Performance Diagnosis of Mobile Applications and Cloud Services

KANG, Yu

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

August 2016

Thesis Assessment Committee

Professor YIP Yuk Lap (Chair)

Professor LYU Rung Tsong Michael (Thesis Supervisor)

Professor LEE Pak Ching (Committee Member)

Professor Cao Jiannong (External Examiner)

Abstract of thesis entitled:

Performance Diagnosis of Mobile Applications and Cloud Services

Submitted by KANG, Yu

for the degree of Doctor of Philosophy

at The Chinese University of Hong Kong in August 2016

Mobility and scalability are two recent technological trends whose representative techniques are mobile computing and cloud computing. For example, cloud-based mobile computing, as a mix of the two, is becoming popular. However, current mobile computing and cloud services do not satisfy all of the performance requirements of the critical users. To improve user experience, it is necessary to enhance the performance of them. In this thesis, we propose a methodology for tuning the performance of the mobile applications and cloud services. This thesis contributes significantly to the performance tuning for both of them.

First, we examine how to tune the performance of mobile app. Rapid user interface (UI) responsiveness is a key consideration of Android app developers. However, service requests to the cloud often take a long time to execute. To avoid freezing the screen by blocking the UI thread, the requests are usually conducted under Android's complicated concurrency model, making it difficult for developers to understand and further diagnose the performance. This thesis presents `DiagDroid`, a tool specifically designed for Android UI performance diagnosis. The key notion of `DiagDroid` is that the UI-triggered asynchronous executions (*e.g.*, cloud service requests) contribute to UI performance, and hence their performance and their runtime dependency needs to be properly captured to

facilitate performance diagnosis. However, there are tremendous ways to start an asynchronous execution, posing a great challenge to profiling such executions and their runtime dependency. To this end, we properly abstract five categories of asynchronous executions as the building basis. They can be tracked and profiled based on the specifics of each category using a dynamic instrumentation approach carefully tailored for Android. `DiagDroid` can then profile the asynchronous executions in a task granularity, equipping it with low-overhead and high compatibility merits. The tool is successfully applied in diagnosing 33 real-world open-source apps; we find 27 performance issues in 14 apps. These case studies show the effectiveness of our tool for Android UI performance diagnosis.

Even when they are finely tuned, many services still require a long time to execute. Mobile apps should be tolerant of long processing time. Good user interface (UI) design is the key to successful mobile apps. UI latency, which can be considered as the time between the commencement of a UI operation and its intended UI update, is a critical consideration for app developers. There are currently no studies of how much UI latency a user can tolerate, and how to find UI design defects that cause intolerably long UI latency. As a result, bad UI apps are still common in app markets, leading to extensive user complaints. This thesis examines user expectations of UI latency, and develops a tool to pinpoint intolerable UI latency in Android apps. To this end, we design an app to conduct a user survey of UI latency in apps. Through the survey, we examine the relationship between user patience and UI latency. Therefore a timely screen update (*e.g.*, a loading animation) is critical to heavy-weighted UI operations (*i.e.*, those that incur a long execution time before the final UI update is available). We then design a tool that, by monitoring the UI inputs and updates, can detect apps that do not meet this criterion. The survey and the tool are open-source released on-line. We also apply the tool to many real-world apps. The results demonstrate the effectiveness of the tool in combating UI design

defects in apps.

Moreover, we investigate methods for improving the performance of cloud services. In a cloud, the optimal deployment of servers is key to providing better service for a wide range of mobile users. User experience, which is affected by client-server connection delays, is a key concern for optimizing cloud service deployment; however, it is a challenging task to determine the user experience of end users, as users may connect to a cloud service from anywhere. Moreover, there is generally no proactive connection between a user and the machine that hosts a service instance. In this thesis, we propose a framework to model cloud features and capture user experience. Based on the obtained user experience data, we formulate an optimal service deployment model. Furthermore, many services involve multiple clouds. For example, a service provider may use a hybrid cloud that provides several services in private and public clouds, or a service may request another service such as ticket booking agency services. Therefore, we formulate models for the multiple services co-deployment. Experiments based on a real-world dataset prove the effectiveness of the proposed models and algorithms.

In summary, we study the performance tuning problems for mobile apps and cloud services. We propose the mobile app and the cloud service performance tuning methods. Our tools are able to locate previously unknown performance issues in real-world mobile apps. Moreover, we collect a set of real-world QoS data from the Internet. The experiments based on the dataset prove the effectiveness of our models and algorithms. We have public released the dataset as well as the source codes of our tools.

論文題目：移動應用程序以及云服務的的性能調優

作者　　：康昱

學校　　：香港中文大學

學系　　：計算機科學與工程學系

修讀學位：哲學博士

摘要　　：

輕量化和規模化是當前技術發展的兩大趨勢，其各自的代表技術為移動計算和雲計算。例如基於云服務的移動計算作為這兩種技術的融合正變得流行。然而現有的技術仍不能滿足挑剔的用戶對應用程序性能的高要求。提高程序性能，改善用戶體驗對移動應用程序和雲服務都十分關鍵。良好的用戶體驗直接影響到應用程序能否成功吸引并留住用戶。在本論文中，我們提出了分別針對移動應用程序和雲計算的性能調優方法。

在本論文中，首先，我們研究了移動應用程序的性能調優。在移動端，用戶界面的響應速度是用戶體驗的關鍵因素也是衡量應用程序性能的重要指標。然而移動端的一些操作通常需要比較久的時間完成。為了防止由於阻塞主線程導致的界面凍結，耗時操作通常會利用移動操作系統（如 Android）中的複雜並發模式完成。這使得開發人員對於程序性能的理解以及進一步的調優變得困難。本論文針對 Android UI 性能調

優的需求實現了 DiagDroid 工具。DiagDroid 設計的主要觀念是 UI 觸發的異步任務（如雲服務請求）會影響 UI 性能，因此對程序的性能調優時，需要記錄分析這些異步任務的性能和他們之間的運行時依賴關係。然而，安卓提供了及其多樣的方式啟動異步任務，這使得記錄異步任務以及他們之間的依賴關係變得很困難。本論文中，我們將異步任務分為五類。對每一種類型的異步任務，我們設計了針對性的追蹤和記錄的方法。我們利用了 Android 的一些系統特性，基於輕量級動態插樁，實現了這些記錄方法。我們實現了 DiagDroid 工具，其能夠在任務級別記錄異步任務的生命週期，并具有低開銷、高兼容性的優點。我們使用 DiagDroid 對 33 個我們不熟悉的開源應用進行性能調優，發現了其中 14 個程序包含 27 個新的性能問題，這樣的結果是我們確信 DiagDroid 對開發者進行性能調優是有幫助的。

即使經過性能調優，受限於如網絡連接的速度以及服務的複雜度，許多操作仍然需要很長時間來完成。移動端的設計需要能夠容忍長延遲的操作。而好的界面設計能夠提升用戶對長延遲容忍度。而低界面延遲，也即從用戶操作至其相應的界面更新之間的延遲，是好的界面設計的重要因素。現今仍然缺乏一項完整的，針對用戶可以容忍多長的界面延遲，以及如何檢測不良界面設計中導致用戶不耐煩的長界面延遲缺陷的研究。因此，移動應用市場上充斥著很多界面設計有缺陷的應用程序，導致了大量的用戶抱怨應用程序的性能問題。本論文旨在更好的理解移動端用戶對界面延遲的期望，

基於此檢測并定位那些用戶不滿意的長延遲界面元素。為此，我們設計了一項對界面延遲的用戶調查。通過問卷調查我們發現了用戶耐心與界面延遲的關係。因此一個及時的屏幕更新（例如加載動畫）對于長延遲的用戶操作十分重要。我們設計了一個工具監視用戶數據和界面更新，并檢測沒有及時響應的界面元素。我們公佈了用戶調查和工具。我們將工具應用於在實際程序中，實驗結果表明我們的工具能夠有效檢測界面設計的缺陷。

此外，我們研究了雲服務的性能調優問題。在雲端，服務器的優化部署是為分散的移動用戶提供更好服務的關鍵。用戶體驗受用戶端和雲端連接延遲的影響，是服務器的優化部署需要著重考慮的因素。然而用戶可能從各個地方請求雲服務，用戶通常也不會主動訪問雲服務器提供用戶體驗數據，因此用戶體驗不容易獲取。本論文對雲服務建模，提出在雲服務框架中獲取及預測用戶體驗數據的方法。基於獲得的用戶體驗數據，論文設計了優化服務部署的方法，包括單服務部署和多服務協同部署的算法。我們收集了實際的用戶訪問數據，實驗驗證了論文方法的有效性。

綜上所述，本論文研究了移動應用以及雲服務的性能調優的方法，提出了移動端和雲端的性能調優方法。實驗驗證了論文方法的有效性。論文實現的工具集以及使用的數據集已開源發佈以供其他研究者使用。

Acknowledgement

First and foremost, I would like to express my sincere gratitude my supervisor, Prof. Michael R. Lyu, for his continuous support of my PhD study at CUHK. He has provided inspiring guidance and incredible help on every aspect of my research. From choosing a research topic to working on a project, from technical writing to doing presentation, I have learnt so much from him not only on knowledge but also on attitude in working. I will always appreciate his advice, encouragement and support at all levels.

Besides my supervisor, I would like to thank my thesis assessment committee members, Prof. Yuk Lap Yip and Prof. Pak Ching Lee, for their insightful comments and constructive suggestions to this thesis and all my term reports. Great thanks to Prof. Jiannong Cao from The Hong Kong Polytechnic University who kindly serves as the external examiner for this thesis.

I am also grateful to my oversea advisor and colleague, Prof. Kishor S. Trivedi and Dr. Javier Alonso, for their help on my visit to Duke University. During this visit, they have taught me a lot on the methodology of doing good research. I would thank Dr. Jianguang Lou, my mentor during the internship at Microsoft Research Aisa. I also thank Mr. Qingwei Lin, Xinsheng Yang, and Pengfei Chen, for their helps during my stay at Microsoft Research Aisa.

My sincere thank to Dr. Yangfan Zhou for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for the advices on life and career. I would like to thank Hui Xu, Zibin Zheng, Jieming Zhu, and Cuiyun Gao

for their valuable guidance and contribution to the research work in this thesis. I am also thankful to my other groupmates, Guang Ling, Qirun Zhang, Yuxin Su, Shenglin Zhao, Jichuan Zeng, Mingzhen Mo, Shouyuan Chen, Chen Cheng, Chao Zhou, Xin Xin, Yilei Zhang, Pinjia He, Jian Li, Tong Zhao, Jianlong Xu, and Hongyi Zhang who gave me advices and kind help.

My special thanks go to my dear friends, Zhongyu Wei, Ran Tao, Qing Yang, Fei Chen, Yixia Sun, Wei Yu, Zhe Zhu, Jihang Ye, Sheng Cai, Qinglu Yang, Ruolan Huang, Wenjie Wu, Zhiwei Zhang, Xinyuan Shi, Yanyin Zhu, Xin Feng, and many others for all the wonderful memories over these years. Without them, life would never be so enjoyable.

Last but not least, I would like to thank my parents for supporting me spiritually. Without their deep love and constant support, this thesis would never have been completed.

In dedication to my beloved family.

Contents

List of Figures

List of Tables

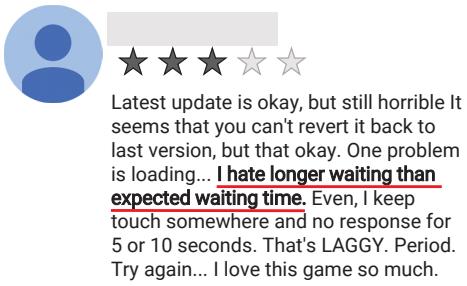
Chapter 1

Introduction

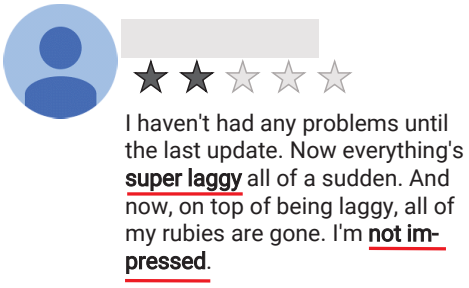
This thesis presents our research on performance tuning of mobile applications and cloud services, which attracts widely interests in both academia and industry. In Section 1.1, we motivate our research and provide an overview of the research problem. In Section 1.2, we highlight the main contributions of this thesis. In Section 1.3, we give the outline of the thesis.

1.1 Overview

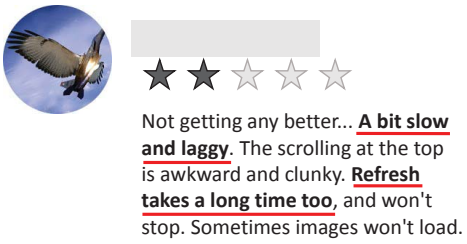
Today, mobile devices are an indispensable part of daily life. It is estimated that about a quarter of the world will use a mobile device in 2016 [2]. One report shows that on average, a US adult spends about three hours every day using mobile devices [16]. As a result, users are demanding high performance in mobile apps (*i.e.*, low UI latency). Mobile apps are expected to provide a responsive user interface (UI). Figure 1.1 depicts several user comments and their ratings of four different apps on the Google Play app store. The comments are negative and the ratings are below average for the apps with poor performance (*e.g.*, laggy). Moreover, users have expectations on the waiting time of some operations and they hate to wait longer than expected, as shown by the user comments in Figure 1.1a: "I hate longer waiting than expected waiting time". Therefore, both



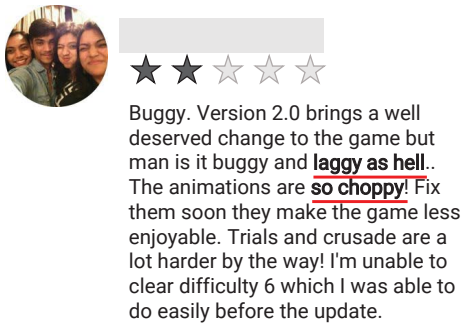
(a) Rates 3 vs. average rating 4.4



(b) Rates 2 vs. average rating 4.3



(c) Rates 2 vs. average rating 4.1



(d) Rates 2 vs. average rating 4.4

Figure 1.1: Examples of user ratings and comments on bad performance

the academic and industry communities are interested in tuning the performance of mobile apps and designing better UI [33, 153, 161].

Google offers several tips for enhancing the performance of apps [10]. Tools like TraceView and systrace [153] help developers to tune app performance. However, these guidelines and the labor-intensive tools cannot solve all of the performance problems. There are millions of mobiles apps (*i.e.*, more than 1.6 million apps on Google Play and 1.5 million apps on the Apple app store [19]). Many of them suffer from performance issues. Many developers do not know how to tune app performance. Handy tools for performance tuning are lacking.

We have also witnessed the rapid growing of cloud services. The tendency of cloud services is to deliver computation, software, and data access as services located in data centers distributed over the

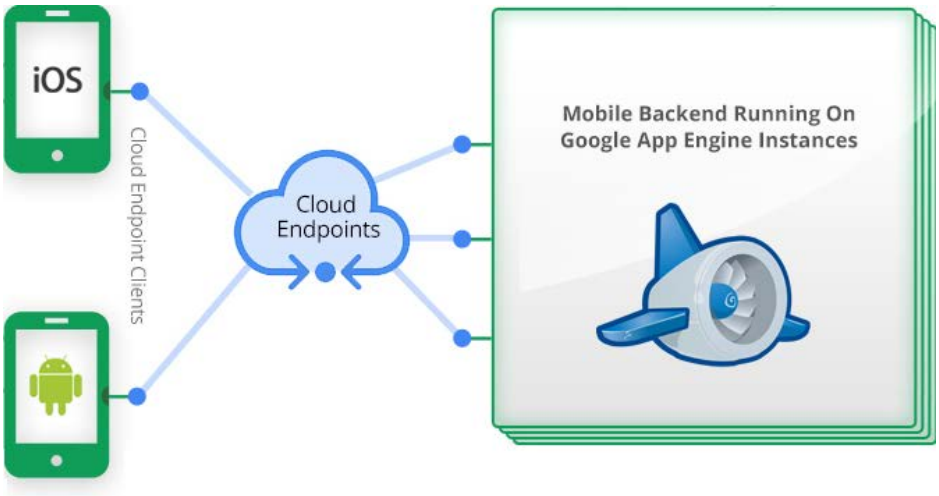


Figure 1.2: Google cloud endpoints & app engine architecture

world [39, 155, 182]. The limited performance of mobile apps is being extended through the use of cloud services. The combination of them, known as cloud-based mobile computing, is becoming popular - "The global mobile cloud market is forecast to hit $46.90 billion by 2019 from $9.43 billion in 2014 growing at a CAGR of 37.8%." [12]. For cloud-based mobile applications, many of the tasks are executed on the cloud. To improve performance of mobile devices, which have limited resources, time-consuming tasks can be offloaded to cloud. Tasks like voice-to-text speech recognition, route planning, and word translation are commonly run remotely on the cloud and the results are sent back to the mobile. Google app engine service combining with Google cloud endpoints [15], shown in Figure 1.2, is a representative architecture that can offload time-consuming tasks to a cloud.

However, the performance of cloud services and mobile apps are still unsatisfying. For example, the widely used railway ticket booking app `12306` [1], which has hundreds of millions of users in China, suffers from performance problems on both mobile app and its cloud service. For its mobile app, the app computes quite slowly and in many cases does not notify users about the progress

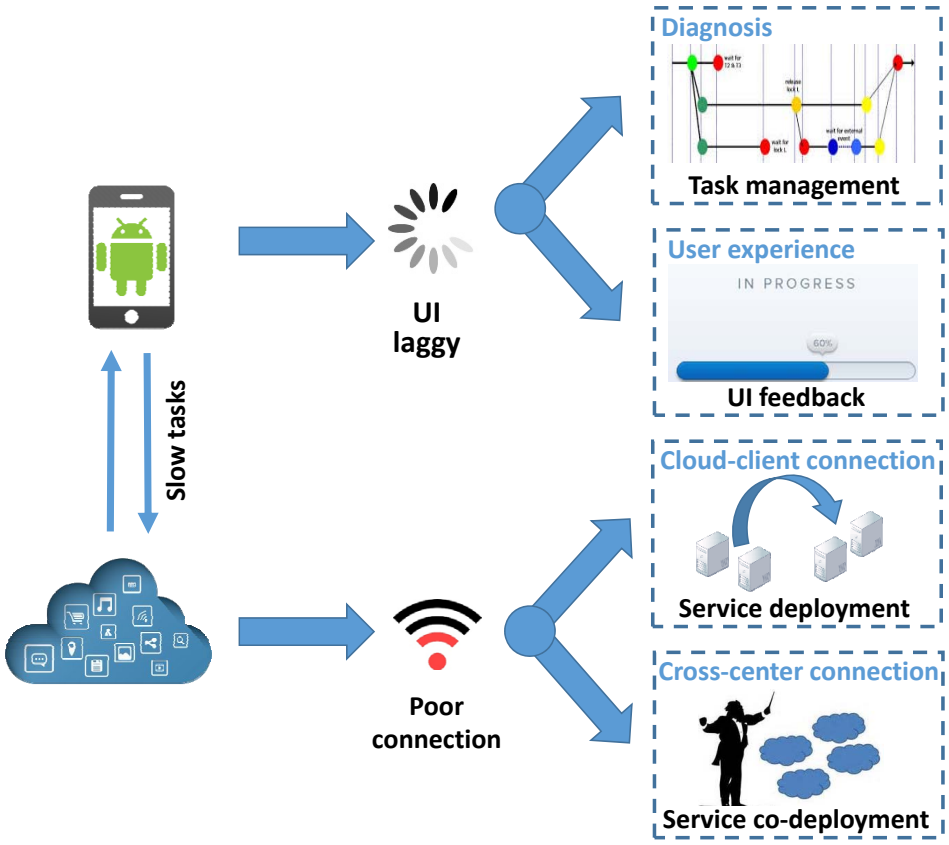


Figure 1.3: An overview of cloud-based mobile app performance tuning

of the processing. For example, the search for tickets from Beijing to Tianjin, sorted by arrival time, takes quite a long time to finish, and it offers no progress updates during the processing. This annoys users a lot. For its cloud service, the cloud server is not configured and deployed optimally for serving a large number of users. It becomes extremely slow at peak times (*e.g.*, Chinese Lunar New Year) when there are too many user requests. All of these issues make the user experience poor and give `12306` an ill reputation for bad performance [11]. This demonstrates that the performance of both mobile apps and cloud services should be improved.

An overview of our proposed procedure for tuning the performance of mobile apps and cloud services is shown in Figure 1.3. We tune the performance of mobile apps and cloud services respectively.

Most of the performance issues of mobile apps are related to the user interface (UI), as the UI lag is directly perceived by users on mobile. On the other hand, many performance problems of cloud services are related to the network, as network delays often degrade performance.

To tackle the UI-related performance issues of mobile apps, two steps are taken. First, we detect and diagnose as many performance issues as possible during pre-release testing to enhance the performance before release. Second, we develop methods for enhancing user tolerance on operation delays, as it is impossible to eliminate all performance issues, due to the limited resource of mobile devices. More specifically, we profile the threads of apps during runtime of testing and detect the slow threads. By localizing the source codes of suspicious threads, we assist developers conducting performance diagnosis. We also study the user tolerance for screen freezes in operations that take a long time to execute. We monitor the screen refreshes to detect those tasks/operations that take a long time to execute without offering feedback, as this annoys users. Then we report the findings to developers to help them design better delay tolerant UI.

To address network delay caused performance issues of cloud services, we reduce the delay in cloud-client communications and cross-(data)center communications. We find that servers in the cloud are distributed in different data centers, which may have variable network distances to client users. Thus, the deployment of servers could affect network delays, which greatly affects the performance of apps. We obtain the network distance between cloud and clients, then model and solve the problem of optimal deployment for cloud servers. There are also many cross-center dependent services (*e.g.*, a service provider may use a hybrid cloud, or a service may request another service like a ticket booking agency service). We measure the cross-center network distance for different servers deployed in diverse data centers (may belong to one or

several clouds), which provide different services. Taking both cross-center and cloud-client communication cost into consideration, we formulate an optimization problem. By solving the problem, we find an optimal deployment of complex services in cloud that reduces the network delay.

1.2 Thesis Contributions

In the thesis, we contribute to the performance tuning of both mobile apps and cloud services. The contributions can be summarized as follows.

1.2.1 Performance Diagnosis for Mobile App

Rapid user interface (UI) responsiveness is a key consideration for Android app developers. To avoid freezing the screen by blocking the UI thread, the requests are usually conducted under the complicated concurrency model of Android. This makes it difficult for the developers to understand and further diagnose an app's performance. This thesis develops `DiagDroid`, a tool specifically designed for Android UI performance diagnosis. The key idea of `DiagDroid` is that UI-triggered asynchronous executions contribute to UI performance, and hence their performance and their runtime dependency should be properly captured to facilitate performance diagnosis. However, there are numerous ways to start these asynchronous executions, posing a great challenge to profiling such executions and their runtime dependency. To this end, we properly abstract five categories of asynchronous executions as the building basis. These executions can then be tracked and profiled based on the specifics of each category with a dynamic instrumentation approach carefully tailored for Android. `DiagDroid` can then profile the asynchronous executions in a task granularity, equipping it with low-overhead and high compatibility merits. The tool is successfully

applied in diagnosing 33 real-world open-source apps; we find 27 performance issues in 14 of the apps. These results show the effectiveness of our tool in Android UI performance diagnosis. We have open-source released `DiagDroid` online.

1.2.2 Delay-Tolerant UI Design for Mobile App

Even finely tuned, due to limited resources, many operations on mobile devices are inevitably slow. Therefore, a good user interface (UI) should be designed to improve user tolerance. UI latency, which is the time between a UI operation and its intended UI update, is a critical consideration for app developers. Current literature still lacks a comprehensive study of how much UI latency a user can tolerate or how to find UI design defects that cause intolerably long UI latency. As a result, bad UI apps are still common in app markets, leading to extensive user complaints. This thesis aims at a better understanding of user tolerance for UI latency, and develops a tool to detect intolerable UI latency in Android apps. To this end, we first design an app to conduct a user survey of app UI latency. A key finding of the survey is that a timely screen update (*e.g.*, a loading animation) is critical to heavy-weighted UI operations (*i.e.*, those that incur a long execution time before the final UI update is available). We design a tool to monitor the UI inputs and updates, and detect apps that do not meet this criterion. The survey and the tool have been released open-sourced online. We also present our experiences on applying the tool on many real-world apps. The results demonstrate the effectiveness of the tool in combating these app UI design defects.

