow at University College London. Previously, she was a postdoctoral researcher at the Delft University of Technology. Before that, she obtained a PhD degree from the Athens University of Economics and Business and a MSc degree from Imperial College London. Her research interests include program analysis, software testing, automated program repair, and software analytics. Web page: [http://www0.cs.ucl.ac.uk/staff/M.Kechagia/](http://www0.cs.ucl.ac.uk/staff/M.Kechagia/)

Federica Sarro is a Professor of Software Engineering at University College London. Her research covers Predictive Analytics for Software Engineering (SE), Empirical SE and Search-Based SE, with a focus on software effort estimation, software sizing, software testing, and mobile app store analysis. On these topics, she has published several articles in prestigious international venues including ICSE, FSE, TSE, TOSEM, and has received several international awards, including the FSE’19 ACM Distinguished Paper Award and the ACM SIGEVO HUMIES GECCO’16 Award. She has also been invited to serve on several programme, organisation and steering committees, and editorial boards of well-renowned venues, such as ICSE, FSE, ACM TOSEM, IEEE TSE, IEEE TEVC. Web page: [http://www0.cs.ucl.ac.uk/staff/F.Sarro/](http://www0.cs.ucl.ac.uk/staff/F.Sarro/)

Mark Harman works full time at Facebook London as a Research Scientist in a team focussing in AI for scalable software engineering. He also holds a part-time professorship at UCL. Previously, Mark was the manager of the Facebook team that deployed Sapienz to test mobile apps, which grew out of Majicke, a start up co-founded by Mark and acquired by Facebook in 2017. In his more purely scientific work, Mark co-founded the field Search Based Software Engineering (SBSE), and is also known for scientific research on source code analysis, software testing, app store analysis and empirical software engineering. He received the IEEE Harlan Mills Award and the ACM Outstanding Research Award in 2019 for this work. In addition to Facebook itself, Mark’s scientific work is also supported by the European Research Council (ERC), with an advanced fellowship grant, and has also been regularly and generously supported by the UK Engineering and Physical Sciences Research Council (EPSRC), with regular grants, a platform and a programme grant. Web page: [http://www0.cs.ucl.ac.uk/staff/M.Harman/](http://www0.cs.ucl.ac.uk/staff/M.Harman/)