1.2.3 Service Deployment on Cloud

The cloud service has attracted much interest recently from both industry and academia. Optimal deployment of cloud services is critical for providing good performance to attract users. Optimizing

user experience is usually a primary goal of cloud service deployment. However, it is challenging to access the user experience, as mobile users may connect to cloud service from any point. Moreover, there is generally no proactive connection between a user and the machine that will host the service instance. To address this challenge, in this thesis, we first propose a framework to model cloud features and capture user experience. Then, based on the collected user connection information, we formulate the redeployment of service instances as *k-median* and *max k-cover* problems. We propose several approximation algorithms to efficiently solve these problems. Comprehensive experiments are conducted by employing a real-world QoS dataset of service invocation. The experimental results demonstrate the effectiveness of our proposed redeployment approaches.

1.2.4 Multiple Services Deployment on Cloud

Multiple cloud services tend to cooperate with each other to accomplish complicated tasks. For example, a service provider may use a hybrid cloud that provides several services in its private and public clouds, or one service may request another service like a ticket booking agency service. Deploying these services independently may not lead to good overall performance, as there are many interactions among the different services. Determining an optimal co-deployment of multiple services is critical for reducing latency of user requests. If the services are highly related, taking only the distribution of users into consideration is not enough, as the deployment of one service affects others. Thus, we employ cross service information and user locations to build a new model in integer programming formulation. To reduce the computation time of the model, we purpose a sequential model running iteratively to obtain an approximate solution. We collect and publicly release a real-world dataset to promote future research. The dataset involves

307 distributed computers in about 40 countries, and 1881 real-world Internet-based services in about 60 countries. Extensive experiments have been conducted over the dataset to show the effectiveness of our proposed model.

1.3 Thesis Organization

The remainder of this thesis is organized as follows:

- Chapter 2
  In this chapter, we review some background knowledge and related work on performance tuning of mobile applications and cloud services. Firstly, in Section 2.1, we briefly introduce cloud-based mobile apps, mainly focusing on the characteristics of mobile apps and the usage of cloud service. Secondly, in Section 2.2, we review some related work on performance tuning of mobile applications, including performance diagnosis related techniques and delay-tolerant UI design. Thirdly, in Section 2.3, we review some related work on performance tuning of the cloud services, including the background knowledge of the cloud architecture and the cloud service deployment.

- Chapter 3
  In this chapter, we present `DiagDroid`, a tool for performance diagnosis of mobile apps on Android OS. The performance diagnosis is done via anatomizing asynchronous executions. We first introduce the problem in Section 3.1. Then we introduce some Android specifics as the preliminary knowledge in Section 3.2. In Section 3.3, we show two motivating examples of performance issues caused by asynchronous executions. Section 3.4 and Section 3.5 elaborate the design and implementation of `DiagDroid`. We then demo the successful application of our tool on real-world mobile apps and how previously unknown performance issues are found in

Section 3.6. We discuss our tool design and offer some tips for developers to avoid performance issues in Section 3.7. Finally, Section 3.8 summarizes this chapter.

- Chapter 4
  In this chapter, we present `Pretect`, a tool which could detect poor-responsive UI components in Android apps, for delay-tolerant UI design. The detection is done via monitoring the user inputs and UI updates. We first introduce the problem in Section 4.1. Section 4.2 motivates the work by introducing an example of common program defect that causes poor-responsive UI. Section 4.3 presents the results of our study on the relationship between user patience and operations delay levels, which also provides the motivation for developing our new tool. Section 4.4 and Section 4.5 illustrate the framework design and implementation of our tool. Section 4.6 demonstrates the correctness and effectiveness of the tool. Section 4.7 provides some discussions on the tool design considerations. Finally, Section 4.8 summaries this chapter.

- Chapter 5
  In this chapter, we present a cloud service redeployment mechanism which is optimized for performance. We first introduce the problem and requirement of cloud service redeployment in Section 5.1. Then we overview the cloud service hosting mechanism in Section 5.2. Section 5.3 discusses the method of obtaining performance in terms of user experience. We present two different models and objective functions for modeling instance redeployment problem in Section 5.4. Section 5.5 conducts experiments and discusses the experimental results. Finally, the chapter is summarized in Section 5.7.

- Chapter 6
  In this chapter, we present a latency-aware co-deployment mechanism for optimizing performance of cooperative cloud-

based services. First, in Section 6.1, we briefly introduce the problem definition and motivation of cloud-based services co-deployment. Then we illustrate the framework of cloud-based multi-services and the data processing procedure in Section 6.2. Section 6.3 reviews the model of single service deployment. Section 6.4 presents our multi-services co-deployment model. Section 6.5 discusses experimental results. Finally, Section 6.6 summarizes the chapter.

- Chapter 7
  The last chapter concludes this thesis and provides some discussions on possible future work in this research area.

Notice that to make each chapter independent, in some of the chapters we may repeat critical contents like motivations and model definitions.

□ **End of chapter.**

Chapter 2

Background Review

Chapter Outline

We separately tune performance of mobile applications and cloud services (Figure 1.3). We review related work on the architecture of cloud-based mobile applications, mobile applications performance tuning and cloud service performance tuning.

2.1 Mobile Applications and Cloud Services

Building on two mature technologies, cloud-based mobile computing is a promising development. Cloud-based mobile computing has the portability of mobile computing and the scalability of cloud computing. It has attracted widely interests from academia [77] and is growing in share of the mobile app market [12].

Previous studies have surveyed cloud-based mobile applications. Abolfazli *et al.* [25] identify four scenarios in which cloud services increase the computing capabilities of mobile devices: load sharing, remote execution, cyber foraging, and mobile computation augmentation. Mobile computation augmentation includes techniques such as computation offloading, remote data storage, and remote service

requests. Dinh *et al.* [70] list five advantages of using cloud-based mobile computing: 1) extending battery lifetime; 2) improving data storage capacity and processing power; 3) improving reliability; 4) dynamic provisioning; and 5) scalability.

With so many advantages (*e.g.*, augment computational power, battery lifetime of mobile devices), it is unsurprising that a number of studies have proposed offloading computational tasks to clouds. MAUI [66] enables energy-aware offloading to clouds. It provides method-level, semi-automatic offloading of the .net code. Although it improves the performance of some applications, performance enhancement is not its focus. CloneCloud [61] enables unmodified mobile applications running in an application-level virtual machine (VM) to offload part of their execution from mobile devices onto VMs in clouds. CloneCloud automatically partitions applications into pieces for migrating to a VM at runtime. The partitioning is determined offline. However, CloneCloud has limitations with regards to input applications and native resources. It also needs to be updated for every new application version. ThinkAir [113] creates VMs of complete mobile systems on the cloud to address the scalability issues of MAUI, and adopts an online method-level offloading to address the restrictions of CloneCloud. It also supports on-demand resource allocation and parallelism. Kovachev *et al.* [114] design MACS middleware that gives developers better control over offloadable components. The above mentioned approaches work on generic applications. Lee *et al.* [116] propose Outatime which is specially designed for mobile cloud games. For these mobile cloud games, cloud servers execute the games while the mobile clients only transmit UI input events and display the output rendered by the servers.

There are also studies of the fundamental issues in cloud-based mobile computing architecture, such as the tradeoff between performance improvement and energy saving [191].

Our work does not focus on the architecture of cloud-based

mobile applications and its implementation. We do not bind our approach to any specific architecture or implementation. We examine the general performance tuning of mobile apps and cloud services. Therefore, our methods will work to improve performance in a variety of architectures and their implementations.

2.2 Performance Tuning of Mobile Applications

Our tool assists developers to tune the performance of mobile apps during testing. Here, we first review related research on testing mobile apps. Then we review related work on performance diagnosis. As performance diagnosis cannot solve all performance issues, we further investigate UI design techniques to make users more patient with bad performance.

2.2.1 Testing Mobile Applications

Performance diagnosis often requires executing the target app automatically. Script-based testing is widely used (*e.g.*, UIAutomator [177], Monkey runner [138], and Robotium [164]). The record-and-replay approaches (*e.g.*, MobiPlay [156], Reran [88], and SPAG-C [124]) record an event sequence during the manual exercising of an app, and generate replayable scripts. Complementary to these semi-automatic approaches, fuzz testing approaches (*e.g.*, Monkey [179], Dynodroid [132], and VanarSena [160]) generate random input sequences to exercise Android apps. Symbolic execution-based testing approaches (*e.g.*, Mirzaei *et al.* [137], ACTEve [32], Jensen *et al.* [102], EvoDroid [133], A^3E [42], and SIG-Droid [136]) explore the app functions in a systematic way. Model-based testing approaches (*e.g.*, Android Ripper [30], Swift-Hand [59], and PBGT [140]) generate a finite state machine model and event sequences to traverse the model. TestPatch [86] uses the GUI ripping methods for regression tests. Test case selection

techniques (*e.g.*, [94, 176]) can also be adopted to exercise the target apps. These app-exercising tools can work as plugin modules for our tools. In other words, a developer can exploit the merits of a testing tool by simply applying it as a plugin to exercise the target app.

2.2.2 Performance Diagnosis of Mobile Applications

Performance Diagnosis of General Applications

Performance diagnosis has been well studied for many systems. Much work has been conducted on predicting the performance of configurable systems [167, 198]. CARAMEL [146] detects performance problems caused by unnecessary loop executions with static analysis. Yu *et al.* [195] propose a performance analysis approach based on real-world execution traces. However, such traces are usually not available in our problem setting. PPD [189] uses dynamic instrumentation for goal-oriented *known* performance issues searches. Lag Hunting [107] searches for performance bugs in JAVA GUI applications in the wild without addressing concurrency issues. Performance issues in Javascript programs are also well studied [168]. SAHAND [27] visualizes a behavioral model of full-stack JavaScript apps' execution, but it does not take the contentions of asynchronous tasks into consideration. Existing work [29, 103] also considers "thread waiting time" (*i.e.*, when a thread waits for other threads during its execution) as a metric to find performance bottlenecks. In contrast, we focus on queuing time, *i.e.*, the time a task waits before being executed, to model task dependency. More specifically, a lock contention (e.g., db operations) pauses a thread from executing, whereas queuing asynchronous tasks does not. One task waits for another task in the same worker thread to finish, while that thread continuously runs. Therefore, our tool is specially tailored to profile waiting time and task dependency in continuous threads.

Performance Diagnosis of Mobile Applications

Performance is a critical concern for mobile apps [31, 162]. Liu *et al.* [125] show that many performance issues are caused by the blocking operations in the main thread. StrictMode [90] analyzes the main thread to find such blocking operations. Asynchronizer [123] provides an easy way to refactor specific blocking synchronous operations into standard `AsyncTask` implementations. AsyncDroid [122] further refactors `AsyncTask` to `IntentService` to eliminate the memory leakage problems. CLAPP [82] finds potential performance optimizations by analyzing loops. However, such static analysis-based tools cannot capture runtime execution dependency. Banerjee *et al.* [43, 44] design static analysis-driven testing for performance issues caused by anomalous cache behaviors. Tango [91], Outatime [116], and Cedos [139] optimize WiFi offloading mechanisms to maintain a low-latency for apps. SmartIO [143] reduces the app delay by reordering IO operations. Offloading tasks to remote servers can also reduce the delay [188, 187]. Resource leakage is a common source of performance issues and has been widely investigated [194]. These approaches solve specific performance issues. In contrast, our work solves general UI performance issues caused by the runtime dependency of asynchronous executions.

UI Performance Diagnosis of Mobile Applications

User interface (UI) design is one of the key factors in mobile app development [105]. Many studies have focused on methods for diagnosing UI performance. Method tracing [153] is an official tool that is often used to diagnose performance issues, but it has a high overhead and is suitable for diagnosing known issues only. QoE Doctor [57] bases its diagnosis on Android Activity Testing API [26], but can only handle pre-defined operations. Appinsight [161] is a tracing-based diagnosis tool for Windows phone

apps. It traces all of the asynchronous executions from a UI event to its corresponding UI update, and identifies the critical paths that influence the performance. Panappticon [196] adopts a similar approach on Android. However, these approaches typically require framework and kernel recompilation, which limits their compatibility with various devices. Moreover, these studies focus on finding the anomalous task delays; in contrast, our work identifies not only the runtime dependency between tasks, especially those for different UI operations, but also the delays that affect user experience.

2.2.3 Performance and UI Design of Mobile Applications

Performance and General UI Design

Many studies have examined the relationship between user satisfaction and system response time.

Early studies of user-tolerable waiting time focus on general computer applications. Miller [135] notes that different purposes and actions have different acceptable or useful response times. Shneiderman [171] states that response times to simple commands should not exceed 2 seconds. Nielsen [144] suggests that 0.1 second is the limit at which users feel that the system is reacting instantaneously and 1.0 second is the limit for users' flow of thought to remain uninterrupted.

There are also many research investigations on the Web user tolerance. These studies indicate that over time user tolerance for delays on Web pages has decreased. The tolerable delay in early studies was more than 30 to 40 seconds [159, 169]. Hoxmeier *et al.* [97] construct an Internet browser-based application to study how Web delay affects user satisfaction, and find a 12 seconds threshold. Galletta *et al.* [83] vary the website delay time from 0 to 12 seconds. They examine the performance, attitudes, and behavioral intentions of the subjects to understand user tolerance for website delays. The results suggest that users are tolerant of delays of about 4 seconds.

Nah [141] finds that most users are willing to wait for only about 2 seconds for simple information retrieval tasks on the Web.

The above-mentioned research focuses on the absolute value of delays and does not consider how feedback affects user satisfaction with a delay. Visual and non-visual feedback is an important design element in delay-tolerable UI. Duis *et al.* [71] point out that a system should let a user know immediately that her input has been received, but the authors offer these suggestions without experimental study. Johnson [104] also provides many insights into system responsiveness and user satisfaction. He suggests that showing a progress bar for long-term operations is much better than showing nothing or only a busy bar.

Performance and Touch Screen UI Design

Touch screen user interfaces have attracted research interests for some time. Findlater *et al.* [78] compare adaptive interfaces for small and desktop-sized screens to study the impact of screen size on user satisfaction. They show that high accuracy adaptive menus are highly beneficial for small screen displays. Forlines *et al.* [79] study the different characteristics of direct touch screen input and mouse input. They show different input methods have different benefits.

Tolerance for delays on mobile devices has also been studied. Oulasvirta *et al.* [149] reveal that user attention spans vary from 4 seconds to 16 seconds on mobile devices once the page loading has started. Anderson [33] suggests that user tolerance for touch screen latency for common tasks is below 580 ms. Jota *et al.* [106] and their follow-up work [68] show that 1) the initial delay feedback the users can perceive ranges from 20 to 10 ms, and 2) the user detectable threshold of direct and indirect operations for different tasks (tapping and dragging) range from 11 ms to 96 ms. Ng *et al.* [142] show that user notice improvements in speed at well below 10 ms. These results have implications for touch screen UI design; our research is designed to detect violations of the tolerance

thresholds.

Visual and non-visual feedback is widely used in the design of delay tolerable UI. Roto *et al.* [166] show that multimodal feedback for delays of more than 4 seconds is required. Lee *et al.* [117] note that the absence of feedback affects user performance (*e.g.*, typing on flat piezo-electric buttons that have no tactile feedback significantly reduces expert typists performance). Kristoffersen *et al.* [115] suggest using audio feedback on mobiles. Ng *et al.* [142] recommend providing low-fidelity visual feedback immediately. Poupyrev *et al.* [151] find that tactile feedback is most effective when the GUI widgets need to be held down or dragged on the screen. Ripples [190] provide a special system on top of the screen that can give feedback about the successes and errors of the touch interactions. In contrast, we do not modify the current mobile UI framework. Our work focuses on detecting UI elements with long delays or with no feedback that may leave users uncertain of the status of their commands. This information can be used by developers to avoid such problems.

2.3 Performance Tuning of Cloud Server

2.3.1 Cloud Architecture and Datacenter Characteristics

Cloud Architecture

Many previous studies have examined the cloud framework. There are several good survey work in this area [39, 65, 93, 155, 184]. Luis *et al.* [182] give a clear picture of cloud computing, and provide definitions of key components of the cloud.

At first glance, cloud computing may look very similar to grid computing [80], but they are not identical. Foster, who posted the grid framework, compares grid computing to cloud computing in detail [81]. They can both be used to do parallel computations and thus reduce the cost of computing. They both manage large

numbers of facilities and offer users flexibility. The difference is that grid computing uses distributed resources and an open standard, whereas cloud computing is sponsored by a single company. The business models of these two are different and so their target users are different.

There are three kinds of cloud infrastructures, which correspond to three layers of services: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). IaaS abstracts the user from details of infrastructure (*e.g.*, physical computing resources, location) to virtual machines. Users are usually required to install OS images and their application by themselves. PaaS provides typical development platform for developers. The platform includes operating system, programming-language execution environment, database and web server. For SaaS, the provider install and operate application software in the cloud and users access the software from the clients. Three real world examples are Amazon Elastic Compute Cloud (EC2) [5, 87] for Iaas, Microsoft's Windows Azure Platform [18] for Paas and Google App Engine [14] for SaaS. Lenk *et al.* [119] give an overview of the features of existing cloud providers. Lin *et al.* [121] define high level IT as a service (ITaaS) that can be viewed from both technology and business model perspectives.

Currently, there is limited cooperation between different cloud providers. Rochwerger *et al.* [165] identify the shortcomings of contemporary clouds as due to the inherently limited scalability of single-provider clouds, lack of interoperability among cloud providers, and no built-in Business Service Management (BMS) support. They present an architecture for a federation of disparate data centers based on the principles of BMS.

Apart from commercial services, there are open source examples of cloud services. Eucalyptus [147] is an open-source software framework for cloud computing. It implements the IaaS principle.

Brandic [49] offers a structure for self-manageable cloud ser-

vices. This structure uses the concept of autonomic systems. The procedure has four stages: monitoring, analyzing, planning, and execution. He uses this life cycle in each work period (meta negotiation, negotiation, service execution, and post-execution) of a cloud service to keep the architecture self-manageable. Our work improves his analysis procedure.

Cloud computing is a commercial product. Buyya *et al.* [51] examine cloud issues from a market-orientation. They talk about the commercial and technological implementations of cloud computing for industry. Our research considers the effect of budget limitations and thus also takes the commercial factors into consideration. Subsequent research by Buyya *et al.* [52] discusses the impact of cloud computing on the provision of computing power.

As claimed by Walker [185], clouds (*e.g.*, Amazon EC2) perform worse in high performance computing contexts than current HPC clusters. However, there are also some variants that suit high performance cloud computing. These variants include GPUs combined with CPUs to form a mixed cloud environment. GPUs provide more computation power than CPUs, so including GPUs in a cloud can improve the computing capability greatly. Amazon claims to provide clusters of GPU based instances for high performance computing [9]. Farivar *et al.* [75] design an architecture, MITHRA, that leverages NVIDIA CUDA technology [20] along with Apache Hadoop [6] to produce scalable performance gains using the MapReduce [67] programming model. Barak *et al.* [45] present a packet for running OpenMP, C++, and unmodified OpenCL applications on clusters with many GPU devices. Their experiment demonstrates how powerful GPU clusters can be used to speedup applications. Our model could improve deployment of both GPU clusters and CPU clusters, as we do not differentiate them.

Data Center Characteristics

We have a special interest in data center traffic characteristics. Our deployment methods are based on the analysis of data center traffics; therefore, understanding traffic characteristics is very important. Luckily, there are many studies of data center characteristics.

Wang *et al.* [186] evaluate the performance of Amazon EC2. They measure: 1) processor sharing; 2) packet round-trip delay; 3) TCP/UDP throughput; and 4) packet loss. The TCP/UDP throughput referred to, like other network properties, are measures of network performance inside the cloud; that is, they measure a pair of instances in the cloud, but do not consider an outside visit. We still need to understand the complete traffic pattern for users using service in a cloud.

Ersoz *et al.* [73] first characterized network traffic in a cluster-based multi-tier data center. They find that 1) in most cases, the request inter-arrival rates follow log-normal distribution, and self-similarity exists when the data center is heavily loaded; 2) message sizes can be modeled by the log-normal distribution; and 3) service times fit reasonably well with the Pareto distribution and show heavy tailed behavior at heavy loads.

Benson *et al.* [48] present an empirical study of end-to-end traffic patterns in data center networks. The same group [47] has conducted experiments in ten data centers belonging to three different categories, including university, enterprise campus, and cloud data centers. They collect and analyze SNMP statistics, topology, and packet-level traces, to provide insights into different data center characteristics. Srikanth *et al.* [109] instrument the servers to collect socket-level logs. They obtain and report detailed views of traffic and congestion conditions and patterns.

Qian *et al.* [154] propose a hierarchical modeling approach that can easily combine all of the components of a cloud environment. Their model is a very useful analytical tool for online service pro-

viders who are evaluating cloud computing providers and designing redirection strategies.

2.3.2 Virtual Machine Live Migration and Management

The virtual machine live migration problem has been well studied. The Xen migration mechanism is first proposed by Clark *et al.* [63]. They could transfer an entire machine in hundreds of milliseconds. Hines *et al.* [96] propose a post-copy based approach that defers the transfer of the contents of a machine's memory until after its processor state has been sent to the target host. It ensures each memory page is transferred at most once, thus avoiding the duplicate transmission overhead of pre-copy. The Remote Direct Memory Access (RDMA) technique is used to further reduce the migration time and application downtime [99].

The virtual machine migration technique has been extended to management problems. Van *et al.* [180] study the problem of deciding what types of virtual machines to use for multiple applications. They consider latency and the service level agreement with computation cost. Stage *et al.* [174] consider how much bandwidth is consumed during the migration. They propose a network topology-aware scheduling model to minimize the number of hosts. Andreolini *et al.* [34] present a management algorithm of VM placement that improves performance and resource utilization by considering the load profile of the hosts and the load trend behavior of the guest.

Although all of the virtual machine management algorithms consider the load balance across a set of hosts or reduce the number of required hosts, user experience is seldom considered important factor. These models omit the network cost in their model, whereas our model emphasizes network latency and makes the assumption that the computation cost remains the same, regardless of the host locations.

2.3.3 Cloud-based Service Deployment and Algorithms

Web Server Placement

There are solutions that can accelerate Web applications by taking advantage of extra cache servers such as AICache [3] and Akamai application performance solutions [7]. However, these commercial services increase costs. As their technique is merely to use extra servers, we can generalize the approach into our framework by increasing the number of servers.

Previous research has examined service deployment in gird computing environments. Kecskemeti *et al.* [111] design an automatic service deployment method for a grid that creates virtual appliances for grid services, deploys service from a repository, and influences the service schedules by altering execution planning services, candidate set generators, or information systems. Unlike a cloud, a grid is not centralized which limits application of these methods on cloud.

Burg *et al.* [181] focus on software deployment, especially selecting which machine to load and to run the source codes on according to the QoS. In contrast, our approach is on the application level. Qiu *et al.* [157] study the placement of Web server replicas using several placement algorithms. They evaluate the algorithm by comparing the output to the result of a super-optimal algorithm based on Lagrangian relaxation which may not be feasible in practice. Zhang *et al.* [197] study the placement problem in shared service hosting infrastructures. Instead of modeling the placement problem as a *k-median* problem, they consider it similar to the capacitated facility location problem. They define their own penalty cost instead of using response time directly. However, these studies are not tailored for cloud computing.

Cloud-Based Service Deployment

The Web server placement problem has been widely studied.

Web server replicas has been studied by Qiu *et al.* [157]. They

use the result of the *k-median* model as a super-optimal result. They use different algorithms to give an approximate solution of the model. Zhang *et al.* [197] study service placement in shared service hosting infrastructures. They formulate a model similar to the general capacitated facility location model. They do not use response time directly, but define a new penalty cost.

However, these studies are not tailored for cloud computing. In our work, the properties of cloud computing are considered. The user experience is highlighted and a general framework of service deployment in a cloud environment is proposed. Building on previous studies of single service deployment, we propose a model for the co-deployment of multiple services.

k-Median Problem

K-median problem is the problem of finding k medians such that the sum of distances of all the points to their nearest medians is minimized. It is a common model for facility location problem. More discussion of this problem can be found in Section 5.4.1. Jain and Vazirani [101] provide a 6-approximation for the *k-median* problem. Their algorithm uses the Lagrangian relaxation technique. Their main contribution is that they use the algorithm for the facility location problem as a subroutine to solve the *k-median* problem. They prove that a Lagrangian multiplier preserving α-approximation algorithm for the facility location problem gives rise to a 2α-approximation algorithm for the *k-median* problem. Based on this approach, many improvements have been achieved. Charikar and Guha [55] use a similar idea and achieve a 4-approximation algorithm for the *k-median* problem. Jain *et al.* [100] obtain a new greedy approach for the facility location problems. By improving the subroutine, they also get a 4-approximation using the same procedure as in the previous algorithm.

Arya *et al.* [40] first analyze the local search algorithm for *k-median* and provide a bounded performance guarantee. Their

analysis supports our algorithm, used in Section 5.4.1, which can approximate the optimal solution to the ratio $3 + \varepsilon$, the best result currently known.

Max k-Cover Problem

Max k-cover is the problem of selecting at most k among n sets - which have many elements in common - that could cover maximum number of elements. More discussions could be found in Section 5.4.2. The *Max k-cover* problem is related to the set cover problem. Many algorithms have been proposed (*e.g.*, [62, 76, 127]) to solve this problem. The greedy algorithm [76] is one of the best polynomial time algorithms for this problem; it gives a $(1 - 1/e)$-approximation.

There is also an algorithm for solving the online set cover [28] problem, which deals with the given elements one-by-one. This algorithm can fit the needs of users executing services in the cloud.

k-Median Model and Multi-commodity Facility Location

A series of facility location problems have been well studied in the supply chain management field, as reviewed by Melo *et al.* [134]. One discrete variation of the facility location problem is the *k-median* (also *p-median*) model [130], which has been studied in the context of Web service deployment [157]. The original *k-median* model considers the facility location problem for only one commodity. Our model is closer to the multi-commodity facility location problem.

Pirkul *et al.* [150] propose the PLANWAR model, which is an established formulation of the multi-commodity, multi-plant, capacitated facility location problem. Shen [170] modifies the cost function and obtains a new model. Cao *et al.* [54] propose a variation of the *k-median* model for the problem. However, these models do not consider cross-plant transportation; instead the commodities are

regarded as rather independent.

Thanh *et al.* [178] propose a very complex dynamic model with about 40 constraints. The model considers a multi-echelon, multi-commodity production-distribution network with deterministic demands. They make the assumption that the production process can be divided into several steps and can be shared between several plants. The production process does not rely on other productions or sub-routines. The relation between two commodities is that they can be manufactured/stored in one facility simultaneously. The multi-echelon is divided according to the life cycle of a commodity but not a cross-commodity, as in our model.

□ **End of chapter.**

Chapter 3

Android Performance Diagnosis via Anatomizing Asynchronous Executions

Android Performance Diagnosis

This chapter presents `DiagDroid`, a tool specifically designed for Android UI performance diagnosis. The key notion is that the UI-triggered asynchronous executions contribute to the UI performance, and hence their performance and runtime dependency should be properly captured. We list the points of this chapter as:

- Group tremendous ways to start the asynchronous executions into five categories; Track and profile them with low-overhead and high compatibility.
- Implement and open-source release the tool.
- Apply `DiagDroid` in diagnosing real-world open-source apps which we are unfamiliar with their implementations; Successfully locate and diagnose tens of previously unknown performance issues.

3.1 Motivation and Problem Definition

As daily-use personal devices, mobiles are required to provide quick response to the user interface (UI). UI performance of a mobile app is a critical factor to its user experience, and hence becomes a major concern to developers [125, 126]. Many recent research efforts have therefore been put on addressing the performance issues of Android apps (*e.g.*, Asynchronizer [123], Panappticon [196]). However, poor UI performance of Android apps remains a widely-complaint type of issues among users [125, 126]. App developers are still lacking a handy tool to help combat performance issues.

Android provides a non-blocking paradigm to process UI events (*i.e.*, user inputs) for its apps. The UI *main* thread dispatches valid UI events to their corresponding UI event procedures (*i.e.*, the UI event-handling logic). A UI event procedure generally runs in an asynchronous manner, so that the main thread can handle other UI events simultaneously. After the asynchronous part is done, the UI can be updated with a call-back mechanism. This paradigm will lead to complicated concurrent executions. Asynchronous execution processes may bear implicit dependency during their runtime. For example, two may be scheduled to run in the same thread by Android, and one may consequently wait for the other to complete. Such unexpected waiting may result in a longer delay for a UI procedure, leading to UI performance issues. However, it is hard to predict such runtime dependency during the coding phase due to the complications of Android's asynchronous execution mechanisms [152]. Performance issues are hence inevitable.

Concurrency is a notorious source of bugs [129]. Current tools for diagnosing Android UI performance issues generally consider either the synchronous part of the UI event procedure [90], or the execution process of one UI event procedure *per se* [196]. They do

not focus on the dependency of multiple asynchronous execution processes. Hence, they are still not enough to cope with the UI performance issues largely caused by such runtime dependency.

Long-term testing is a well-known, viable means to trigger bugs caused by concurrency [118]. Unfortunately, we lack an automatic mechanism to check whether there exists a performance issue in the long-term testing. Manual inspection of the tremendous traces produced by current method tracing tools (*e.g.*, Traceview [153]) is extremely labor-intensive, if not infeasible, not to mention their huge overhead.

We find that unlike general concurrent programs [50, 53, 98, 108], an Android UI event procedure can be anatomized into a set of trackable *tasks*, which can then be properly profiled so as to facilitate the detection and localization of performance issues. Specifically, although Android supports tremendous ways to schedule asynchronous executions, we conclude that they can actually be abstracted as five categories. Each can be tracked and profiled in task granularity according to the specifics of each category. UI performance can hence be modeled by the performance of the tasks. We further tackle the complication of runtime dependency via examining the dependency of tasks, which can be solved by checking whether the tasks request the same execution unit (*e.g.*, a thread pool). Via modeling task performance by not only its execution time, but the time when it waits for execution (*i.e.*, the time between when it is scheduled and when it starts execution), we can model how a task is influenced by the others. Thus, performance issues due to asynchronous executions can be properly captured.

Hence, this chapter proposes `DiagDroid` (Performance `Diagnosis` for `Android`), a novel tool to exercise, profile, and analyze the UI performance of an Android app without modifying its codes. First, via a light-weight static analysis of the target app, `DiagDroid` obtains the necessary app information for its profiling mechanism. Then it employs a plugin testing approach

(*e.g.*, Monkey [179], a random testing approach) to exercise the original app. The required runtime data are then captured during the testing run via its profiler. The data are then processed offline to generate a human readable report. The report can unveil potential performance bugs to developers and direct them to suspicious locations in the source codes. Human efforts can greatly be reduced in diagnosing UI performance issues. Finally, `DiagDroid` solves the compatibility and efficiency challenges generally faced by the dynamic analysis tools by slightly instrumenting only the general Android framework invocations with a dynamic instrumentation approach. Hence, it can be applied to most off-the-shelf mobile models and apps.

We have implemented and open-source released `DiagDroid` with a tutorial [69]. We show that it is easy to apply DiagDroid to real-world practical apps with light configurations. In the 33 open-source real-world apps we study, 27 performance defects in 14 apps are found, and we receive positive feedbacks from their developers. These defects are all caused by the complicated dependency of asynchronous executions, which can hardly be located by current UI performance diagnosis practice. This indicates the effectiveness of `DiagDroid`.

3.2 Android Application Specifics

3.2.1 UI Event Processing

Designed mainly for user-centric usage patterns, Android apps are typically UI oriented: An app will iteratively process user inputs, and accordingly update the display to show the intended contents. The `main` thread of an app is the sole thread that handles the UI-related operations [152], such as processing user inputs and displaying UI components (*e.g.*, buttons and images). When a valid user input (*i.e.*, a UI event) comes, the `main` thread can invoke its

```
public class MyActivity extends Activity {
    private class RetrieveDataTask extends AsyncTask<String, Void, String> {
        ...
        @Override
        // doInBackground will be executed asynchronously in a worker thread
        protected String doInBackground(String... urls) {
            try {
                ...
                // Start downloading task
                InputStream is = httpResponse.getEntity().getContent();
                while ((length = is.read(buf)) != -1) {
                    ...
                }
            } catch ... //Error handling codes
            return content;
        }
        @Override
        // onPostExecute will be invoked in the main thread after doInBackground
        // completes, which shows the downloaded content in the UI.
        protected void onPostExecute(String content){
            this.textView.setText(content);
        }
    }
    ...
    private class MyOnClickListener implements OnClickListener{
        @Override
        protected void onClick(View v){
            RetrieveDataTask task1 = new RetrieveDataTask(textView1);
            // call execute method according to the example of official document
            task1.execute(url1);
        }
    }
}
```

Figure 3.1: An AsyncTask example

corresponding *UI event procedure*, *i.e.*, the codes that handle the UI event.

Some UI event procedures may be time-consuming, for example, one to download a file from the Internet. To avoid blocking the `main` thread, the UI event procedures conduct heavy-weighted work in an asynchronous manner so that the `main` thread can handle other UI inputs simultaneously [152]. After such *asynchronous executions* are done, the UI can be updated in the `main` thread with a call-back mechanism.

Figure 3.1 shows the codes of an `Activity` (*i.e.*, a window container for the UI components to be displayed). It retrieves data from the Internet and displays the data in a `TextView` (*i.e.*, a UI component to display text) after a button is touched. The Internet access is done asynchronously in another thread while the

Table 3.1: Asynchronous execution examples

Class	Code Segment

Thread	`//Create a new thread and download in that thread`
	`Thread thread = new Thread(new DownloadRunnable(url));`
	`thread.start();`

Thread Pool Executor	`//Download in one thread of a thread pool with capacity 10`
	`ExecutorService threadPool = Executors.newFixedThreadPool(10);`
	`threadPool.execute(new DownloadRunnable(url));`

Handler	`//Download in a HandlerThread by posting a task on the attached handler`
	`HandlerThread handlerThread = new HandlerThread("DownloadHanderThread");`
	`handlerThread.start();`
	`Handler handler = new Handler(handlerThread.getLooper());`
	`handler.post(new DownloadRunnable(url));`

Intent Service	`//Download in a user-defined Service`
	`Intent downloadIntent = new Intent(this, DownloadService.class);`
	`downloadIntent.putExtra(DownloadService.URLKEY, url);`
	`startService(downloadIntent);`

Download Manager	`//Use standard DownloadManager Service, utilizing ThreadPoolExecutor implictly`
	`DownloadManager dm = (DownloadManager) getSystemService (DOWNLOAD_SERVICE);`
	`DownloadManager.Request req = new DownloadManager.Request(Uri.parse(url));`
	`dm.enqueue(req)`

`TextView` update is done in the `main` thread. More specifically, the `RetrieveDataTask` extends the `AsyncTask` class. It overrides the `doInBackground` method to allow accessing the Internet asynchronously in a worker thread. Its `onPostExecute` method is a call-back mechanism to allow the corresponding update of the `TextView` object in the `main` thread. These codes are abstracted from an open-source project, namely, RestC [24]. It shows a common coding practice of processing UI-operations.

3.2.2 Asynchronous Executions

Android provides high flexibility to implement asynchronous executions. There are tremendous ways for an app to start asynchronous executions. Examples include using `AsyncTask`, `ThreadPoolExecutor`, and `IntentService`. Actually we can find hundreds of classes or methods in the Android framework that can start asynchronous executions, by inspecting the Android source codes. The implicit ways to start asynchronous executions include, for example, those via the customized classes that override

the Android framework classes such as `AsyncTask` or `HandlerThread`.

Table 3.1 shows various ways of conducting asynchronous executions. For a simple task to download Internet contents, we could name at least 6 ways (including examples shown in Figure 3.1 and Table 3.1). Choosing which way generally depends on the developer's own preference.

No matter how an asynchronous execution starts, it is executed by the operating system (OS) via the thread mechanism so as to implement concurrency. However, Android may start a new thread or reuse a running thread for the asynchronous execution. As a result, different asynchronous executions may share the same thread and run sequentially. In other words, they may compete for the same execution unit. Unfortunately, it is hard for the developer to be aware of such dependency of asynchronous executions: she may not know exactly how the asynchronous executions run.

The complex ways of starting asynchronous executions, together with their complicated runtime dependency, make it difficult for the developers to comprehend the performance of UI event procedures they write. Performance issues are hence hard to be eliminated without a proper tool. Next, we will show two representative performance issues.

3.3 Motivating Examples

Since there is only one sole `main` UI thread for each app, UI events are handled one by one by the thread. Hence, in order not to block the `main` thread, the official Android development guide [152] suggests that asynchronous executions should be conducted for time-consuming tasks if an event handler includes such tasks.

Actually app developers may overlook such suggestions occasionally and write time-consuming codes in the synchronous part of the UI event handler (*i.e.*, that executes in the `main` thread).

This is long been known as a notorious cause of performance problems [152], including frequent ANR (Application Not Responding) reports which indicate the consecutive UI events are blocked for more than 5 seconds [112]. Numerous tools have been designed to find such bugs. Examples include the official Google developer tool StrictMode [90] and Asynchronizer [123], which typically address this issue via static analysis.

However, addressing the performance problems solely in the synchronous executions is still far from enough. When a user regards that she is suffering from slow and laggy UI, she is actually experiencing a long period of time between her UI operation and its corresponding intended display update. Even when asynchronous executions are introduced and the synchronous part completes its execution quickly, she may still feel that the UI is laggy if the asynchronous executions are slow and consequently cause the laggy display update of the intended contents.

The complex ways of starting asynchronous executions of Android may introduce various tricky performance problems. Next we will show even if simple, seemingly-correct codes may contain performance problems.

We adopt `AsyncTask` as an example. `AsyncTask` is a simple, handy class that allows developers to conveniently start self-defined asynchronous executions and notify the `main` thread to update the UI [41].

We show two typical cases where performance problems are introduced. The first is caused by unexpected sequentialized asynchronous executions, while the second by not or not properly canceling expired asynchronous executions.

```
private class MyOnClickListener implements OnClickListener {
    @Override
    protected void onClick(View v){
        RetrieveDataTask task1, task2, task3;
        task1 = new RetrieveDataTask(textView1);
        task2 = new RetrieveDataTask(textView2);
        task3 = new RetrieveDataTask(textView3);
        // the frist trial on executing tasks in parallel
        task1.execute(url1);
        task2.execute(url2);
        task3.execute(url3);
    }
}
```

Figure 3.2: Example codes which may cause potential performance problems

3.3.1 Sequential Running of Multiple Asynchronous Executions

Let us suppose that an event handler will show three text views in an `Activity`. The content of each text view should be loaded from the Internet. Since Internet access is slow, the developer may resort to asynchronous executions to download the contents, and expect to download the three in parallel. Her codes are illustrated in Figure 3.2, where the class `RetrieveDataTask` is defined in our previous example shown in Figure 3.1.

Invoking the `execute` method is shown as a usage example by the official guide [41], which is commonly used to start the asynchronous executions defined by the `AsyncTask` extensions (*e.g.*, the `RetrieveDataTask` class in this example). The developer may consider that every `RetrieveDataTasks` would be executed in separated threads, and hence invoking three `execute` methods in sequential will make them run in parallel.

However, even such simple codes contain subtle causes of potential performance problems. Since the `execute` method of `AsyncTask` cannot be overridden, all the execution methods in this example will actually call the `execute` method implemented in their super class (*i.e.*, `AsyncTask`). In the recent versions of

```
// example code of executing tasks in parallel, notice the change
// of API in Android 3.0
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    task1.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, url1);
} else {
    task1.execute(url1);
}
```

Figure 3.3: Correct codes to execute an `Asynctask`

Android framework, invoking the `execute` method will insert the corresponding task into a global queue and all tasks will be executed in sequential in one sole thread instead of in parallel in multiple threads. This inevitably incurs more time to complete the download tasks and update the UI accordingly. As a result, the user will experience laggy UI.

It is worth noting that such sequential execution mechanism is introduced recently in Android systems with version numbers larger than 3.0. In previous versions, the codes will, in contrast, execute in parallel as expected. It is quite possible for the developers to neglect such changes and introduce potential performance problems. Android has a quick evolution on the framework. Though providing good backward compatibility in APIs, the implementations of these APIs are not guaranteed to be stable. However, current tools like StrictMode [90] and Asynchronizer [123] only locate problems in the synchronous executions. As a result, such performance problems in the asynchronous executions caused by unexpected sequentialized asynchronous executions cannot be located conveniently by current tools. Nonetheless, such code defects are quite common among developers. For example, the Facebook Widget [148] project contains a similar issue in its class `StreamFragment.StatusAdapter`.

Note that the execution time values of the tasks *per se* between the sequential case and the parallel case may not be quite different. However, for the sequential case, a task may be queuing for execution for a longer time after it is scheduled. If a tool can capture such a queuing time, it can greatly facilitate performance diagnosis.

```
public class MyActivity extends Activity {
    private class RetrieveDataTask extends AsyncTask<String, Void, String> {
        ...
        @Override
        protected String doInBackground(String... urls) {
            ...
            // monitor the cancelation, stop as soon as the task is cancelled
            while (!isCancelled() && (length = is.read(buf)) != -1) {
            ...
        }
    }

    private RetrieveDataTask retrieveDataTask1, retrieveDataTask2,
        retrieveDataTask3;
    ...

    @Override
    public void onStop() {
        // should cancel tasks explicitly
        if(retrieveDataTask1 != null) retrieveDataTask1.cancel(true);
        if(retrieveDataTask2 != null) retrieveDataTask2.cancel(true);
        if(retrieveDataTask3 != null) retrieveDataTask3.cancel(true);
        super.onStop();
    }
}
```

Figure 3.4: An example of cancelling AsyncTask

`DiagDroid` is a tool that can well capture such queuing time. As a consequence, it is able to detect and locate such performance problem.

Finally, such a code defect can be resolved by invoking the `executeOnExecutor` method instead of the `execute` method by assigning a new task queue for each download task. We show the modifications of the first `execute` statement in Figure 3.2 as an example in Figure 3.3.

3.3.2 Not Canceling Unnecessary Asynchronous Executions

Let us again consider the example codes shown in Figure 3.1. After the codes are modified as shown in Figure 3.3 so that the three tasks can execute in parallel, the codes still may cause performance problems. Suppose the app allows its user to switch the activities during the three views being loaded in the current activity, for instance, with a right-to-left sliding operation. Such a mechanism is commonly used in Android apps, which facilitates users to quickly

locate the activity of interest. An example is that an email client may allow the user to switch from the list-email activity to the read-email activity, even if the email list is not completely shown in the list-email activity.

When the user switches to another activity, the three tasks which are loading contents from the Internet are no longer required since their associated views and activity are invisible. It is therefore unnecessary to continue the three asynchronous tasks in downloading the Internet contents. But the example codes do not explicitly cancel the asynchronous tasks. The tasks will run until the entire contents are downloaded. Such unnecessary asynchronous executions may incur resource races (*e.g.*, occupying the Internet bandwidth), and therefore deteriorate the performance of other asynchronous executions. Sometimes, they may even block other asynchronous executions: For example, they can occupy the working threads in a thread pool and cause other tasks waiting for free threads.

The codes can be further improved, as shown in Figure 3.4. Note that such code defects are very common to Android apps. We will show in our experimental study that many developers of the popular Android apps (*e.g.*, rtweel [158] and BeerMusicDroid [183]) have made such mistakes.

Note that a canceling operation is typically required when the task is time-consuming and should be terminated in the middle. If not properly canceling such a task, its execution latency will be relatively large. Hence, the key to detect and locate such a performance issue is to find out the execution latency of the asynchronous tasks. `DiagDroid` can well profile the task with its execution latency, which can greatly facilitate the diagnosis of such a performance problem.

The above mentioned performance issues shown in Section 3.3.1 and 3.3.2 actually can hardly be tackled by current tools. Tools like StrictMode [90] and Asynchronizer [123] consider only the synchronous part of a UI event procedure, which cannot locate the

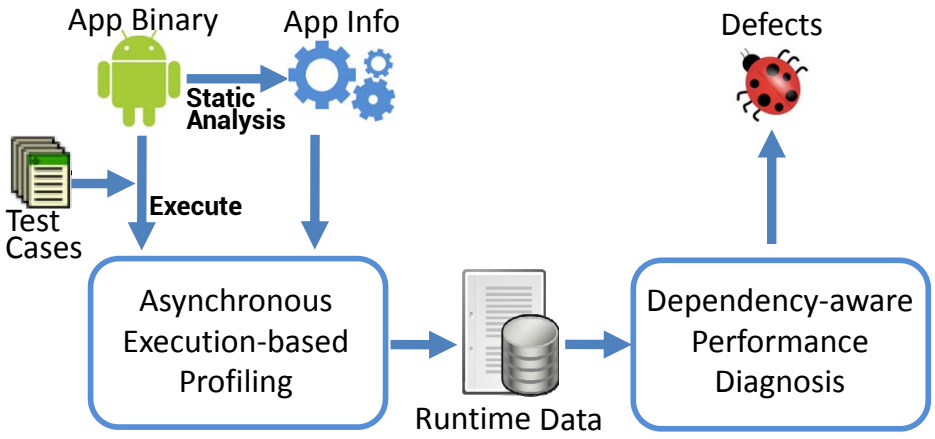


Figure 3.5: Overview of DiagDroid Framework

issues caused by the asynchronous executions. Other tools like Panappticon [196], Method Tracing [153] can track such executions. However, they largely do not focus on the runtime dependency of asynchronous executions. It is hard to find out such dependency via examining the tremendous traces produced by these tools. Hence, they are still not enough to cope with UI performance issues. Fixing this gap is the purpose of DiagDroid.

3.4 UI Performance Diagnosis

We notice that the key to pinpoint the above performance issue is to know not only the execution time of an AsyncTask, but also the time between when it is scheduled and when it starts to execute (*i.e.*, the queuing time), as well as the other AsyncTasks that are in the same thread pool. Specifically, an unexpectedly long queuing time of an AsyncTask can indicate a performance issue. We can instantly know there are too many AsyncTasks in the thread pool. By examining which AsyncTasks are in the pool together with the pool capacity, we can easily locate these performance issues.

The above notion can also be applied to other mechanisms that start asynchronous executions. This is the basis of the DiagDroid

design, which we overview in Figure 3.5. DiagDroid anatomizes the Android UI event procedures into a set of tasks and then quantize them so that data analysis can be conducted towards automating performance diagnosis.

Specifically, as shown in Figure 3.5, DiagDroid first performs a light static analysis of the target app and obtains some required information to assist runtime profiling. It then exercises the original app via a plugin testing approach which can involve random test cases (*e.g.*, Monkey [179]) or user-defined ones (*e.g.*, Robotium [164], UIAutomator [177]). During the testing run, the profiler can track asynchronous executions, so as to anatomize the UI event procedures into a set of tasks. The performance of the tasks, together with their runtime dependency, can then be captured. Based on the profiling data, DiagDroid detects performance issues and analyzes their causes. A report can finally be generated with an aim to direct the debugging process.

To this end, we need to address several critical considerations including the profiling granularity and how to do the profiling and diagnosing. Next, we will discuss the profiling granularity of DiagDroid, and the required runtime data for modeling asynchronous executions and their runtime dependency (in Section 3.4.1). Then, we illustrate how such data can facilitate UI performance diagnosis (in Section 3.4.2).

3.4.1 Modeling Tasks and Their Dependency

As shown in Section 3.3, subtle runtime dependency of asynchronous tasks can result in tricky performance issues. Analyzing such dependency is a key concern to DiagDroid. We analyze the app runtime in *task* level, defined as follows.

Definition 3.1. *An **asynchronous task** (or task) is a segment of codes that run in sequence in a single thread. It defines a developer-intended asynchronous execution process.*

`DiagDroid` profiles the app runtime in task granularity. The reasons are as follows. First, it is good enough for performance diagnosis to profile in such a granularity. A task is a short segment of codes that can also be well understood by its developer. If the developer can know which task is anomalous, she can instantly reason its cause by inspecting the task-related codes. Second, such a granularity will not incur too much profiling overhead, compared with the finer granularity (*e.g.*, in method level or in line level). Most importantly, profiling app runtime in task granularity can well capture the runtime dependency of two asynchronous tasks. As a consequence, UI performance issues caused by such dependency can be easily detected and located.

The task performance naturally reflects the performance of the entire UI event procedure: a slow task may result in a slow UI event procedure, leading to a laggy UI. Next, we discuss how to model task performance. We are aware of the fact that it is possible for a task to queue in an execution unit before it is executed. As a result, to model its performance, we should consider not only its execution time, but also its queuing time in the unit.

Definition 3.2. *The **queuing time** of a task is the interval between when the task is scheduled (*e.g.*, when* `execute` *method is called to start an* `AsyncTask`*) and when it starts to execute.*

We propose to use both the queuing time and the execution time to model the task performance. Note that this is generally different from the current diagnosis practice with tools like TraceView and dumpsys [153]. It is hard, if not infeasible, for these tools to obtain such data as they focus only on the execution time of individual methods.

The queuing time of a task is influenced by the other tasks that may compete for the same execution unit. We formally define task runtime dependency as follows.

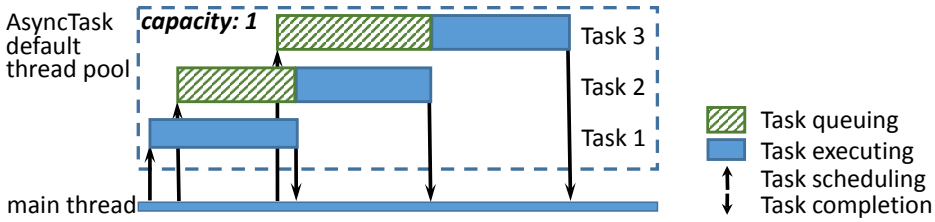


Figure 3.6: Asynchronous tasks runtime

Definition 3.3. *Two tasks bear **execution dependency** if 1) they run in the same execution unit; and 2) one task is scheduled in during the other task's queuing time.*

As shown in Section 3.3, task runtime dependency is a critical factor that influences the UI performance. Care must be taken to model task execution dependency. We propose to employ three queue-related features to model the task execution dependency. Specifically, they are the *queuing time*, the *pool capacity*, and the *queuing length* of a task.

Definition 3.4. *The **pool capacity** of an execution unit is the maximum tasks that the unit can simultaneously execute.*

For example, for a thread, the pool capacity is 1; while for a thread pool the pool capacity is its size. Pool capacity is usually set once and remains unchanged during runtime. Suppose k tasks bear runtime dependency in an execution unit with capacity N and $k > N$. $k - N$ tasks have to wait for execution in the unit. Then, when one task completes its execution, one of the waiting tasks can be executed.

Definition 3.5. *The **queuing length** of a task is the total number of tasks waiting for execution in the execution unit after it is scheduled.*

A queuing length L indicates that the task should wait for the completion of other L tasks before it can be executed.

Figure 3.6 illustrates how the three tasks discussed in Section 3.3.1 are executed. The thread pool capacity of `AsyncTask`

is 1. When Task 2 is scheduled, it has to wait until Task 1 finishes its execution. So the queuing length of Task 2 is 1. Similarly, the queuing length of Task 3 is 2.

The *queuing time* of a task reflects how other tasks influence its performance. The *queuing length* and *pool capacity* indicate the cause of a bad-performance task. Such information can greatly help performance diagnosis. However, existing tools for performance diagnosis (*e.g.*, Panappticon [196], Method Tracing [153]) cannot provide such information. As a result, it is difficult for them to diagnose the subtle performance issues caused by execution dependency.

We will elaborate how `DiagDroid` collects these runtime data in Section 3.5. Next, we will first discuss how `DiagDroid` conducts performance diagnosis with the collected data.

3.4.2 Dependency-aware Performance Diagnosis

When the UI is laggy, it indicates that a UI event procedure requires longer time to complete. As discussed, this can be rooted in either the synchronous part or the asynchronous executions. Although many tools (*e.g.*, [123, 90]) have well addressed the former case, `DiagDroid` moves a step further by focusing on the latter case, a far more difficult task in addressing the subtle performance issues caused by the asynchronous executions.

If the asynchronous part is laggy, it means at least one of the asynchronous tasks requires more time to complete. Consequently, `DiagDroid` should detect performance anomaly by checking whether there are any anomalous asynchronous tasks. Human inspection of all the involved tasks is prohibitively labor-intensive. `DiagDroid` requires an automatic way to detect anomalous tasks. A possible approach is to group the tasks in such a way that we can assume the tasks in the same group have similar performance. Then we can perform anomaly detection in a group basis.

But easy as it looks, how to group the tasks is challenging. An instant approach is to consider the *method call-stack* when a task is scheduled. We name such call-stack the *execution context* of the task. The execution context actually links to the source codes that define the task and how the codes are reached. Two tasks with the same execution context mean that they are corresponding to the same specific source code segment and execution sequence. Hence, they should naturally be grouped together.

However, this simple consideration will result in extensive debugging efforts. A code defect may manifest in similar tasks with slightly-different execution contexts. Reporting all such tasks one by one based on their execution contexts may be very tedious, and even make the diagnosis difficult with such tedious information. For example, two UI event procedures of two buttons may invoke the same buggy asynchronous task (in the source codes). In these two cases, the two task invocations have different execution contexts since they are invoked by different event procedures. But they should be grouped together to reduce human efforts in inspecting the codes.

`DiagDroid` addresses this challenge by putting *similar tasks* into a group with a proper definition of task *similarity*. By considering each method call as a symbol, an execution context can be encoded into a sequence. Then the difference of two tasks is the *edit distance* of their execution contexts. We adopt such an edit distance as a similarity measure due to the following considerations. First, it is suitable to model the differences of two tasks. Again consider the above example, if two UI event procedures of two buttons invoke the same buggy asynchronous task (in the source codes), the edit distance of the execution contexts of the two task invocations will remain close. As a result, they can be grouped together. Second, it also takes the invocation order information into consideration, where such an order in execution context is important to describe the program runtime. Consequently, two close execution contexts

indicate that the corresponding tasks are similar during program runtime. With such a similarity measure, `DiagDroid` conducts the single-linkage clustering, a widely-adopted sequence clustering method, to form groups [84].

`DiagDroid` examines whether performance anomaly manifests in each group of tasks by the execution context, queuing time and execution time. `DiagDroid` considers both the maximum value of queuing time and that of the execution time of all the tasks within each group A. The values are denoted by $M_q(A) = max(Q(a))$ and $M_e(A) = max(E(a))$ ($\forall a \in A$), in which $Q(a)$ and $E(a)$ are queuing time and execution time for task a respectively. $M_q(A)$ and $M_e(A)$ are then considered as the performance metrics of group A, since either a long execution time or a long queuing time can result in anomalous performance.

`DiagDroid` considers a group is anomalous if one of its two performance metrics is larger than a threshold τ. It ranks the anomalous groups according to their performance metrics as well as their execution contexts. Since each group is corresponding to a specific source code segment, the ranking can direct the manual debugging efforts towards a suspicious source code segment that may cause the performance anomaly. Moreover, a key consideration of `DiagDroid` is that the runtime dependency of tasks may also cause performance issues. In other words, the anomalous task *per se* may not always be the root cause of its poor performance. Especially, when the queuing time of the task is too long, the poor performance may be due to other tasks that bear runtime dependency with the poor-performance task. Therefore, `DiagDroid` also employs the performance data (the queuing length and performance) of such tasks to help locate the root cause. We will show in our experimental study that the localization approach can greatly reduce human efforts in locating performance issues.

The maximum values of queuing time and execution time are defined as performance metrics by `DiagDroid`. We consider the

maximum values instead of the average values. A large average value means that many tasks in the group are of poor performance, and is hence a good indicator of performance issues. However, in case that only a small portion of the tasks are with poor performance, the average value may still be small. But this can still be an important symptom of performance issues [72]. Since in the above two cases the maximum value will be large, it can indicate the performance issues. Hence, we consider the maximum value as the performance metric.

In the above discussions, we have considered a performance threshold τ as an indicator of whether a group contains poor-performance tasks. τ is selected empirically based on the developer's consideration on laggy UI. Previous work (*e.g.*, [141, 145, 166]) has suggested user-tolerable waiting time in web browsing, mobile web browsing and mobile services, which are from two to several seconds. One second is regarded as the limit for the user's flow of thought to stay uninterrupted [144]. We consider that mobile app users are more sensitive to UI response time. Therefore, we use $500ms$ as the value of τ. We will show in our experimental study in Section 3.6 that such a value is an effective choice.

3.5 Profiling Asynchronous Tasks

`DiagDroid` requires to profile the queuing time, the execution time, and the queuing length of task, as well as the pool capacity of the execution unit. Hence, `DiagDroid` must firstly track the life-cycle of a task, *i.e.*, when it is scheduled, when it is executed, and when it completes.

However, as discussed in Section 3.2, there are hundreds of ways for developers to implicitly or explicitly start asynchronous tasks. There are no sole entry/exit points for diverse types of tasks. It is difficult, if not infeasible, to design specific profiling mechanism for each. We attack this challenge with a separation-of-concerns

Table 3.2: Categories of asynchronous tasks

Category	Type	Representative classes
Reusing existing threads	Looper & Handler	HandlerThread
		IntentService
	Pool-based executor	ThreadPoolExecutor
		AsyncTask
Creating new threads	Thread	Thread

approach. We first find that the tremendous ways to start tasks can be classified into five categories (Section 3.5.1). Then based on the classification, we can specifically track and profile the necessary runtime data for each category (Section 3.5.2).

3.5.1 Categorizing Asynchronous Tasks

Via carefully inspecting Android source codes, we notice that the underlying mechanisms for Android to execute a task can be narrowed down into two approaches: 1) reusing existing threads created beforehand, and 2) creating a new thread. The former case can be further divided into two types: One directly schedules a task and the other requests the scheduling of a task by a delegate via sending a message. We call them the *Pool-based Executor* mechanism and the *Looper & Handler* mechanism respectively. We list them in Table 3.2, together with their representative classes in Android. We discuss them in what follows.

Both `HandlerThread` and `IntentService` depend on the Looper & Handler mechanism to start tasks. They create a *worker thread* and wait for new tasks to the *looper* associated with the thread. The request of scheduling a task is sent via a *handler* attached to the looper. The requested task will then wait to be processed in the worker thread. Since there is only one worker thread, it can process one message at a time. The other requests should wait in a queue.

`ThreadPoolExecutor` and `AsyncTask` both use the Pool-based Executor mechanism. They maintain a pool of worker threads with its number not exceeding a preset capacity. When a task comes, the task will be executed in one of the threads in the pool if there are available threads (*i.e.*, the number of threads which are executing other tasks is smaller than the capacity). Otherwise, the task has to wait for an available thread.

The `Thread` mechanism is relatively simple. Its building basis, *i.e.*, the `Thread` class in Android, is the same as the traditional Java one. This mechanism starts a task immediately in a new thread.

Although there are hundreds of ways for developers to implicitly or explicitly start asynchronous tasks, the underlying mechanisms are mostly based on these five representative classes. For example, the `AsyncQueryHandler` class, which is a convenient API for querying data from a content provider, is based on the `HandlerThread`. The `CursorLoader` class, which is often used for acquiring data from the database, is based on `AsyncTask`. Moreover, the `DownloadManager` which we have discussed in Section 3.2 employs the `ThreadPoolExecutor`. Verifying by the Android source codes, most types of asynchronous tasks are covered by the five classes except `TimerTask`. `TimerTask` is a class that defines periodical background tasks which usually would not update UI, and thus is not our focus.

Each of the five classes for conducting asynchronous tasks has its pros and cons. The *Thread* class is the most flexible one. A developer can get full control on the threading mechanism. The disadvantage is that more efforts are required to manage the thread. Moreover, creating a new thread for every task consumes system resources. The *HandlerThread* class requires many development efforts to customize both the background threads and the handler for the tasks. The *ThreadPoolExecutor* class is a traditional JAVA class for multi-threading. It is widely used to manage a pool of worker threads. However, it is not suitable for tasks that will update UI

since Android framework prohibits UI updating in worker threads. The *AsyncTask* class is specifically customized for Android development, which is convenient for UI update after the task finishes. The *IntentService* class can start a background service which is independent of the activity life cycle. It is a relatively heavier container compared with the above-mentioned classes. It requires more system resources on execution. Choosing which way to start asynchronous executions depends on the specific programming requirements, as well as the developer's preference.

3.5.2 Profiling Asynchronous Tasks

`DiagDroid` tracks the tasks with a dynamic instrumentation mechanism on the Android framework methods. It requires no changes to the target app, or recompiling the underlying OS and the Android framework. This can guarantee the compatibility of `DiagDroid` with diverse Android versions and device models. Moreover, it also requires little human efforts in installing and applying the tool.

Specifically, Android processes of its apps, unlike general Linux processes, are all set up by duplicating a system process called `Zygote` [64]. Android framework functionalities have already been loaded in `Zygote` before such duplication. Therefore, we can instrument the `Zygote` process and "hijack" the Android framework methods of interest before an app runs. Then when it runs by forking `Zygote`, the method invocations are inherently hijacked by `DiagDroid`. Hence, we can easily track the methods. We adopt such a mechanism implemented in the tool named Xposed [193], usually used for modifying UI [192]. We program our own codes to hijack the methods of our interest. Next we introduce how `DiagDroid` tracks the tasks in each category.

1) `Thread`: An asynchronous task that implements as a thread always starts with its `start` method. Hence, we can instantly obtain the time when the task is scheduled by tracking the `start`

method. However, the task is executed in the overridden `run` method of a `Runnable` object. An abstract method of an interface like `Runnable` cannot be instrumented directly. Hence, we resort to static analysis to find the implementations of the abstract `run` method and instrument these implementation methods instead.

The static analysis is performed via the tool `apktool` [38]. It decompiles the binary into well-structured Dalvik bytecode. They can be parsed to obtain the implementations of the abstract `run` method, which can direct our dynamic instrumentation approach to obtain the execution time. It is worth noting that `DiagDroid` only decompiles and discovers these methods, instead of modifying and recompiling the app before installation.

2) `HandlerThread`: `HandlerThread` is a thread that provides a `Looper` object attached to it. A `Handler` is associated with the `Looper` object and handles messages for the `Looper`. Hence, we can obtain the request time of a task by tracking the time when a `Message` object is sent to the `Handler`. Since eventually `sendMessageAtTime` or `sendMessageAtFrontOfQueue` must be invoked to send a `Message`, we record the invocation time of these two Android framework methods as the time when a task is scheduled. Since `Handler` actually performs the task by processing its corresponding `Message`, we track task execution by instrumenting its `dispatchMessage` method.

3) `IntentService`: An `IntentService` task always starts by invoking the `startService` method of the Android framework class `android.app.ContextImpl`. Hence, the invocation time of this method is recorded as the task scheduling time. `IntentService` actually relies on a `Hanlder` to process the task by an extended class `ServiceHandler` of `Handler`. Hence, we can track task execution by tracking the `dispatchMessage` method of `Handler`.

4) `ThreadPoolExecutor`: `ThreadPoolExecutor` is a pool-based execution mechanism. The class has an elegant pattern.

A task is always requested via invoking the `execute` method. Moreover, the task always starts immediately after the `beforeExecute` method, and is followed by the `afterExecute` method. Hence, we track tasks via these methods.

5) `AsyncTask`: A task can base its implementation on the complicated Java class inheritance of the basis `AsyncTask` class. It turns out that no matter how many layers of class inheritance are applied, a task is always scheduled by the `execute` method or the `executeOnExecutor` method eventually. Hence, we can record the task scheduling time by monitoring the invocation time of the two methods. We find in the Android framework source codes that for both cases, `AsyncTask` actually relies on the `ThreadPoolExecutor`. Hence, we can utilize the similar way as that for `ThreadPoolExecutor` to track task execution.

For the tasks in categories 2-5, they are put into a queue before they are executed. To model task runtime dependency for these cases, we use the hash code of the execution unit (*e.g.*, `ThreadPoolExecutor` object) as the *queue identifier*. Such a hash code is easy to obtain during runtime according to Java specifics. Two tasks with the same queue identifier suggest that they may bear runtime dependency. Finally, the pool capacity could be obtained via checking some internal field of the queue object (*e.g.*, the maximumPoolSize field of a `ThreadPoolExecutor` object). Details are omitted here, but can be found in our source codes [69].

The concrete methods for hooking may vary among different Android framework versions. We have remarked the lines of `DiagDroid` source codes that need to be modified if the implementation of the framework changed. Thereafter, validating `DiagDroid` for new Android versions is acceptable since currently Google usually releases one new version per year from 2013.

3.6 Experimental Study

We have implemented `DiagDroid` and released the project together with a tutorial open-source online [69]. In our experimental study, we target on open-source apps since we need to inspect the source codes to verify the effectiveness of `DiagDroid`. To this end, we download such apps from F-Droid, the largest app market that hosts only free and open-source Android apps [74]. It is also a popular app source for the research community [132]. We employ Monkey [179] to exercise our target apps. It is an official random testing tool delivered by Google and also known as the most efficient tool in terms of code coverage by empirical study [60]. Among the apps we download, we exclude those that require a login account for convenience consideration (so that we do not have to perform user registrations). Note that `DiagDroid` can easily handle such apps as well, by applying a login script when the apps start. The process is trivial and will not affect the effectiveness of `DiagDroid`. We thus get 33 target apps covering diverse categories including *Reading*, *Multimedia*, *Science & Education*, *Navigation*, *Security* and *Internet*.

We verify the compatibility of DiagDroid on four smartphone models covering a wide range of device capacities: Samsung GT-I9105P (Android 4.1.2), Huawei G610-T11 (Android 4.2.2), Huawei U9508 (Android 4.2.2), and Lenovo K50-T5 (Android 5.1). Experiments are conducted on the four devices simultaneously to save time. We also conduct stress tests by injecting loads on CPU, memory, sdcard IO and network respectively with customized Android background services. We implement five background services occupying 80% CPU, five background services occupying 80% memory, five background services consuming 2 Internet downloading threads each, two background services each reading 1 file and writing 1 file on sdcard in separate threads. The parameters are chosen by common practice. Developers could configure with their

own preferences. We design a system app to guarantee that these services would run persistantly (i.e., they will not be terminated by LowMemoryKiller [128]). Four devices with five configurations each (four with load injections and one without load injection) come up to 20 test configurations, each configuration is under Monkey testing for 30 minutes. We run 19,800-minute test in total for the 33 apps. `DiagDroid` reports overall 48 performance issues marked as highly suspicious for 14 apps, on average 3.4 issues for each. The reports are also issued on our website. Via inspecting the related source codes in the reports for several minutes per case and understanding of the original project, we surprisingly find 14 of the target apps contain 27 performance issues. The bug cases are ranked high in the report, (with an average rank of 1.7). Although unfamiliar with the target app design, we find it very convenient for us to pinpoint the root causes of the issues.

We categorize the 27 detected issues into 5 categories. Ten representative issues are presented in Table 3.3, together with their causes, the locations of their defects and their rankings in the reports. Detailed descriptions of all issues can be found in our website [69]. We have reported the issues to the app developers, many of which have been confirmed and corrected accordingly. We have got positive feedbacks like "for faster search results :)", "I've modified and I see the performance improvements." after the developers fix the performance problems. Next, we will elaborate our experiences in applying `DiagDroid` to the performance diagnosis via five representative cases.

3.6.1 Case studies

Case 1: Unintended Sequential Executions

We provide our experiences on applying `DiagDroid` to diagnose `OpenLaw` (`https://openlaw.jdsoft.de/`), an app under Science & Education category providing access to over 6,000 laws

Table 3.3: Representative performance issues found (Rank: ranking of the buggy issue / total number of issues reported)

Category	Issue description	Class@App	Rank
Sequential execution	Not awaring AsyncTask.execute() method results in undesired seqential execution	LawListFragment@OpenLaw	1/4
	Loading tens of icons in sequence	AppListArrayAdapter@AFWall+	1/3
Forgetting canceling execution	Improper cancelation of asynchronous tasks	GetRouteFareTask@BartRunnerAndroid	1/4
	Not canceling obsolete queries when new query arrives	AsyncQueryTripsTask@Liberario	2/2
Improper thread pool	Failed to set optimal size of the thread pool	ZLAndroidImageLoader@FBReader	1/2
	Use the same pool for loading app list and app icons	MainActivty@AFWall+	2/3
Overloading message queue	Posting various types of tasks (e.g., update progress, store book) to the same backgroundHandler	ReadingFragment@PageTurner	3/9
	Executing Filter method of AutoComplete-TextView occupies the Handler of a public message queue	LocationAdapter@Liberario	1/2
Misusing third-party library	Not canceling the tasks implemented by third-party library, Android asynchronous http client - loopj	HeadlineComposerAdapter@OpenLaw	4/4
	Use the deprecated findall method of WebView class which causes blocking	MainActivity@Lucid Browser	1/5

and regulations. `DiagDroid` reports 3 highly suspicious performance issues. It took us about half an hour to inspect the related source codes according to the report, and we can summarize 2 performance issues and localize the causes. One case is that the task group with context c_1 is anomalous. It contains the tasks with queuing time longer than the threshold $500ms$. This case is found in 14 test configurations, mostly under those with heavy CPU or sdcard IO load, which indicates the case may be related to some IO intensive operations. We demonstrate the content of `DiagDroid` report in Figure 3.7 (The line numbers are added for discussion convenience).

We can instantly find that such a long queuing time is because the corresponding anomalous task should wait in queue till the completion of other tasks with long execution time based on Lines

1. asynchronous tasks with context c_1
2. max queuing time: 1650ms
3. pool capacity: 1
4. cases with queuing time ≥ 500ms:
5. avg. queue length: 1.00
6. avg. execution time of the in-queue tasks: 1666.00ms
7. runtime dependency: c_2

Context c_1
Class name: de.jdsoft.law.data. UpdateLawList
Call-stack: android.os.AsyncTask.executeOnExecutor (Native Method)
android.os.AsyncTask.execute (AsyncTask.java:535)
de.jdsoft.law.LawListFragment.onCreate (LawListFragment.java:91)
•••
Context c_2
Class name: de.jdsoft.law.data.LawSectionList
Call-stack: android.os.AsyncTask.executeOnExecutor (Native Method)
android.os.AsyncTask.execute (AsyncTask.java:535)
de.jdsoft.law.LawListFragment.onCreate (LawListFragment.java:87)
•••

Report

```
public void onCreate(Bundle savedInstanceState) {
    •••
    // Load actual list
    final LawSectionList sectionDB = new LawSectionList(LawSectionList.TYPE_ALL);
    •••
    sectionDB.execute(adapter);
    // And parallel update the list from network
    UpdateLawList updater = new UpdateLawList();
    updater.execute(adapter);
    •••
}
```

Codes

Figure 3.7: Report and code segments of case 1

4-7. We know on average *one* task (Line 5) bears runtime execution dependency with an anomalous task, of which the context is c_2 (Line 7). In other words, the anomalous c_1 tasks are due to the heavy-weighted c_2 tasks. The pool capacity is only 1 (Line 3), which indicates the tasks have to run in sequence. Waiting for the completion of another heavy-weighted task should generally be avoided via proper scheduling. We can then focus on its cause.

The c_1 and c_2 contexts are included in the report, as shown in Figure 3.7. We can conveniently find the source codes of the two

tasks and where they are scheduled. Also shown in the figure, the two tasks are scheduled via `execute` methods. The developer has commented "parallel update", which shows the intention is to execute the two tasks in parallel. But this is a well-known mistake [36], since calling `execute` will actually call the same method of the super class `AsyncTask`, which will insert the tasks into a global thread pool with capacity one. Hence, we can instantly notice such a defect via inspecting the short code segment. The fix is to call `executeOnExecutor` instead with a larger pool.

Note that such bugs are quite common in Android apps. Shown as another issue in Table 3.3, developers of popular Android Firewall app (AFWall+) are also unaware of their inefficient sequential loading of icons until we report it to them. Developers are generally not aware of how their customized tasks are scheduled during app runtime, and they wrongly assume that the execution unit is free when scheduling a task. Sometimes, such *sequential* tasks may be defined and scheduled in different source files, making it harder to capture their execution dependency manually.

In the example, `OpenLaw` requires to handle tens of UI events. It is hence difficult, if not infeasible, to manually test and detect performance issues in tens of UI event procedures. Even if the developer is aware of which UI event procedure (loading `LawListActivity` in this case) is laggy, it is still hard for her to locate the defect by inspecting nearly 300 hundred lines of codes distributed in several files, which even involves complicated third-party library invocation.

Existing tools (*e.g.*, Method Tracing [153], Panappticon [196]) however focus essentially on the execution time of methods. They generally lack the capability to model the queuing time of tasks and to identify the execution dependency. It is therefore difficult for the developers to detect the subtle symptoms and reason the defect caused by task execution dependency, especially when two dependent tasks are executed in different places. Take the sequential

```
1.    asynchronous tasks with context c1:
2.          max queuing time: 31885ms
3.          pool capacity: 1
4.          cases of queuing time ≥ 500ms:
5.                avg. queue length: 19.50
6.                avg. execution time of the in-queue tasks: 1240.60ms
7.          runtime dependency: c1, c2
8.     execution time of c1   max: 19337.00ms   avg. 1419.95ms
9.     execution time of c2   max: 3446.00ms    avg. 786.69ms
```

Report

```
protected String doInBackground(Params... paramsArray) {
    Params params = paramsArray[0];
    if (!isCancelled()) {
       return getFareFromNetwork(params, 0);
    } else {
        return null;
    }
}
```

Codes

Figure 3.8: Report and code segments of case 2

execution issue in Osmand (reported in [69]) as an example. The buggy class has over 500 lines of codes (LoC), and there are over 300 LoC between two sequentially-scheduled tasks. Moreover, it is worth noting that the method tracing-based tools will generate a trace of thousands of methods for a UI event procedure. The performance diagnosis based on such tracing data is like finding a needle in the hay stack, given the fact that GigaBytes of data will be produced with Method Tracing for a simple 30-minutes testing run.

In contrast, `DiagDroid` properly models the task execution dependency, and provides tidy but helpful information to guide the diagnosis process. We show that it can greatly reduce the human efforts by directing the developer to several lines of codes that cause the performance issue.

Case 2: Not Canceling Obsolete Tasks

The cancelation of a time-consuming task is necessary when the task is no longer required. For example, consider an app allowing activity-switching with a sliding operation, which is commonly used in facilitating to quickly locate the activity of interest. Suppose the current activity shows the Internet content being downloaded by a task. When a user performs activity-switching, the Internet content is not necessary since its associated activity is invisible. The downloading task becomes obsolete. Obsolete tasks may occupy resources (*e.g.*, the Internet bandwidth), and therefore deteriorate the UI performance. Sometimes, they may even block other tasks. For example, they can occupy the thread pool and cause other tasks to wait. However, canceling obsolete tasks is not obligatory and hence it is often neglected by developers. Our experiment reveals that many popular Android apps contain performance issues caused by not canceling obsolete tasks.

We take `BartRunnerAndroid`, a public transport app, as an example. `DiagDroid` finds 5 highly suspicious performance issues. We can then conveniently locate 5 bugs in the source codes with the report. Specifically, we detect the anomalous c_1 tasks under all 20 test configurations. The corresponding content of the report is demonstrated in Figure 3.8. Similar to Case 1, we can instantly find that the long queuing time is caused by waiting for various tasks that bear execution dependency with the anomalous tasks (Lines 4 to 7).

It seems that we can apply a fix like that in Case 1 to this issue, *i.e.*, by allowing each type of tasks to run in its own execution unit. However, we are aware both tasks could be executed for a long period (Lines 8-9). `DiagDroid` tells us the information that many of these two long term tasks block the others. By inspecting the codes of the c_1 and c_2 tasks, we find the developers have already intended to cancel the obsolete tasks. Taken the c_1 task as example, it is actually inherits from `GetRouteFareTask`. Its source code

```
1.  asynchronous tasks with context c1:
2.      max queuing time: 514ms
3.      pool capacity: 3
4.      cases of queuing time ≥ 500ms:
5.          avg. queue length: 1.50
6.          avg. execution time of the in-queue tasks: 659.29ms
7.      runtime dependency: c1
```

Report

```
private static final int IMAGE_LOADING_THREADS_NUMBER = 3;//TODO: how many threads?
private final ExecutorService myPool = Executors.newFixedThreadPool(
    IMAGE_LOADING_THREADS_NUMBER, new MinPriorityThreadFactory());
...
void startImageLoading(...){
    final ExecutorService pool =
        image.sourceType() == ZLImageProxy.SourceType.FILE ? mySinglePool : myPool;
    pool.execute(...);
}
```

Codes

Figure 3.9: Report and code segments of case 3

for execution is illustrated in Figure 3.8. The cancelation checking is done before the time-consuming networking task begins which means it will not be canceled during its execution even when it is obsolete. Similar analysis could be applied on c_2. In other words, the developers fail to conduct proper cancelation steps for these tasks via cancelation checking.

Note that the correct way of cancelation involves two steps: 1) call `cancel` method of the `AsyncTasks` in the `onStop` method of the container `Activity`, and 2) periodically check the result of `isCancelled` function in `doInBackground` and release the resource when it returns `true`. Note that releasing the resource in `AsyncTask.onCancel` is also a common mistake that should be avoided.

Case 3: Improper Thread Pool Size

Improper thread pool size is a typical cause of poor performance. It may cause long queuing time of tasks since the pool is often busy. We will show how `DiagDroid` handles such defects via our

experiences on diagnosing `FBReader` (`https://fbreader.org/`), a popular e-book reader.

`DiagDroid` reports only one performance issue for this app as shown in Figure 3.9. We detect this case under all 20 test configurations. We then inspect this performance issue in the source codes and we find the bug in 20 minutes. Both the execution time and the queuing time of the c_1 tasks are anomalous. According to the context of c_1, we can quickly locate the source codes and confirm that the execution time is reasonable since the tasks load images. However, the reason for the long queuing time is that a c_1 task has to wait for other c_1 tasks to complete (Lines 4-7). Unlike those in Case 2, c_1 tasks however cannot be canceled since they should run in parallel to load many images simultaneously. Hence, a quick fix for this defect is to set a larger pool size (`DiagDroid` confirms that 5 is a good choice).

Properly setting a pool size is often not easy for a developer during the coding phase. It is difficult for her to predict the possible number of concurrent tasks in the same pool. For example, in this case study, the developer is not sure about the proper setting of the pool size, and hence put down a to-do comment in the source codes (see Figure 3.9). We find four such cases in our experimental study.

Finally, note that the existing approaches [153, 196] cannot identify the performance issues caused by the defects in Cases 2 and 3, since these approaches focus only on the execution time. Even if the developer is aware of the bug symptoms, the existing approaches do not provide a way to automatically analyze the runtime dependency of the tasks. As a result, it requires daunting manual efforts to find that a task *sometimes* has to wait for other tasks by the inspection of tremendous runtime traces.

Case 4: Overloading Message Queue

Similar to the thread pool, message handler is also a queue-based execution unit. But a handler has only one thread to process

```
1.  asynchronous tasks with context c1:
2.      max queuing time: 6138ms
3.      pool capacity: 1
4.      cases of queuing time ≥ 500ms:
5.          avg. queue length: 1.38
6.          avg. execution time of the in-queue tasks: 3274.31ms
7.      runtime dependency: c1
```

Report

```
// This method could be optimized a lot, but hey processors are fast nowadays
protected FilterResults performFiltering( ... ) {
    ...
    resultList = autocomplete.execute().get().getLocations();
    ...
}
```

Codes

Figure 3.10: Report and code segments of case 4

incoming messages which makes it vulnerable to performance bugs. Messages could come from separate UI operations (*i.e.*, continuous text inputs). If messages come too quickly, a message may wait for the handler to be available as the handler may be busy processing the previous messages. This type of issues can also be easily pinpointed with `DiagDroid`. We find two such cases in our experimental study. Next we discuss the `Liberario` (a public transport app) case. `DiagDroid` reports two highly suspicious performance issues, from which we find two code defects in less than an hour. We report the cases and the developers have confirmed them and modified the apps accordingly.

Figure 3.10 demonstrates part of the report. We can see that the c_1 tasks are with anomalous queuing time and execution time. This problem is detected under all 20 test configurations. The cause of the long queuing time is that a c_1 task has to wait for other c_1 tasks to complete (Lines 4-7). The long execution time indicates that a c_1 task may long occupy the corresponding handler. Via inspecting the source codes of c_1, we can easily find the cause. The c_1 task is a `Handler` for filtering input text, which invokes `performFiltering` directly. The method requests a list of

```
1.  asynchronous tasks with context c1:
2.      max queuing time: 664ms
3.      pool capacity: 1
4.      cases of queuing time ≥ 500ms:
5.          avg. queue length: 3.00
6.          avg. execution time of the in-queue tasks: 323.00ms
7.      runtime dependency: c1
```

Report

```
public void onTextChanged( ... ) {
    WV.findAll(s.toString());
}
```

Codes

Figure 3.11: Report and code segments of case 5

suggested locations with an incomplete location. It involves a time-consuming Internet query. Consequently, the upcoming messages will have to wait in the message queue. Actually this is not needed since the old query is no longer useful, and hence should be canceled so that the handler can process new messages.

This case is very difficult to be identified. The developers, after we report the case to them, update 11 files to fix the issue. If using existing approach [153], we can possibly know that the execution time is long. However, we may accept the fact that Internet query is time-consuming. Moreover, Panappticon [196], as an event tracing system to identify critical execution paths in user transactions, is not aware of the dependency between tasks invoked in different UI operations. For example, in this case, tasks bearing execution dependency are invoked by independent text inputs in the `AutoCompleteTextView`. It is generally hard to know how a long execution time may influence other tasks, without a tool like `DiagDroid` to properly model the runtime dependency of the concurrent tasks.

Case 5: Misusing Third-party Library

Misusing third-party libraries is also a source of performance issues. Developers usually call a third-party library without knowing its

implementation details. Misunderstanding the usage of the library may introduce performance issues. For example, unaware of the asynchronous tasks in a third-party library, the developer will neglect to cancel them when they are obsolete. `DiagDroid` can also reduce the efforts in troubleshooting such defects as well. We detect three such cases. We present how `DiagDroid` find such a defect in `Lucid Browser`, a web browser app. We infer this issue from the 3 suspicious issues reported by `DiagDroid`.

The report is presented in Figure 3.11. We can know this is a similar defect like that in Case 1 based on Lines 3-7. It involves unintended sequential executions. Checking source codes related to c_1, we can locate the invocation of the `findAll` method in the third-party library `WebView`. It finds the occurrences of a specific text in a webpage when the text is modified. Revisiting `findAll`, we can find that it is deprecated, and should be replaced by `findAllAsync`.

Note that the case is only detected on 3 devices other than Lenovo K50-T5 (Android 5.1). `findAll` method does not introduce performance issues in Android versions above 4.4, as the `Webkit`-based `WebView` is replaced by the `Chromium`-based `WebView` in recent Android versions. Existing tools like Panappticon [196] which require to recompile the kernel, can work only on a small set of devices. It is hard for such tools to cope with such defects that do not persist in all Android versions [46].

3.6.2 Why Clustering

As mentioned in Section 3.4.2, to reduce the number of reporting suspicious cases, we cluster the execution contexts (*i.e.*, call-stacks) belonging to the same asynchronous task triggering by similar running sequences. First we extract two features via scanning through massive contexts: 1) similar call-stacks are similar in line level, and 2) similarity of call-stacks is transitive. Then we perform

	Context c_1	Context c_2	Context c_3
1.	de.jdsoft.law.data.UpdateLawList	de.jdsoft.law.data.UpdateLawList	de.jdsoft.law.data.UpdateLawList
2.	android.os.AsyncTask.executeOnExecutor(Native Method)	android.os.AsyncTask.executeOnExecutor(Native Method)	android.os.AsyncTask.executeOnExecutor(<Xposed>)
3.	android.os.AsyncTask.execute(AsyncTask.java:534)	android.os.AsyncTask.execute(AsyncTask.java:534)	android.os.AsyncTask.execute(AsyncTask.java:539)
4.	de.jdsoft.law.LawListFragment.onCreate(LawListFragment.java:91)	de.jdsoft.law.LawListFragment.onCreate(LawListFragment.java:91)	de.jdsoft.law.LawListFragment.onCreate(LawListFragment.java:91)
	…	…	…
9.	android.support.v4.app.FragmentActivity.onCreateView(FragmentActivity.java:285)	android.view.LayoutInflater.createViewFromTag(LayoutInflater.java:676)	android.view.LayoutInflater.createViewFromTag(LayoutInflater.java:727)
	…	…	…

Figure 3.12: Similar contexts without clustering

clustering accordingly. In this section, we show that the clustering is necessary and effective with a randomly selected example of app `OpenLaw`.

For `OpenLaw`, there are totally 1462 distinct contexts found under all 20 test configurations. As a result, 226 suspicious performance cases are reported. However, we find many of the reported cases are with very similar contexts. We select three similar contexts as examples in Figure 3.12. Actually, the three contexts refer to the same asynchronous task `UpdateLawList` presented as context c_1 in Case study 1. They differ from each other only until the 9th line of the call-stacks; more specifically, they only have slight difference in low-level VM processing sequences. There is only one performance issue instead of three in developers' viewpoint. To lighten the workload of developers, the three contexts should be grouped into one.

Considering the aforementioned features, we cluster contexts with a customized edit distance (feature 1) plus single-linkage strategy (feature 2). After the clustering, we successfully reduce the amount of total contexts from 1462 to 75 (groups). Moreover, only 7 performance cases are reported without losing meaningful cases. This result indicates the effectiveness of our clustering mechanism.

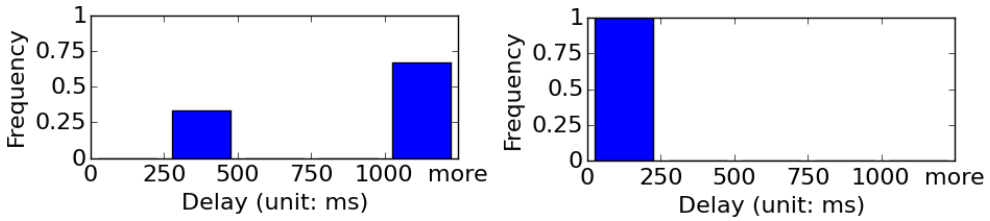


Figure 3.13: Message handler blocking delays before (left) and after (right) fix of Transportr

3.6.3 Performance Enhancement

`DiagDroid` is able to present to developers with the performance enhancement after fixing performance issues. `DiagDroid` offers the distribution of the queuing & execution delays of *asynchronous executions*. Besides confirming the disappearing of the related case in the report of the fixed version, developers can ensure the performance gain via double-checking the delay distributions of related asynchronous executions before and after fixing the issue. Next, we illustrate how to visualize the performance enhancement via demonstrating two official fixes by developers. Notice the example distributions are simplified (yet good enough) to demonstrate the enhancement. Developers could tune the parameter to obtain finer distributions.

Developers of `Transportr` fix an issue of message queue overloading with our report. They modify 11 files with 364 additions and 274 deletions. Since the message processing is network related, we show the performance enhancement on the Huawei G610-T11 with network load injected. The result is depicted in Figure 3.13. Similar patterns can be found in other test configurations. It could be seen that there is no more blocking problem for the new app version.

With our report, developers of `AFWall+` fix the sequential loading issue on displaying the app list. They modify the source from `(new GetAppList()).setContext(this).execute()` to `(new GetAppList()).setContext(this).execute-OnExecutor(AsyncTask.THREAD_POOL_EXECUTOR)`. We

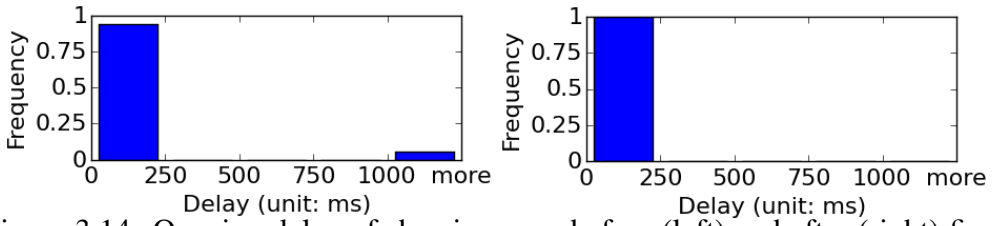


Figure 3.14: Queuing delay of showing apps before (left) and after (right) fix of AFWall+

notice that the queuing effect is more obvious with CPU load injected on a poorer device. We illustrate in Figure 3.14 the distribution of queuing delay of the `AsyncTask` with testing configuration of Samsung GT-I9105P with CPU load injected. Similar patterns can be found in other test configurations. It could be seen that there is no more queuing problem in the new app version.

3.6.4 Discussions of Experiment

Next, we discuss the threats to the validity of our experiments, and the measures we take to address them. First is the overhead of `DiagDroid`, which directly reflects how it affects the test efficiency. We conduct $10,000$ Monkey operations with `DiagDroid` on and off. The interval between two consecutive operations is $200ms$. We employ the Android `time` command to obtain the CPU time for both cases. Their difference is then the overhead of `DiagDroid`, which is 0.8%. This shows that `DiagDroid` does not have a considerable impact on testing efficiency. Moreover, we use the maximum latency among all test runs of one type of task as indicator of problematic tasks, commonly the value would be much larger than the threshold (*i.e.*, 500ms). Therefore, the overhead of `DiagDroid` seldom makes a task to become problematic.

We choose Monkey as our test executor for its efficiency [60]. However, `DiagDroid` does not rely solely on `Monkey` as its test executor. Test executor is a plug-in in `DiagDroid`. It also allows to incorporate other automate script-based testing tools like

`UIAutomator` [177] and `Monkey runner` [138].

We test 30 minutes for each of 20 configurations per app to show the capability of `DiagDroid` in such a short-term test. The settings can be changed according to the app specifics to explore the app more thoroughly. We show the capability of `DiagDroid` in even such a short-term test. Note that `DiagDroid` also carefully addresses its compatibility issues. Our tool requires no changes to the target app, or recompiling the underlying OS and Android framework. It is proven to be compatible over various device models and Android versions. Parallel testing in multiple mobile devices with diverse models is feasible.

We have detected 27 real world performance issues among 48 reported cases. The remaining 21 issues are mainly reasonable time-consuming tasks (*e.g.*, download tasks). We have provided developers a way to filter out such tasks by name while to be genetic for all apps we do not use such filter in our experiment.

Finally, `DiagDroid` resorts to dynamic analysis to diagnose performance issues. We focus on the performance issues caused by complicated runtime dependency of asynchronous executions. It is hard, if not infeasible, for the static analysis approaches to deal with such types of issues. For example, determining the proper pool size based only on the source codes is hard, since it is impossible to predict the possible number of concurrent tasks. Also, it is difficult to determine when to cancel a task beforehand based only on the source codes. Human efforts are inevitable. The aim of `DiagDroid` is to reduce such efforts, rather than approaching the task of automatic code correction. As shown in our case studies, such efforts are light. `DiagDroid` is able to provide a small set of possible performance issues. The issues that really contain bugs are with high rankings. As shown in Table 3.3, the buggy cases rank 1.7 averagely. This indicates that we can easily be directed to where the code defect lines. The debugging time of each case is generally less than an hour, even for us who are familiar with the

app implementations.

3.7 Tool Insights and Discussions

3.7.1 Tips for Developers

We have learned many kinds of bugs from the experiment. Some tips for developers could be concluded from these bugs.

Use a Private Pool Instead of the Public One When Necessary

It is necessary to define your own thread pool instead of using a public pool like `AsyncTask.THREAD_POOL_EXECUTOR` in some cases. The thread pool is used either for directly executing tasks on or being called by `executeOnExecutor` of `AsyncTask`. Current implementation of `AsyncTask.THREAD_POOL_EXECUTOR` has the thread pool size under 10 in usual situations. Therefore if too many `AsyncTasks` are put into the pool, it is easy to be used up. We suggest defining your own thread pool especially for the case that the asynchronous execution takes a long time or the asynchronous execution will not update UI after finishes. In the former case asynchronous execution would occupy a thread in the pool for a long time. In the latter case, the asynchronous execution is not necessary to be implemented as `AsyncTask`.

Set Reasonable Pool Size

When using pool-based asynchronous execution, the developers should carefully set the pool size. A small pool may increase the chance for an asynchronous execution to wait in the queue. On the other hand a large pool may waste system resource if starving from works. A practical choice is that the developers could just try to set the pool size several times and run our tool. After getting the reports

for every run, a best choice among the previous settings could be found.

Use Third-party Library Carefully

The developers should inspect the specification of third-party libraries carefully when utilizing them especially when there are potential asynchronous executions. It is convenient to use third-party library. However, the developers are not generally aware of the implementation details of the library. The library may provide multiple implementations for an operation to suit different requirements and situations. The implementations are separated by parameters passed when calling the function. The default setting of parameters might be only a moderate setting that meets most requirement. If the developers did not read the specification carefully, they may set problematic parameters that do not fit into their requirement. In which case, there would be potential performance bugs.

Keep Effective Response

Most developers know they should keep the screen response when an asynchronous execution is performed. But they may sometimes forget to make an effective response. In some cases, the developers simply display a progress bar/circle in the entire screen to show the execution is in progress. The user may get annoyed if a progress bar displays for long. It could get even worse when the back button is disabled and the user could do nothing but wait for the execution. It is acceptable that some operations could be delayed until an asynchronous execution is finished. Nevertheless, at the same time the app should keep response to user interacts and better display some meaningful pages.

Cancel When No Longer Needed

The developers often do not take care of cancelling the asynchronous executions when they are no longer needed. For example, an asynchronous task which loads figures in an activity should be cancelled after the activity is hidden or destroyed. Another example is when downloading a list of figures that would be displayed on the screen, the figures swiped out should be cancelled from downloading. All asynchronous execution requires resources including thread of pool-based executions, system resource like CPU, network and disk I/O, etc. If long term asynchronous executions are not cancelled properly, thread pool may become exhausted or system resources may become shorted (*e.g.,* network becomes slow).

Use Proper Type of Asynchronous Execution

There are different types of asynchronous executions, the inexperienced developers may not know which one is better to suit their requirement. For example, `AsyncTask` is a class simplifying generic asynchronous execution. Asynchronizer [123] would refactor long concurrent works into `AsyncTask`s. However, it is not a panacea. Officially it is suggested that an asynctask should be used for short operations (a few seconds at the most) [41]. For long term operations, it is common to introduce bugs in the design of `AsyncTask`. For example, in the codes shown in Figure 3.1, there is a bug in `RetrieveDataTask` class. It may take 20 seconds to retrieve the data from Internet. If during that time the activity is destroyed, however, the `RetrieveDataTask` will run until it finishes. When the `onPostExecute` is invoked, the text view no longer exists and the program would be crashed if the error is not catched. Therefore it is not appropriate to use `AsyncTask` for network requests which may take very long. Developers then resort to the `Loader` class. However, As suggested in RoboSpice project [163], `AsyncTask` has its limitation on performing network

requests. RoboSpice project [163] may be a better choice in this case. The developers should better understand the features of different types of asynchronous executions and apply suitable one when needed. Misusing of asynchronous executions could introduce performance bugs.

3.7.2 Discussions on the Implementation

Hooking Methods of Apks

Method hooking is the first step (pre-run step) of our testing framework. When talking about hooking methods in app, the first idea is to decompile and modify the apk file. There are already several tools that could do the decompile & recompile like smali [172] and apktool [38]. However, many apks would obfuscate the codes when release. Most obfuscated codes can hardly be hooked and recompiled by these tools. Therefore we resort to runtime hijacking the framework code which is more reliable.

3.7.3 Limitation of Our Tool

The `main` thread is different from worker threads. The `main` thread is not task-oriented. It keeps running during the entire life cycle of the app. We do not work on problems on the `main` thread. Therefore those performance issues caused by bad design of the `main` thread will not be detected by our tools. As mentioned in Section 3.3 several tools address and attempt to solve this problem.

Currently the tool is built only on Android platform. While as the methodology is generalized, the tool could be easily applied to other mobile platforms such as Window Phone, iOS. However, the framework methods interception of these platforms are more involved.

3.8 Summary

To conclude, this chapter focuses on an important type of Android performance issues caused by task execution dependency. We carefully model the performance of the asynchronous tasks and their dependency. Taken task dependency into consideration, we design `DiagDroid` for task-level performance diagnosis. It is equipped with a set of sophisticated task profiling approaches based on the Android multithreading mechanisms.

To make `DiagDroid` a practical, handy tool for performance diagnosis, we carefully consider the system design requirements like *compatibility*, *usability*, *flexibility* and *low overhead*. Specifically, `DiagDroid` relies solely on the general features of Android. Hence it works for most mainstream Android devices, depending on no manufacturer specifics. Moreover, `DiagDroid` is convenient to be used via a simple installation. It requires no efforts to recompile the operation system kernel and the Android framework. With a plugin mechanism, `DiagDroid` provides the flexibility in selecting the test executor that is required to exercise the app. It is convenient to apply different test executors based on the test requirements. Finally, `DiagDroid` keeps low overhead by instrumenting slightly on the framework.

We show `DiagDroid` can effectively reduce human efforts in detecting and locating performance issues. We apply the tool successfully in finding bugs in tens of real-world apps which we are unfamiliar with.

□ **End of chapter.**

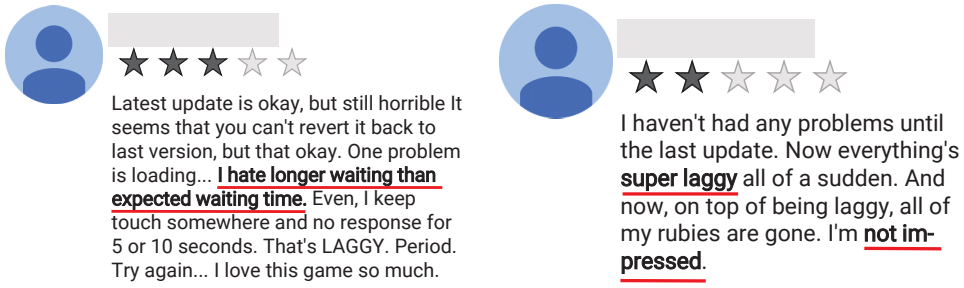
Chapter 4

Detecting Poor Responsiveness UI for Android Applications

Delay-Tolerant UI Design

This chapter presents `Pretect`, a tool that could detect poor responsive UI for Android apps, for delay-tolerant UI design. A key finding is that a timely screen update (*e.g.*, loading animations) is critical to heavy-weighted UI operations (*i.e.* those that incur a long execution time before the final UI update is available). We list the points of this chapter as:

- Conduct a real-world user study on finding many insights on the user experiences and the UI latency.
- Implement and open-source release the tool.
- Apply `Pretect` in diagnosing synthetic benchmark apps, open-source apps and commercial apps; Proven the correctness of the tool and successfully locate hundreds of poor UI designs.



(a) Rating 3 vs. average rating 4.4 (b) Rating 2 vs. average rating 4.3

Figure 4.1: Examples of user ratings and comments about bad UI responsiveness

4.1 Introduction

Rapid user interface (UI) responsiveness is a critical factor in the software quality of mobile apps. Apps with poor UI responsiveness lead to many user complaints [85]. Such performance defects are threat to software reliability [31, 187, 188]. Figure 4.1 presents two examples of user complaints on Google Play, a popular Android app market. Users give the app a low rating due to its poor responsiveness. Users may also have different expectations about UI latency (*i.e.*, the time between the commencement of a user operation and the corresponding UI update) in different UI operations. As suggested by the comments shown in Figure 4.1a, users "hate longer waiting than expected waiting time." Achieving rapid UI responsiveness and designing better UIs to boost user patience have long been goals of both the academic and industrial communities [33].

The key to user satisfaction with UI responsiveness is to offer timely UI feedback (*e.g.*, showing an animation to indicate a background task is being conducted) on user operations [104]. It is widely accepted that mobile devices may not be able to immediately complete all of the tasks intended by a UI operation due to resource limitations. For example, operations that involve loading Internet resources or accessing remote databases are types of "heavy-weighted" operations that require long waits before they are

complete. In such cases, it is necessary to provide quick UI feedback to the user to let her know that the operation is being processed.

However, designing such feedback for every possible heavy-weighted UI operations requires a daunting amount of development effort. Moreover, developers generally have no idea on the latency on each UI operation. Without a comprehensive performance test and a handy tool to record UI latency, developers may neglect potential heavy-weighted operations.

We consider operations that may require a long execution time without offering quick UI feedback as poor-responsive operations. Poor-responsive operations, as software design defects, should be detected before an app is released to limit their influence on user experience. However, there are currently no methods for detecting such defects. First, there is no comprehensive understanding of what degree of UI latency (*i.e.*, the latency threshold) leads to poor-responsive operations. The Android framework expects Android apps to be responsive in 5 seconds, otherwise it produces an "Application Not Responding (ANR)" alert [112]. Therefore, 5 seconds can be viewed as a loose upper bound for the latency threshold of poor-responsive operations. Alternatively, Google suggests that 200 ms as a general threshold beyond which users will perceive slowness in an application [112]. Therefore, 200 ms can be viewed as a lower bound for the latency threshold of poor-responsive operations. However, how much UI latency a typical user can really tolerate remains unknown, this information is a prerequisite for detecting poor-responsive operations.

Second, currently there is no tool that can detect poor-responsive operations during an app test run. The official tool StrictMode [90] or other tools like Asynchronizer [123] can detect operations that block the UI thread, but they cannot detect non-blocking operations that incur long UI latency. The official performance diagnosing tools Method Tracing (and TraceView) cannot correlate operations with UI updates [153]. Other performance diagnosis tools such

as Panappticon [196] also cannot capture the performance of UI feedback while a UI operation is being processed.

Hence, a tool that can detect poor-responsive operations during an app test run is necessary to combat poor UI responsiveness. In this work, we first provide a threshold for classifying poor-responsive operations based on a real-world user study. Moreover, we observe that Android apps generally use a unique pattern when conducting UI updates. Specifically, although Android has a complicated procedure for conducting UI updates that involves diverse components, we find that every app uses a similar communication pattern based on a specific system process of Android when it conducts UI updates. Therefore, we can design a tool to track such patterns and detect poor-responsive operations.

The contributions of this chapter are as follows.

1. We conduct a real-world user study via a comprehensive survey. Where we find many insights on the user experiences and the UI latency. We thus find a reasonable threshold to detect poor-responsive operations.

2. A handy tool, called `Pretect` (*P*oor-*Re*sponsive UI De*tect*ion in Android Applications), is implemented and has been open-source released [37]. The tool aims at detecting poor-responsive operations. We use the tool to find many UI design defects in many real-world Android apps.

4.2 Motivation

Android has unique UI design patterns. The main thread of an app is the sole thread that handles the UI-related operations [152], such as processing user inputs and displaying UI components (*e.g.*, buttons and images). When a valid user input (*i.e.*, a UI event) comes, the main thread invokes the corresponding *UI event procedure*, *i.e.*, the codes that handle the UI event. However, some UI event


```
private class ImageDownloader extends AsyncTask<String, Void, Bitmap> {
    protected Bitmap doInBackground(String... urls) {
        return downloadBitmap(urls[0]);
    }

    protected void onPostExecute(Bitmap result) {
        imageView.setImageBitmap(result);
    }
}
```

Figure 4.2: Screenshot and related source codes of a simple gallery

procedures may be time-consuming; for example, a procedure may involve downloading a large file from the Internet. Android prevents such procedures from blocking the main thread, *i.e.*, unable to respond to other user operations, by introducing the Application Not Responding (ANR) [112] mechanism.

Commonly, Android apps conduct heavy tasks in an asynchronous manner so as not to block the UI thread. More specifically, when accepting a user operation, an app will start executing time-consuming tasks asynchronously in threads other than the main thread. Once the asynchronous task is finished, there is usually a callback to the main thread to update the UI accordingly [152].

The UI responsiveness is poor if the asynchronous task takes a long time to execute and there is no feedback (*e.g.*, no loading animation) during its execution. Providing no feedback to users is a common mistake of developers. Figure 4.2 shows a simple

gallery design and the source codes of its core functionality. The `ImageDownloader` task downloads the image with the given URL and then updates the imageView. Although such functionality is trivial, it contains a defect that may lead to slow UI responsiveness. When the image is large, or the Internet connection is poor (*e.g.*, on a cellular network), the user has to wait a long time before the image is loaded. The app provides no feedback during this waiting period. The user may be uncertain whether she has successfully touched the "next" sign shown in Figure 4.2. A better design for this case is showing the progress of loading with `onProgressUpdate` function or displaying an instance of `ProgressBar`/`ProgressDialog`.

Although Google has offered some responsive Android widgets such as `SwipeRefreshLayout`, in most cases, they only provide suggestions for improving responsive UI design (*e.g.*, showing the progress of background work using a ProgressBar) [112]. In next section, we present a user study that shows how poor-responsive UI design hurts an app.

4.3 User study

Unlike traditional PC apps, mobile apps are usually executed in an exclusive manner. Users do not generally switch their focus to other windows while an app is being used. Therefore, users tend to be more impatient with delays in mobile apps. This effect is not studied before, thus we design a user study to reveal it. The study is available online [37]. Our study examines the relationship between UI latency and user patience. The results can guide the design of our tool.

We implement a mobile app with different delay levels and collect user feedback. The feedback reveals user tolerance under different delay levels. The app interacts with the users using common multimedia (*e.g.*, video, audio, microphone, etc.). The app requests user interaction (e.g., clicking buttons) before it continues

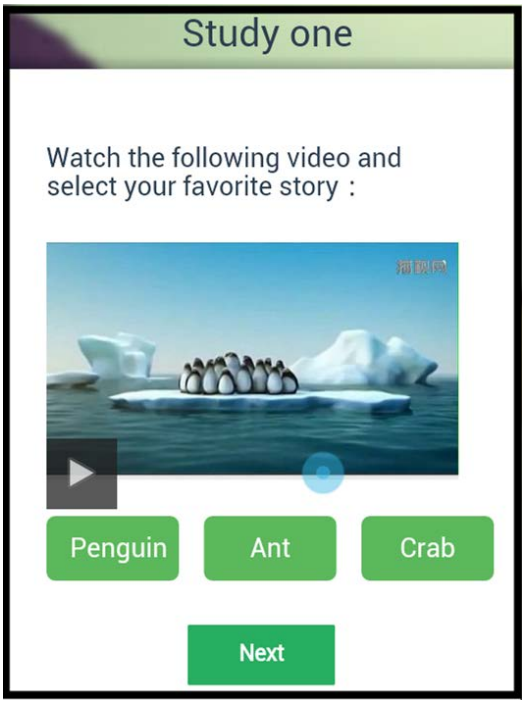


Figure 4.3: Screen shot of survey app

to the next page (*i.e.*, an Android activity). A designed delay is attached to each user interaction. A sample page of the app is shown in Figure 4.3. After running the app with a designed procedure, users are required to answer several questions related to their patience, each rated from 1 to 9 (*e.g.*, rate 1 for user impatient, 9 for user patient). The statistical analysis of the ratings show the trends in user patience.

4.3.1 Test settings

In the study, we set three delay levels: 200 ms, 500 ms, and 2 seconds (*i.e.*, we set feedback after the delay time). As it is hard for a single user to rate fairly each separate operation with a different delay, we use a between-subject design [89, 97]. Thus, each user has one assigned delay level throughout the test and rates the app's overall performance; then the rates are compared between the sets

of similar users.

The parameter settings are chosen based on the previous study of Johnson [104]; 200 ms is the editorial "window" for events that reach user consciousness, 500 ms is the attentional "blink" (inattentiveness to other objects) following recognition of an object, and 2 seconds is the maximum duration of a silent gap between turns in person-to-person conversation [104]. Google suggests 100 to 200 ms as the threshold at which users will perceive slowness in an application [112]. More levels may be helpful to obtain a finer threshold of user impatience. However, the threshold varies among different circumstances a conservative threshold is enough for us in our study. Moreover, too many delay levels would disperse the subjects (noted that we use between-subject test) which makes the result less meaningful in statistics.

We take the following measures to address possible threats to the validity of our user study. 1) Subjects do not rate objectively if they know the purpose of the study. Therefore, we design more questions than required to hide our purpose. We ask the subjects to rate for about ten common questionnaire questions (*e.g.*, [58, 92, 175]), only two of which are related to user patience. We also ask the subjects their understanding on the purpose of the study. The results verify that the subjects are unaware of our real purpose. 2) The ability to learn a new app varies among subjects. To remove this effect, we conduct a practice before the study, but do not include the results of the practice in our analysis. 3) Internet delays are a common source of delay [31, 173], which can make subjects impatient. Therefore, to avoid introducing extra Internet delays, our tests are all conducted on local area networks.

4.3.2 Results

We collect 116 valid replies. All of the subjects are college students between the ages of 20 and 27. We first collect the user feedback

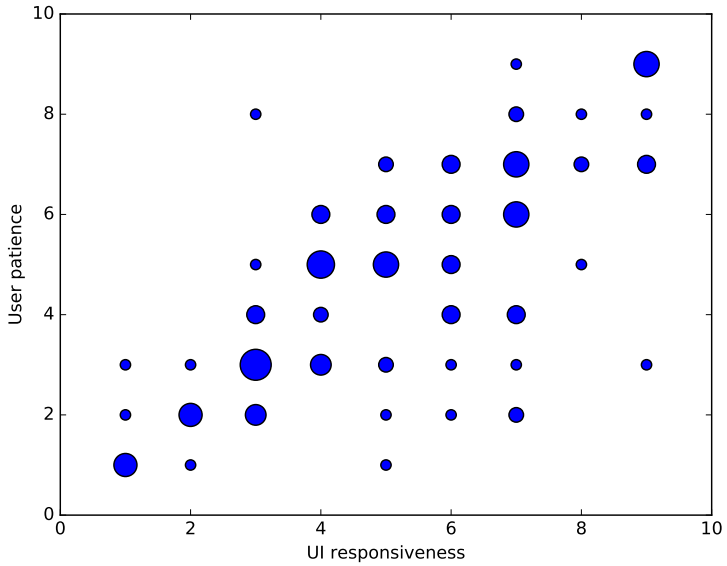


Figure 4.4: Rating on UI responsiveness and user patience (size of a circle is related to the number of repetitions of that point)

to get a general impression of user patience. Then we utilize SPSS tool for a detailed statistical analysis of the rating scores.

The written user feedback gives a general impression of the variation in user patience. Under the 200 ms delay condition, the subjects seldom complain about the responsiveness of the app, whereas under the 500 ms delay condition, there are many complaints. For example, “very slow response”, “Lagged response”, “poor performance and UI”, “very irresponsive”, “It really responds slowly”, etc. When the UI delay reaches 2000 ms, most subjects complain about the responsiveness of the app, and some even become angry. For example, “slow as f**k”, “too lag to use”, “seriously, it is irresponsive”, “extremely irrsponsive!”, etc. We can already see a relationship between user patience and UI responsiveness. We resort to statistical analysis for a more detailed understanding of the relationship.

To increase the reliability of the test, we ask questions about both UI responsiveness and user patience. The scatter plot of the relationship between the two factors are shown in Figure 4.4.

Table 4.1: Patience measurement under different delay levels

Delay level	Mean	Std. Deviation	N
200 ms	5.59	2.14	38
500 ms	4.68	1.80	37
2000 ms	4.24	2.43	41
Total	4.82	2.21	116

Table 4.2: Pairwise patience measurement comparisons on different delay levels

delay_level Simple Contrast			Dependent Variable B2Items	delay_level Simple Contrast			Dependent Variable B2Items
500 ms vs. 200 ms	Contrast Estimate		-0.92	2000 ms vs. 200 ms	Contrast Estimate		-1.35
	Hypothesized Value		0.00		Hypothesized Value		0.00
	Difference (Estimate -		-0.92		Difference (Estimate -		-1.35
	Std. Error		0.50		Std. Error		0.48
	Sig.		0.07		Sig.		0.01
	95% Confidence Interval for Difference	Lower Bound	-1.90		95% Confidence Interval for Difference	Lower Bound	-2.31
		Upper Bound	0.07			Upper Bound	-0.39

The statistical analysis demonstrates that the ratings of perceived responsiveness and participants' patience are highly correlated, with a .75 Pearson correlation value (significant at $p < .01$). Therefore, we average the two user ratings as a single patience measurement.

The patience measurement shows a clear negative relationship with the delay level. The descending trend in the ratings with the delay levels can be instantly observed in the mean and standard deviation values in Table 4.1: 200 ms ($M = 5.59, SD = 2.14$), 500 ms ($M = 4.68, SD = 1.80$), and 2000 ms ($M = 4.24, SD = 2.43$). The between-subjects test shows that the differences in the measurements are significant with $F(2, 113) = 4.00$ under significance level ≈ 0.021. Further pairwise comparisons between different delay levels, shown in Table 4.2, reveal that users perceive a great difference between 200 ms and 2000 ms delay (Mean diff $= 1.35$, SD Error $= 0.48$, Sig $= 0.01$). Users' impatience also

increases between 200 ms and 500 ms delay with a marginal significance (Mean diff $= 0.92$, SD Error $= 0.50$, Sig $= 0.07$). However, the 500 ms and 2000 ms delays do not provoke significantly different levels of impatience (Sig > 1).

These results suggest that 1) users are impatient when using mobile apps and are sensitive to delays and 2) the developers should be careful about operations with delays larger than 500 ms.

4.4 Overall framework for poor-responsive UI detection

In this section, we first introduce the UI design problem of our interest. Then, we propose a workflow to solve the problem. Finally, we consider how our tool fits within the execution flow.

4.4.1 Problem specification

As mentioned in Section 4.2, the Android framework is specially tailored to suit the UI-driven requirements of Android apps. Developers try to keep apps responsive. A common practice when processing long-term operations is to provide a loading bar/circle animation as feedback to users. However, it requires daunting human efforts to design feedback for every possible heavy-weighted operation. In practice, developers usually only notice extremely long-term operations that may trigger ANR. However, our user study shows that a 500 ms delay is already long enough for users to perceive bad UI design.

Developers try their best to keep the app responsive. A common practice is setting loading bar/circle animation as feedback to users when processing long term operations. However, it requires daunting human efforts to design feedbacks for all possible heavy-weighted operations. In practice, in general developers merely notice extremely long-term operations that may trigger ANR. While

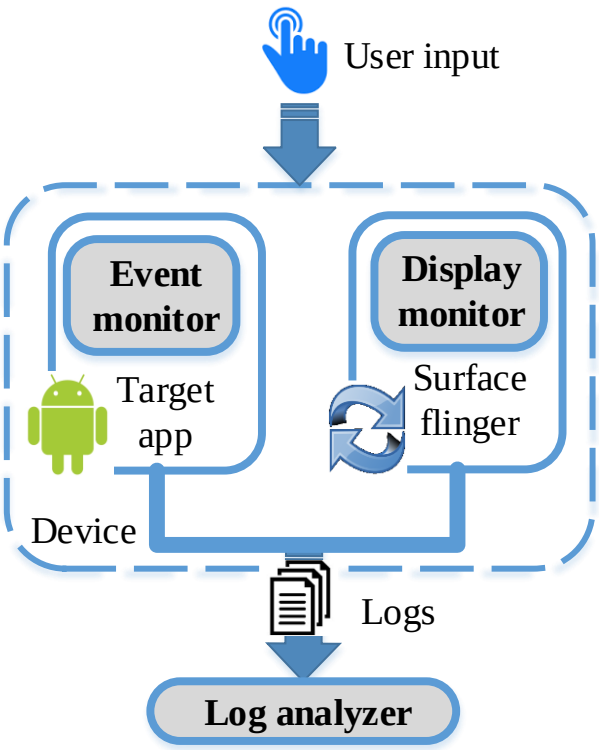


Figure 4.5: Detecting poor-responsive operations

as our user study shows, 500ms delay is already long enough for users to perceive bad UI design.

To facilitate the following discussions, we define several terms related to UI design.

Definition 4.1 (Operation feedback)**.** *A screen update that is triggered after an app receives a user operation (i.e., a button click).*

Definition 4.2 (Feedback delay)**.** *The latency between a UI operation and the first UI update that it triggers.*

Definition 4.3 (Poor-responsive operations)**.** *A UI operation is* poor-responsive *if its feedback delay is not less than a threshold T.*

In this work, we aim at detecting poor-responsive operations with feedback that takes longer than T. The UI feedback should be given as soon as the input event is accepted, but not until the event has

finished processing. The feedback can reassure users that their input being processed by the app. Without timely feedback, users are unsure whether their operations have been accepted or the touch screen is insensible. As a result, they become impatient.

4.4.2 Proposed execution flow

We propose an execution flow, shown in Figure 4.5, for detecting poor-responsive operations. First, the system takes a user input I from either a testing tool (e.g., Monkey [179], MonkeyRunner [138]) or human input. Then an *event monitor module* records the input event without interfering with the execution of the app. After the input event is processed, the display may or may not update within a preset time window. A *display monitor module* captures all of the display updates.

All of the related information is logged and analyzed offline. A *log analyzer module* analyzes the logs by calculating the feedback delays for each input event. It then generates a report about the poor-responsive operations. With the report, developers can easily detect the UI designs that should be improved.

4.4.3 Framework Design

We design a framework called `Pretect` (*P*oor-*re*sponsive UI *De*te*ct*ion) for Android apps that realizes the proposed execution flow shown in Figure 4.5. The main modules are as follows.

Event monitor

The event monitor module monitors the input events conducted on the touchscreen. Whenever an input event (*e.g.*, touch a button) is performed, the event monitor module would record the event information including the type of event, the related UI component, and the time when the input is conducted in the log.

Display monitor

The display monitor module monitors the screen updates. Whenever the screen refreshes, this module logs the UI update information including the source of the screen update (*i.e.*, the process that requests the UI update) and the update time.

Log analyzer

The log analyzer module offline analyzes the logs of the input events and screen updates. The purpose of this module is to identify the UI designs that could be improved. It reports the poor-responsive operations. The supporting information it provides in the report includes the input event information, the feedback delay, and the related logs.

4.5 Implementation details

We have implemented a poor-responsive UI detection tool based on the proposed `Pretect` framework. The implementation of the main modules in Section 4.4.3 is described in detail in this section. Details could also be found from the released source codes [37].

It is worth noting that we rely on a dynamic instrumentation mechanism to keep `Pretect` compatible with most Android versions and devices. The mechanism requires no changes to the target app *per se*. It also does not require us to recompile the underlying OS and the Android framework. Moreover, the tool requires little human effort to install and apply.

We intercept the Android framework methods in both Java and C. This approach is more light-weight and easier to implement than tracking the functions at the OS level, which typically requires heavy-weighted and sophisticated tools for kernel instrumentation. More importantly, we can thus rely on an Android-specific feature to conveniently track the relevant methods.

For Java method tracking, we note that, unlike general Linux processes, all Android app processes are created by duplicating a system process called `Zygote`. The framework binaries are loaded in `Zygote` before this duplication. Therefore, we can instrument the `Zygote` process and "hijack" the Java-based framework methods we are interested in before the app runs. When the app is running, the method invocations are inherently hijacked by `Pretect` via the forking of `Zygote`. Hence, we can easily track the methods. We implement this idea by adopting a tool called Xposed [193], which is usually used to improve the appearance of user interfaces [192]. It can substitute the original `Zygote` process with an instrumented one. We rely on its mechanism, and program our own codes to hijack the Java methods we are interested in.

For our C method interception, we note that the Android OS is based on the Linux kernel. A well-known Linux system tool named `ptrace`, which is commonly used in debugging tools (*e.g.*, gdb), is also available on Android. `Ptrace` makes it possible to inspect the child process of the parent process. `Ptrace` enables the parent process to read and replace the value of the register of the child process. We can utilize `ptrace` to attach code to a target process (with a known process ID `pid`). Then, we are able to take over the execution of the target process. By analyzing the elf-format library files of the target process, we can locate the memory addresses of the methods with the relevant names and invoke them accordingly. Therefore, it is feasible to invoke the `dlopen`, `dlsym` library-related system calls of the target process. We implement the idea by adopting a tool called LibInject [120].

4.5.1 Event monitor

We implement the event monitor by instrumenting the related Android framework Java methods to obtain the input event information. In particular, we intercept several event dispatch methods of the

1. … 2365 …: com.cyberlink.youperfect[Event]com.cyberlink.youperfect.widgetpool.common.ChildProportionLayout{425193c8V.E...C....P....270,0-540,67#7f0a051dapp:id/cutout_tab_artistic}_null-Motion-UP : 125696

2. … 138 …: BIPC:****android.gui.SurfaceTexture****, sender_pid:2365, UptimeMilli: 127932

Figure 4.6: Example logs of `Pretect`

`View` class; all touchable widgets such as Buttons, ImageView and ListView are subclasses of the `View` class. We carefully select a set of methods to cover all types of possible input events. The methods include `dispatchKeyEvent`, `dispatchTouchEvent` and some rarely used methods such as `dispatchKeyShortcutEvent`, `dispatchKeyEventPreIme` and `dispatchTrackballEvent`.

A sample log of an event is shown in line 1 of Figure 4.6. From this line we can see that the input event is a touch event, as indicated by the *Motion-Up* action. The touch event is conducted on a `ChildProportionLayout` with the ID *cutout_tab_artistic*. With the ID information, we can locate the component easily via the Hierarchy Viewer [95] tool published along with Android SDK. The event is performed 125696 ms after the system is booted. The highlighted information is important for the subsequent steps of the analysis.

4.5.2 Display monitor

There are numerous functions that can update Android app displays (*e.g.,* `TextView.setText`, `ImageView.setImageBitmap`). Therefore, it is hard to list and instrument all of them. Moreover, updating UI display is a cross-layer procedure. Multiple layers created by the app, Android framework, kernel, and driver are involved in the procedure. Many components such as `SurfaceFlinger`, `OpenGL ES`, and `FrameBuffer` are included. The complicated nature of the UI display update mechanism makes

it hard to trace. Luckily, we find that all of the UI updates are done via the `surfaceflinger` process provided by the Android OS. All of the UI update requests from the app are sent to the `surfaceflinger` process via `binder`, which is the standard inter-process communication (IPC) mechanism in Android.

Therefore, we can obtain the UI update information by intercepting the communication related functions of `binder`. More specifically, we intercept the `surfaceflinger` process on `ioctl` method of the shared library `libbinder.so`, which the binder mechanism is embedded in. The `ioctl` method is responsible for reading and writing the inter-process communication contents. We successfully intercept the invocations of `ioctl` to get detailed information about the UI update requests.

There are hundreds of `binder` communication messages per second; the UI updating messages are the one type of request that `SurfaceTexture` sends from the app under test (`pid` 2365 in this example). We show a UI update message in line 2 of Figure 4.6. The corresponding UI request time is 127932 ms after the system is booted.

4.5.3 Log analyzer

The log analyzer extracts information from the logs collected during the offline tests. We implement the analyzer in Python.

The most important task of the log analyzer is to correlate the input events with their associated UI updates. The log analyzer first scans the logs to retrieve the input events. For each input event I, we check the following `binder` requests set ($\mathbb{R}$), until it reaches the next input event. With the information about the process ID (`pid`) of the sender process (*e.g.*, we record the sender's `pid`, as shown in the second line in Figure 4.6); the analyzer is able to distinguish the source of the `binder` request. We then search for the first `binder` request from set $\mathbb{R}$ such that 1) the sender's `pid` is the same as that

of the app process (which we can determine from logs, for example, line 1 shown in Figure 4.6); and 2) the requested component is `SurfaceTexture`, which means it is a UI update request. We regard this UI update request as feedback of the input event I. Then it is not hard to calculate the feedback delay which is the time span between the input event and the feedback.

Worse than long feedback delay, some operations may have no feedback at all. For these operations, there are no following UI update logs. We filter such events by calculating the interval between the input event of interest and the following event. If the interval is too large, we regard the event as having no feedback. The log analyzer reports poor-responsive operations as these input events without feedback (*i.e.*, no following UI update request by the app) or without timely feedback (*i.e.*, feedback delay $\geq T$).

From lines 1-2 in Figure 4.6, we can infer that: 1) the two lines are a relevant input event and its first UI update; and 2) the feedback delay of the input event is $127932 - 125696 = 2236$ ms.

4.6 Experimental study

In this section, we first conduct several experiments on synthetic apps and open source apps to show the effectiveness of our tool. Then we illustrate how our tool improves the UI designs by presenting several case studies.

4.6.1 Tool effectiveness validation with fault injection

We examine the accuracy of `Pretect` by evaluating ten apps, including five synthetic benchmarks and five open source apps. These apps are selected to represent those with common heavy-weighted UI operations that may incur long UI latency. The apps and the selected operations are listed in Table 4.3.

As mentioned in Section 4.2, various types of asynchronous tasks

Table 4.3: List of applications

Type	Name	Operation
Synthetic	Worker Thread	load an image by worker thread
	AsyncTask	load an image by AsyncTask
	ThreadPool-Executor	load an image by ThreadPool-Executor
	HandlerThread	dump a table of a database
	IntentService	load an image by IntentService
Open source	K9Mail	refresh email Inbox
	ASqliteManager	dump a table of a database
	AFWall+	start network firewall
	Amaze	scan disk for all images
	ePUBator	convert a pdf file to epub type

and UI updates are common sources of poor-responsive operations. We implement all the five Android asynchronous mechanisms for this purpose, including Worker Thread, AsyncTask, ThreadPoolExecutor, HandlerThread, and Intent Service. The app loads an image after an asynchronous task finishes. We use sleep method to ensure the asynchronous tasks finish in about 500 ms. To validate `Pretect`'s ability to detect poor-responsive operations, we update the UI for each type of asynchronous mechanisms with two settings: separately with timely feedback and without timely feedback. More specifically, for the setting with timely feedback leading to responsive operations, we set a loading circle to appear while the asynchronous tasks are executing. For the setting without timely feedback, leading to poor-responsive operations, we do nothing when the asynchronous tasks are executing.

We validate our tool on open source projects. We select five open source projects that have representative heavy-weighted UI operations that may incur long UI latency, as listed in Table 4.3. These operations may incur UI delays from various sources such as network requests, database operations, system settings, disk scanning with querying content provider, and CPU-intensive computing.

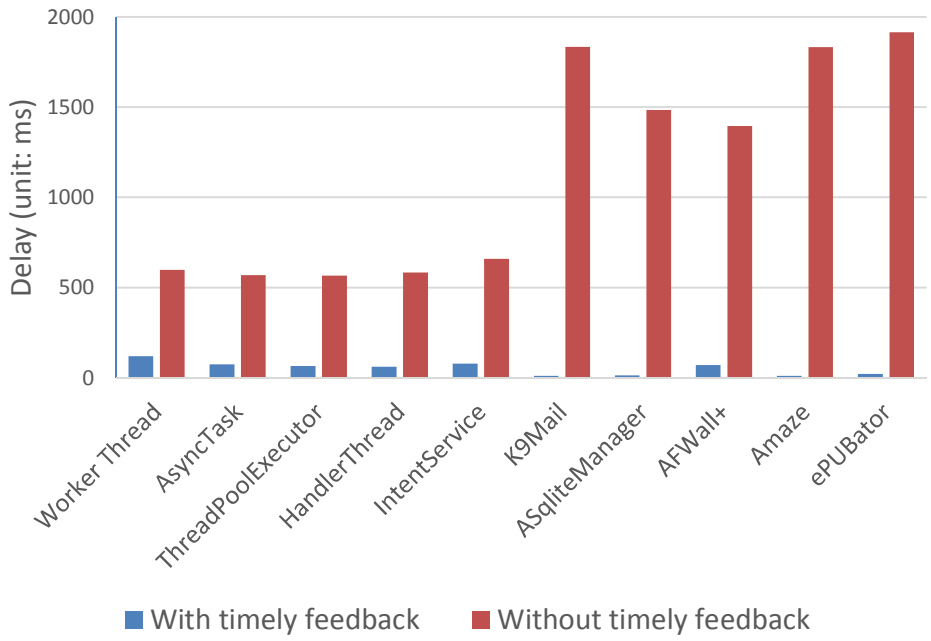


Figure 4.7: Feedback delay of applications detected by `Pretect`

The original apps offer good timely UI feedback on these long-term operations. During the experiment, we manually switch off the UI feedback for comparison.

The results presented in Figure 4.7 show a notable difference between poor-responsive operations and responsive operations. `Pretect` can easily distinguish responsive operations from poor-responsive operations.

4.6.2 Overview of Experimental Results

We apply the tool to 115 popular Android apps covering 23 categories (including BooksReferences, Photography, Sports, etc.). We download apps on AndroidDrawer [35] from all of the categories except the library demo category. We randomly select five popular apps from each category. We conduct the experiments on Huawei G610-T11 with Android 4.2.2.

The overall statistics for the case where the threshold is 500 ms

Table 4.4: Statistics of feedback delay

App Statistics		Issues Statistics	
# apps contain bugs	94	Total	327
Max bugs an app	23	Max	29189.0 (ms)
Min bugs an app	0	Min	504.0 (ms)
Avg. bugs per app	2.8	Avg	1603.9 (ms)
Median bugs per app	2	Stdv	2635.5

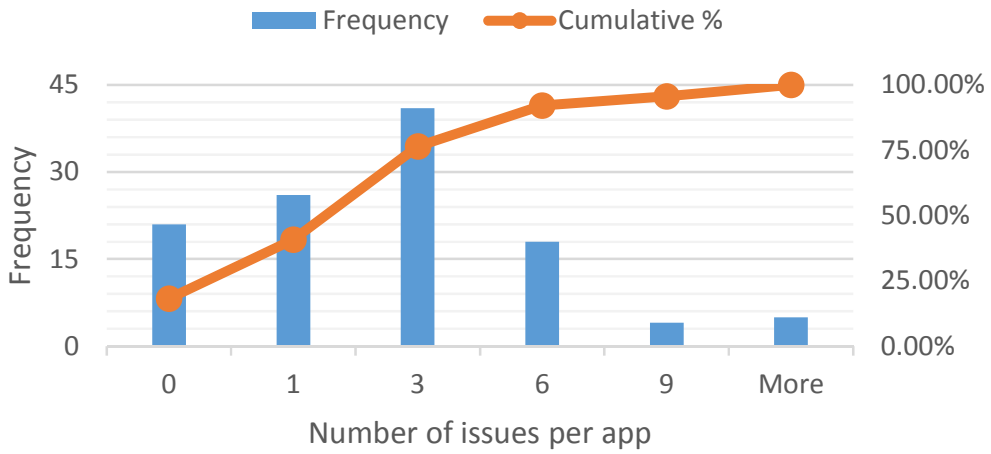


Figure 4.8: Cases number distribution

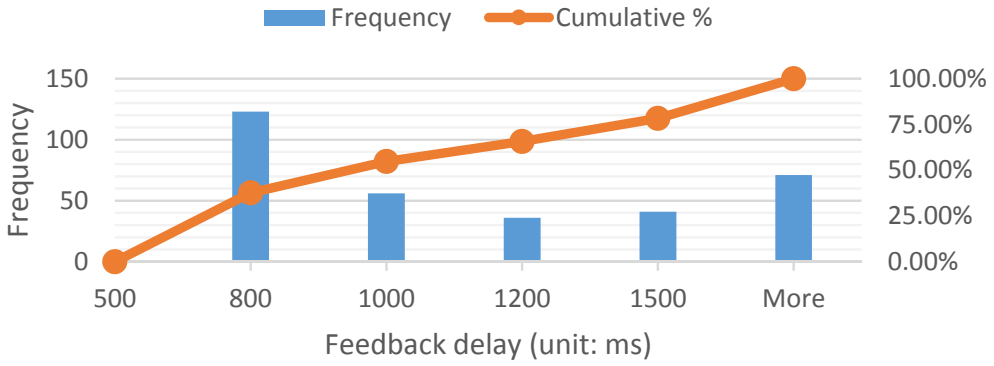


Figure 4.9: Feedback delay distribution

are shown in Table 4.4. We find that poor-responsive operations are common defects in UI designs. Of the 115 apps examined, 94 contain potential UI design defects. The maximum number of defects in a single app is 23 and the minimum number is 0. On

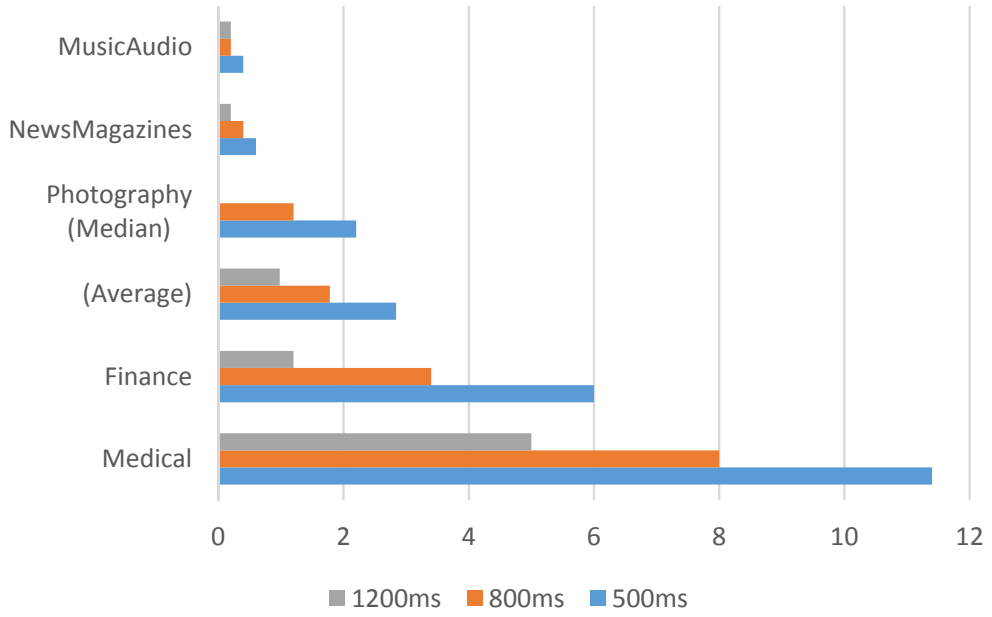


Figure 4.10: Avg. # of cases per category with different thresholds

average, there are 2.8 (median 2) defects per app. Long feedback delays are common. We find in total 327 independent components with feedback delays larger than 500 ms. The maximum delay is larger than 29 seconds. The distributions of the number of defects per app and UI feedback delays are shown in Figure 4.8 and 4.9.

Developers can define their preferred threshold for poor UI designs according to the category of their app. Figure 4.10 demonstrates the correlation of the number of bad components per category with the threshold cutoff. We also note the large differences between apps in different categories. For example, Medical apps contain the most number of bad UI design components (avg. 11.4 bad components), followed by Finance apps (avg. 6). Music Audio contains the least number of bad UI design components (avg. 0.4), followed by News Magazines (avg. 0.6). Moreover, the threshold for feedback delay is a key factor for poor-responsive operations detection. According to the user study, 500 ms is a reasonable choice. Nevertheless, developers could choose their own threshold based on their own criteria. We have chosen 500 ms, 800 ms,

Table 4.5: Top 10 components contain poor-responsive operations

Component	Issues found
android.widget.Button	78
android.widget.ListView	30
android.widget.ImageButton	22
android.widget.EditText	21
android.widget.ScrollView	20
android.widget.RelativeLayout	16
android.widget.ImageView	13
android.widget.TextView	10
android.widget.LinearLayout	10
android.support.v7.widget.Toolbar	7

1200 ms respectively as the thresholds for our tests.

We also investigate the top-ranked components that commonly suffer from poor responsiveness. They are shown in Table 4.5. When designing these UI components, developers must take special care to ensure their responsiveness.

Next, we select three representative cases to show how `Pretect` contributes to delay tolerant UI design.

4.6.3 Case Study 1: YouCam

YouCam Perfect - Selfie Cam (`http://www.perfectcorp.com/#ycp`) is a popular selfie application. YouCam Perfect helps users to create better images via customized filters. The Android app has had more than 60 million downloads.

We apply `Pretect` to test Youcam Perfect, version 4.10.1. A representative testing scenario is shown in the "Steps to trigger bug" section of Figure 4.11. A portion of the report generated by `Pretect` is shown in the lower section of Figure 4.11. We test on the *cutout* functionality of Youcam Perfect; this process cuts out a piece of an image and attaches it to a pre-defined template. As can be seen from the report, the `ChildProportionLayout` with ID

1. Open YouCam Perfect app
2. Open “Cutout” function
3. Click “Artistic” tab
4. Click “Fun” tab
5. Select the first template
6. Select the first image from Image library
7. Click “tick” image
8. Random draw a line to select range
9. Click “tick” image
10. Switch to “Artistic” tab
11. Switch to “Fun” tab
12. Select the first cover
13. Click the “save” text

Steps to trigger bug

Operation: com.cyberlink.youperfect.widgetpool.common.ChildProportionLayout cutout_tab_artistic Click
No screen update delay: 2236 ms
Related logs:
2365 2365 D RefreshMon: com.cyberlink.youperfect[Event]com.cyberlink.youperfect.widgetpool.common.ChildProportionLayout{425193c8V.E...C....P....270,0-540,67#7f0a051dapp:id/cutout_tab_artistic}_null-Motion-UP : 125696

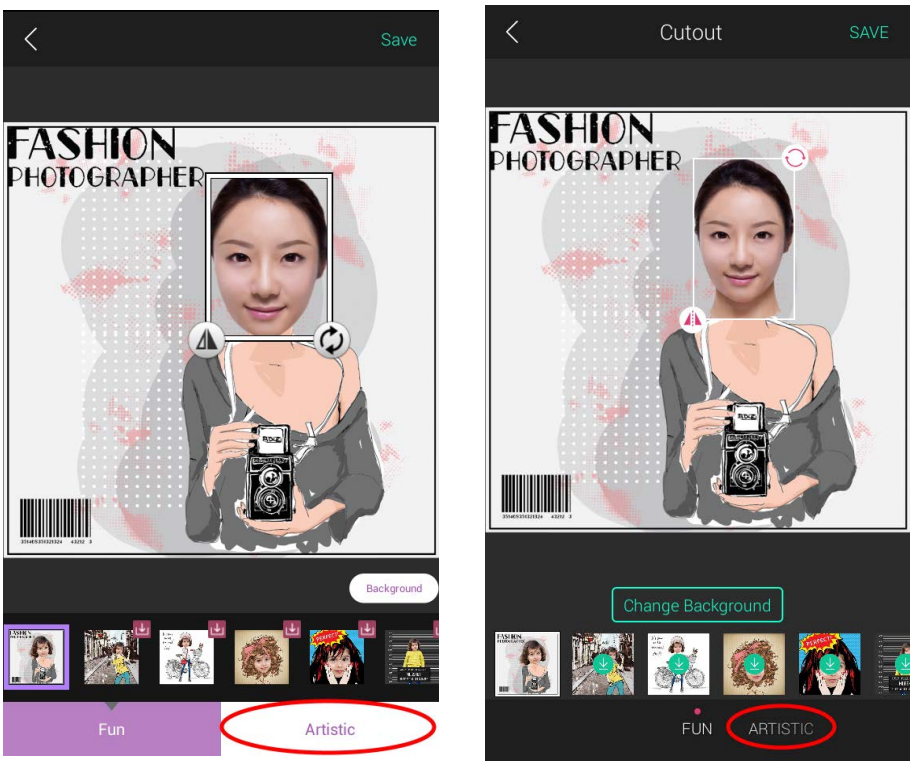
138 138 D RefreshMon: BIPC:****android.gui.SurfaceTexture****, sender_pid:2365, UptimeMilli: 127932

Report

Figure 4.11: Selected report of YouCam Perfect

“cutout_tab_artistic” is problematic. We detect a delay of 2236 ms without UI updates after the button is clicked. This decreases the quality of the software and hurts the user experience. This issue is the only reported issue, which we can immediately identify the defect. Via simply searching the component ID “cutout_tab_artistic” with Hierarchy Viewer [95], we successfully locate the problematic component, circled in Figure 4.12a. The component is the “Artistic” tab line 10 of the test scenario. By repeating the test scenario manually, we observe the latency that occurs without a feedback.

We have reported our findings to Perfect Corp. The leader of the



(a) Version 4.10.1

(b) Version 5.4.1

Figure 4.12: Screenshots of Youcam Perfect

YouCam Perfect development team has provided us with positive feedback on our results: "With your hint, we find that we have used a widget which tends to be slower. We will fix as soon as possible." In a later version of Youcam Perfect 5.4.1, the UI design has been modified, as shown in Figure 4.12b. A test run with `Pretect` shows that this modification has reduced the feedback delay to below 100 ms.

4.6.4 Case Study 2: Bible Quotes

Bible Quotes (`http://www.salemwebnetwork.com/`) is a popular Bible app that provides verses from the Bible. Users can lookup verses in the Bible, save favorites, share quotes with others, etc. The app has had more than a million downloads on Google Play.

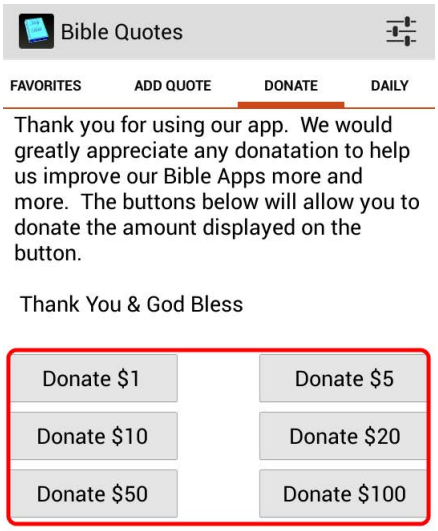
1. Open Bible Quotes app
2. Select “Donate” tab
3. Click “Donate $1” button
4. Click “Donate $5” button
5. Click “Donate $10” button
6. Click “Donate $20” button
7. Click “Donate $50” button
8. Click “Donate $100” button

Steps to trigger bug

Operation: Button btnDonate01 Click
No screen update delay: 2073 ms
Related logs:
2519 2519 D RefreshMon: com.dodsoneng.biblequotes[Event]android.widget.Button{41a3da 0VF.D..C.....0,300-225,372#7f09005aapp:id/btnDonate01}_"Donate $1"-Motion-UP : 78535

2519 2519 D RefreshMon: com.dodsoneng.biblequotes[Event]android.widget.Button{41b63330 VF.D..C.....315,300-540,372#7f09005bapp:id/btnDonate02}_"Donate $5"-Motion-DOWN : 80608

Report



Screen Shot

Figure 4.13: Selected report of Bible Quotes

We apply `Pretect` to test Bible Quotes, version 4.9. A representative testing scenario is shown in the “Steps to trigger bug” section of Figure 4.13. A portion of the report generated by

`Pretect` is shown in the middle section of Figure 4.13. We test the functionality of the *donation* process of Bible Quotes. As can be seen from the report, the `Button` with ID "btnDonate01" is problematic. Note that this is the first issue in our generated report (other issues are about other donate buttons which are similar to this one). Via searching the component ID in Hierarchy Viewer, we immediately locate the button of the app. We find that there are no UI update-related logs generated between the time the "btnDonate01" button is clicked and the time the "btnDonate02" button is clicked (corresponding to the test scenario steps 3-4). This means that the UI does not update for more than 2 seconds after the "btnDonate01" button is clicked. This clearly decreases the quality of the app. We further investigate the app to find the relevant information about the UI component. The logs reveal that the text on the button is "Donate $1"; the component is circled in the lower section of Figure 4.13. The component we are interested in is the "Donate $1" button on line 3 of the test scenario.

4.6.5 Case Study 3: Illustrate

Illustrate - The Video Dictionary (`http://www.mocept.com/illustrate/`) is an effective app for teaching new words and their meanings. It provides videos with context and real-life examples allowing the learner to grasp definitions and usage with ease. ICAL TEFL, a leading provider of English language courses, says that it is fun and will help students. The University of Michigan Campus Life News recommend it as the best app in "7 Helpful Study Apps for GRE, LSAT, and GMAT Preparation"

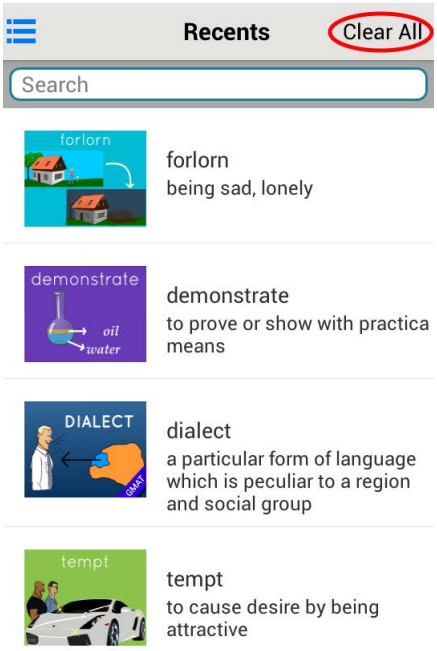
We apply `Pretect` to test Illustrate, version 1.2.7. A piece of the testing scenario is depicted in the upper section of Figure 4.14. Part of the report generated by `Pretect` is shown in the middle section of Figure 4.14. We test the *history clearing* functionality of Illustrate. As can be seen from the report, the `TextView` with

1. Open Illustrate - The Video Dictionary app
2. Click on the first word
3. Click on the *first word* of Explore More Words
4. Repeat step 3 for 3 times
5. Click “Back” button on the top left
6. Click “menu” image on the top left and select “Recents” item
7. Click “Clear All” button

Steps to trigger bug

Operation: TextView clearall Click
No screen update delay: 2351 ms
Related logs:
2319 2319 D RefreshMon: com.mocept.illustrate[Event]android.widget.TextView{41c9c9f0V.ED..
C....P....0,17-100,54#7f09011fapp:id/clearall}_"Clear All"-Motion-UP : 181382
138 138 D RefreshMon: BIPC:****android.gui.SurfaceTexture**********, sender_pid:2319,
UptimeMilli: 183733

Report



Screen Shot

Figure 4.14: Selected report of Illustrate

ID “clearall” is problematic. Note that this is the only issue in our generated report. Via searching the ID in Hierarchy Viewer, we

immediately locate the component. We detect a delay of 2351 ms without UI updates after the `TextView` is clicked. This decreases the quality of the software and tests users' patience. We further investigate the app to find the relevant information about the UI component. An examination of the logs reveals that the text of the `TextView` is "Clear All", which is circled in the lower section of Figure 4.14. The component of interest is the "Clear All" button in line 7 of the test scenario. By repeating the test scenario manually, we observe the latency without feedback.

4.7 Discussions

Our user study results show that more than 500 ms delay can cause the users become impatient. To conduct the survey, we choose most active mobile users (between 20 to 27 years old [8]) and common usage scenarios (*e.g.*, video, audio, map). Users may tolerant 500 ms delay in other scenarios. Whereas, our focus is on revealing the relationship between user patience and UI latency, rather than obtaining the exact threshold of user impatience. We have shown in Section 4.6.2 that the threshold is just a parameter of `Pretect`which can be easily set to suit the requirement of developers.

Current approaches cannot properly detect poor-responsive operations. Approaches including StrictMode [90] and Asynchronizer [123] can detect operations that blocks the UI, but fail to detect asynchronous tasks that do not provide feedback. Other approaches (*e.g.*, Appinsight [161], Panappticon [196]) define a delay as the time interval between the initiation of an operation and the completion of all of the triggered tasks. These approaches aim to detect the abnormal execution of asynchronous tasks rather than feedback delay. In such cases, using `Pretect` to detect feedback delay is a better approach to improving the responsiveness of UI design. Moreover, these approaches may report some long-

term background tasks (*e.g.,* download) as suspicious, as a result of focusing on the life cycle of the tasks. However, this kind of task does not update the display when finished. They are less relevant to the responsiveness. In comparison, `Pretect` pays attention to whether the app notifies the users that the operations that will trigger the background tasks have been accepted.

The screen refreshes even when the app UI does not update, as the action bar on the top of the screen also refreshes. However, the action bar is not our focus. Therefore, intercepting low level system functions related to full screen display refresh such as `eglSwapBuffers` in /system/lib/libsurfaceflinger.so is not a good choice. We further inspect the related screen update procedure and find `binder` used in the process. We then intercept `binder` and measure the screen update time of the app more precisely with the logged `pid`.

Even with the action bar updates filtered out, we are not guaranteed to obtain the right user intended UI update after one user operation. Because UI updates triggered by periodical updating, advertisement banners, or previous long-processed tasks may affect our detection on first UI updating after user operation. As validated in Section 4.6.1, cases reported by `Pretect`, the first tool for detecting poor-responsive operations, are all poor-responsive that need to be taken care with. The improvement of false-negative rate could be leave as our future work. A possible solution is to study the user concentration and app design to understand the user intended UI update after a user operation.

4.8 Summary

In this chapter, we discuss the problem of responsive UI design in Android apps. We motivate the problem of detecting poor-responsive operations by conducting a user survey. The survey results show that users' patience is correlated with UI responsiveness.

We design and implement a tool called `Pretect` that can detect poor-responsive operations. The tool is shown to work correctly on synthetic benchmarks and on open source apps. We further verify `Pretect` with real-world case studies. The results demonstrate the effectiveness of `Pretect`.

□ **End of chapter.**

Chapter 5

Deployment of Single Cloud Service

Single Service Deploy

Making optimal server deployment of cloud services is critical for providing good performance to attract users. The key to making optimal deployment is to know the user experience of end users. With such knowledge, we then model and solve the optimal deployment problem. We list the points of this chapter as:

- Propose a framework to model cloud features and capture user experience.
- Formulate the optimal service deployment considering user experience; propose approximation algorithms.
- Evaluate the proposed model solving method on real-world dataset; public release the dataset and simulation code for repeatable experiments.

5.1 Background and Motivation

In cloud computing systems, computation, software, and data access can be delivered as services located in data centers distributed over the world [39, 155]. Typically, these services are deployed on instances (*e.g.*, virtual machine instances) in the cloud data centers. Recently, numerous systems have been implemented with the cloud computing paradigm (*e.g.*, Amazon Elastic Compute Cloud (EC2) [5]).

In the emerging cloud computing systems, *auto scaling* and *elastic load balance* are keys to host the cloud services. *Auto scaling* enables a dynamic allocation of computing resources to a particular application. In other words, the number of service instances can be dynamically tailored to the request load. For example, EC2 can automatically launch or terminate a virtual machine instance for an EC2 application based on user-defined policies (*e.g.*, CPU usage) [4]. *Elastic load balance* distributes and balances the incoming application traffic (*i.e.*, the user requests) among the service instances (*e.g.*, the virtual machine instances in EC2 [13]).

Auto scaling and elastic load balance directly influence the Internet connections between the end users and the services as they essentially determine the available service instance for an end user. Hence, they are important to the user experience of service performance.

Unfortunately, current auto scaling and elastic load balance techniques are not usually optimized for achieving best service performance. Specifically, typical auto scaling approaches (*e.g.*, that adopted in EC2 [4]) cannot start or terminate a service instance at the data center which is selected according to the distributions of the end users. For example, when the number of users increases dramatically in an area, a new instance located far away, instead of nearby, may be activated for serving the users. Furthermore, elastic load balance generally redirects user requests to the service

instances merely based on loads of the instances. It does not take the user specifics (*e.g.*, user location) into considerations. As a result, a user may be directed to a service instance far away even if there is another available service instance nearby.

In this chapter, we model the features of user experience mainly by latency (other features can be extended in the framework) in cloud service. After that we address these issues by proposing a new user experience-based service hosting mechanism. Our mechanism employs a service redeployment method. This method has two advantages:

1. It improves current auto scaling techniques by launching the best set of service instances according to the distributions of end users.

2. It extends elastic load balance. Instead of directing the user request to the lightest load service instance, it directs user request to a nearby one.

The prerequisite of such a service hosting mechanism is to know the user experience of a *potential* service instance, before we choose to activate the instance and deliver the user requests to it. This is quite a challenging task, as there is generally no proactive connection between a user and the machine that will host the service instance. Measuring the user experience beforehand is hence impossible. We notice that the user experience of a cloud service depends heavily on the communication delay between the end user and the service instance the user accesses, which is mainly caused by the Internet delay between the user and the data center hosting the instance. We therefore propose a viable method to conveniently measure and predict such an Internet delay.

With the predicted user experiences, the service hosting problem is essentially how to redeploy a set of data centers for hosting the service instances, while guaranteeing the user experience for frequent users. We formulate it as a *k-median* problem and a

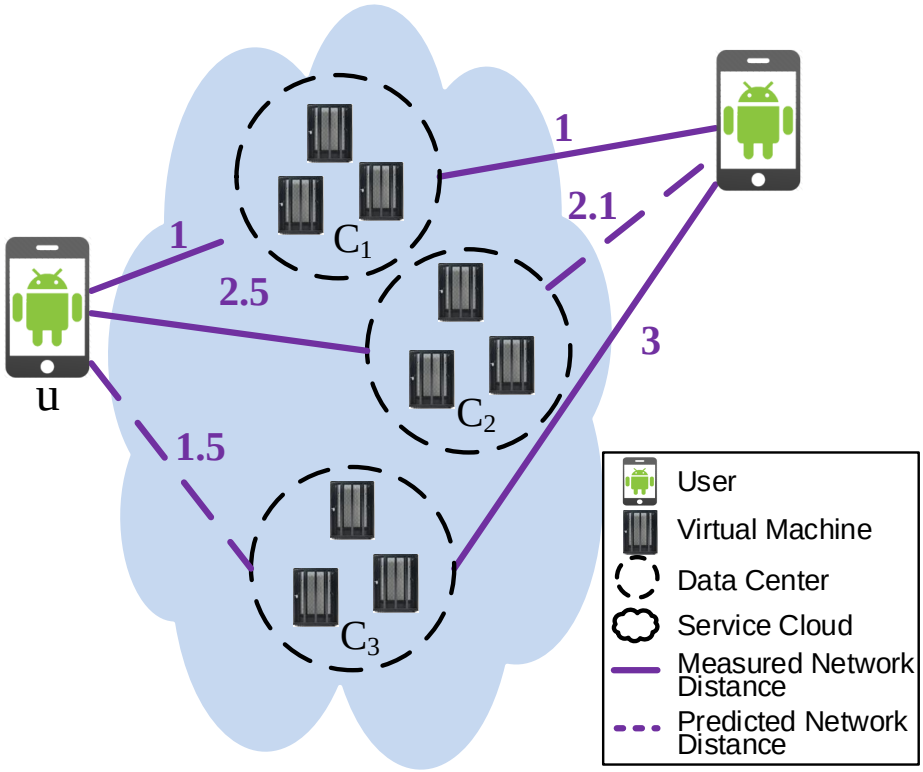


Figure 5.1: Framework of cloud-based services

max k-cover problem, which can be efficiently solved by several algorithms we proposed in this chapter. We evaluate our service hosting mechanism based on a large set of real-world data (roughly 130,000 accesses to Internet-based services). The results demonstrate that our mechanism can approach rapid and scalable cloud service hosting.

5.2 Overview of Cloud-Based Services

5.2.1 Framework of Cloud-Based Services

Figure 5.1 shows the framework of cloud-based services. A cloud contains several data centers (eclipse in Figure 5.1). Physical machines are virtualized as instances in the data center. Service

providers would deploy service running on these instances. An end user normally connects to the cloud to get data and run applications/services. User requests are directed to the service instances.

A good example is the Chrome OS developed by Google. Via such kind of light client, the end user can access data as well as application logic provided by the cloud as services.

Since the instances are inside the cloud, the connection information especially Round Trip Time (RTT) between a user and an instance can be maintained by the cloud provider. The solid lines in Figure 5.1 represent RTT information being recorded between a user and some instances. Some links are not used thus the related RTT information is missing. We can derive new methods to predict these values. Dash lines in Figure 5.1 stand for this situation.

Generally user experience contains three elements: the Internet delay between user and cloud data center, the delay inside a data center in the cloud, and the time to handle the service request. As machines in a data center are typically connected by gigabit links, delay inside a data center can be ignored. Moreover, the time to handle the service requests is only affected by the computing ability of a service instance. As a result, the processing time is almost the same for two service instances. Hence, the user experience is mainly determined by the Internet delay.

5.2.2 Challenges of Hosting the Cloud Services

In order to attract users by low latency, service providers are concerning about where to deploy service instances in the cloud. The challenge of hosting the cloud service in the cloud comes from the difficulties of foreseeing user experience before actually running the service. So normally redeployment is required after knowing the distributions of users better.

After the service operating for a period, the Internet delay between users and every cloud data center can either be measured

or be predicted. We describe this in Section 5.3. This means in the cloud we can obtain all information regarding the potential positions for deploying service instances, while many existing computing infrastructures such as Internet services do not have such a feature. All the information is organized as a distance matrix. An element in the matrix is the distance value between a user and a data center.

Moreover, we notice the fact that the number of data centers is limited, while there is no bound on the number of services. So there are multiple services deployed in the data centers of a cloud. This fact suggests that we can use this measurement for optimizing any service in the cloud.

Consequently, we employ a distance matrix to formulate the redeployment problem as a *k-median* problem in Section 5.4.1. However, we have not taken the limitation of resource for a single service instance into consideration. Another model of *max k-cover* problem is engaged to deal with this limitation and to make the model more realistic. We discuss this formulation in Section 5.4.2. Through solving these problems we can redeploy the service instances intelligently.

The fact is that the physical machines are not required to migrate when redeploying service instances in the cloud. Generally we store the data of a service instance in the cloud storage and to redeploy the instance we could load the image to another machine in the cloud. If the new service instance for redeploying lies in another data center of the original one, the loading overhead is huge and this is usually the case. Since the major part of the users does not vary frequently, we defeat this problem by running our redeployment approach periodically (the period can be set long such as one or two days). Another method is to redeploy the service instances one by one. Then when one instance is under redeploying the other instances could take over the users connecting to the instance and split the workload.

5.3 Obtaining User Experience

5.3.1 Measure the Internet Delay

A user request of service in the cloud is responded by an instance inside the cloud. Therefore the cloud provider is able to record the round trip time (RTT) from the user to the instance. This RTT value is kept as the distance d_{ij} from the user i to the data center j as the delay inside a data center can be ignored. The user generally calls several services and the related service instances are distributed in different data centers throughout the cloud. As a result, we can get plenty of distance values between different users and data centers.

5.3.2 Predict the Internet Delay

A user may not get a response from VM instances deployed in every data center. Therefore we cannot get distance data between each user and every data center directly which means the measured distance matrix is quite sparse. We call it a missing value if the distance d_{ij} between a pair of user and data center (i, j) is not available. The task is to fill in the missing values in the distance matrix. The technique given in [131] can be used to predict the missing values. The idea goes as the following. Some users may come from the same place and use the network infrastructure of the same provider, thus their network performance is similar. We can examine the existing values to find these similar users of user i. In the computing, Pearson Correlation Coefficient is employed to define the similarity between two users based on the distance to data centers they visited in common. As these similar users may get the response from data center j already, we combine distance values between similar users and data center j intelligently to predict the missing distance value d_{ij} between the pair of user and data center (i, j). Technical details can be found in the work [131].

Table 5.1: Alphabet of problem model

Descriptions	Notation
Number of data centers	M
The set of data centers	Z
Number of frequent users	N
Number of instances to deploy	k
Distance between user i and data center j	d_{ij}

5.4 Redeploying Service Instances

Suppose a service provider p will provide a service s in the cloud, and suppose the service will be distributed on k instances in the cloud among totally M data centers (the set of Z) due to budget restriction. At first, the cloud service hosting mechanism h can only guess where to place the k instances. But after a period of running service s, h knows a bunch of (in total N) users who use s frequently. The k instances can then be redeployed.

With this setting, the problem is to redeploy k instances such that the result is optimal for the current N frequent users by considering the distance matrix we obtained in Section 5.3[1]. The notations we used in the following sections are first listed in Table 5.1.

5.4.1 Minimize Average Cost

Suppose for a specific user u who would like to take the service s, our mechanism would direct the user to the closest one of the k instances (in terms of network distance). We define *cost of user u* as the distance between u and the closest instance. Our objective is to minimize the average cost of N users. Note that N is fixed and the

[1]The network link may vary in different time. Average value of distance can be utilized in the algorithm. While we are not interested in the exact value of distance, our task is to deploy service instance. As long as the order of distances between a user and different data centers is preserved, our choice of data center is fine.

target is equivalent to minimize the total cost.

We formulate this problem as the followings:

Given

$Z =$ the set of data centers

$C =$ the set of users

$d_{ij} =$ distance between every pair $i, j \in C \times Z$

Minimize:

$$\sum_{i=1}^{N} \min_{j \in Z'} \{d_{ij}\}$$

Subject to:

$$Z' \subset Z$$
$$|Z'| = k$$

This is exactly the well known *k-median* problem, which is NP-hard. So we resort to the following fast approximation algorithms.

Brute Force

In a small scale (*e.g.*, select 3 instances from M potential data centers), it is possible to list all combinations. We call this *brute force* algorithm in our experiment in Section 5.5. The complexity of this algorithm is $O(M^k \cdot N)$, where M, N and k follow the definition in Table 5.1. If k is small, it can be calculated in reasonable time.

Greedy Algorithm

Greedy algorithm runs as follows. Suppose we would choose k among M data centers to deploy the instances. In the first iteration, we evaluate each of M data centers individually to determine which one is the best to be chosen first. We compute the average distance from each data center to all users. The one achieving the smallest average cost will be chosen. In the second iteration, we search for a second data center. Together with the first data center we have

Algorithm 5.1 Local search algorithm for k-median problem

1: $S \leftarrow$ an arbitrary feasible solution.
2: **for** each $s \in S, s' \notin S$ **do**
3: **if** $cost(S - s + s') \leq (1 - \frac{\epsilon}{p(N,M)}) * cost(S)$ **then**
4: $S \leftarrow S - s + s'$
5: **end if**
6: **end for**
7: **return** S

already chosen, the two data centers yield the smallest average cost. We do the iteration until k data centers are chosen.

Local Search Algorithm

Local search can provide the current best known bound for approximating *k-median* problem [40].

The idea is not complicated, and Algorithm 5.1 shows the approach. $Cost(S)$ in the algorithm means the average cost for all users. $s \in S, s' \notin S$ are two sets containing the same number of elements. $P(N, M)$ is a polynomial in M and N. The constraint $cost(S - s + s') \leq (1 - \frac{\epsilon}{p(N,M)}) * cost(S)$ is to guarantee that the algorithm terminates in finite steps.

Assume sets s and s' are of size t. It is easy to verify that the algorithm runs in $O(l \cdot k^t \cdot M^t \cdot N)$ time where l is related to ϵ, and other notations are given in Table 5.1.

On initializing this algorithm we use mainly two methods. The first one we utilize is the data centers selected by the greedy algorithm to initialize, and the second one we use is a random vector. As local search would naturally find the local optimum, output of the algorithm is always no worse than that of the original one. So the one initialized by the greedy algorithm would return a better or at least equivalent solution to the greedy result. We call this approach *greedy init + single swap*, as we swap only one element in local search algorithm.

Random Selection Algorithm

Random selection algorithm would randomly choose k out of M data centers with equal probability. Every data center has the same possibility to be chosen. This is a simple algorithm. Generally the method to improve the performance of a random selection algorithm is to run it multiple times. The purpose of this algorithm is to designate it as a base line of performance. So instead of running random selection algorithm at fixed times, we determine the times of running dynamically. For example, if we would like to know how good the greedy algorithm is in terms of time complexity and approximation rate, we would use a dynamic random selection algorithm for comparison. We record the run time of the greedy algorithm T_{greedy} and launch the random selection algorithm running several times with the total running time roughly equal to T_{greedy}. Consequently we could compare either their relative performance to the optimal choice or their average distance to all users directly. If the result is comparable then we argue that the greedy algorithm is not a good algorithm as its performance is no better than the random selection algorithm while the approach is more complicated, and vice versa.

In our experiment we also use the random algorithm to evaluate the local search algorithm.

5.4.2 Maximize Amount of Satisfiable Users

In the previous section, we set our target to minimize the average cost of all users. Through later experiments we confirm the *greedy init + single swap* algorithm is good enough in terms of both result and time complexity to solve the *k-median* problem. This makes the redeployment simple; however, there is a limitation of the *k-median* model. Recall on formulating redeployment in the *k-median* problem we have assumed users are evenly distributed and all users can request service from their nearest service instance. In

practice, this condition does not always hold. All the users request to the nearest service instance would cause some service instances overload in real world cases.

It is reasonable to think that we are deploying the instances in the cloud and the cloud has virtually infinite resource. However, in reality we only get limited budget and we can only launch limited (say, k) instances in the cloud. If we want to ease the burden of some overload instances, we are required to put some extra instances in the same data centers. As a result, we usually distribute service instances in k' (where $k' < k$) data centers.

Moreover, in some cases part of the users may be extremely far away from most of the data centers. While considering minimizing the average cost, these users have a tendency to force some service instances deployed in the data center close to them. This kind of users is called outlier in [56]. Though the model of k-median problem with outliers in [56] can successfully deal with outliers, it has two drawbacks. First, it does not suit for the huge amount of data (*i.e.*, thousands of users and hundreds of candidate locations). Second, it does not limit the number of users a service instance can serve. So in this model some instances may overload.

In this chapter we employ another model to deal with the outliers and the service overload problem. On considering QoS of a service, we believe it is unacceptable if some responses take a very long time. In this model we set a threshold value T for the response time. We try several values of the proper threshold in our experiment. If some $d_{ij} > T$, we drop this link by considering it disconnected. Moreover, to simplify the problem, we assume the user accepts the service as long as the response time is shorter than the threshold. Overall, our target is to determine k instances that satisfy as many users as possible. Our mechanism can be extended to direct the user to a light load and close enough but not necessary the closest service instance.

To model this situation, consider following problem:

Given Bipartite graph $B(V_1, V_2, E)$ where

$$|V_1| = M$$
$$|V_2| = N$$
$$i \in V_1, j \in V_2$$

$$\begin{cases} (i,j) \in E, & d_{ij} \leq T; \\ (i,j) \notin E, & \text{otherwise.} \end{cases}$$

Maximize:

$$|N_B(V')|$$

Subject to:

$$V' \subset V_1$$
$$|V'| = k$$

$|N_B(V')|$ is the number of nodes in the neighbor set of V', meaning the target is to choose a subset of V_1 to cover as many vertices in V_2 as possible. Actually it is a *max k-cover* problem.

In this problem we construct M sets by setting i to be the ith vertex in V_1 and the elements in this set are vertices connected to i in V_2. We are trying to find k-sets to cover as many elements as possible.

Max k-cover problem is a classical problem and is well studied. This problem is NP hard hence we do not expect to achieve the exact answer. Again we use approximate algorithms. Greedy algorithm (Algorithm 5.2) is proven to be one of the best polynomial time algorithms for this problem [76]. It could give a $(1 - 1/e)$ approximation, which means it could cover at least $(1 - 1/e)$ of the maximum elements k-sets could cover.

In the greedy algorithm, every round we find a data center s which, when combined with current selection set S, could cover the maximum users. There may be more than one such data centers. In Algorithm 5.2 we choose one of them randomly. It can also be done more precisely as we can use a stack to record and try all choices one

Algorithm 5.2 Greedy algorithm for max k-cover problem

```
1: S ← φ.
2: while Have not covered all users yet & Used less than k instances do
3:    maxcover ← 0
4:    clear list l
5:    for all data centers s do
6:       if use s could cover c > maxcover more users than use instances in S then
7:          clear list l
8:          add s to l
9:          maxcover ← c
10:      else if use s could cover exactly maxcover more users than use instances in S
         then
11:         add s to l
12:      end if
13:   end for
14:   random select s from l
15:   S ← S + s
16:   (random eliminate limit users in cover list of s if s covers more than limit users)
17: end while
18: return S
```

by one. However, we find that with the growing of k, the amount of branches increases so fast that the reward quickly diminishes. So we do not include this implementation in this chapter.

Moreover this model as well as the algorithm can be easily modified to direct the user to a light load and close enough but not necessary the closest service instance. This can be done by restricting the number of users to whom an instance can be connected. To modify the formulation, we can add a constraint $|N_B(u)| <= limit, \forall u \in V'$ after $|V'| = k$, where $limit$ is the number of users an instance can connect to. As for the Algorithm 5.2, in line 15, instead of $S \leftarrow S + s$ we consider the integer $limit$. If s covers $\geq limit$ more users, we randomly pick up $limit$ number of users to cover. Moreover, if s covers $\geq limit$ users we would force to set it as covering $= limit$ users (Line 16). By these changes, we can limit the connections and select a data center to deploy multiple instances if needed. This constraint can make the model more realistic. We call this algorithm modified algorithm.

Local search method would find local extreme solution thus can be utilized to help improving the greedy algorithm. Again we use single swap for the original greedy algorithm. While in the modified algorithm, swapping is not that easy, we launch the algorithm several times to improve its performance.

5.5 Experiment and Discussion

In this section we conduct experiments to show the necessity of redeployment and to compare different algorithms. The dataset as well as the simulation program are open source released for reproducible experiments on our website [22].

5.5.1 Dataset Description

To obtain real-world response-time values of different service instances, we implement a WSCrawler and a WSEvaluator using Java. Employing our WSCrawler, addresses of 4,302 openly-accessible services are obtained from the Internet. We deploy our WSEvaluator to 303 distributed computers of PlanetLab, which is a distributed test-bed made up of computers all over the world. In our experiment, each PlanetLab computer invokes all the Internet services for one time and the corresponding response-time value is recorded. By this real-world evaluation, we obtain a 303×4302 matrix containing response-time values.

5.5.2 Necessity of Redeployment

We first establish the necessity of redeploying service instances. We fix the number of used service instances to 3 and scale the user from 100 to 500. We repeat the experiment for 100 times and find in the worst case the performance is really bad. This is illustrated in Figure 5.2. Suppose we deploy the optimal service instances for

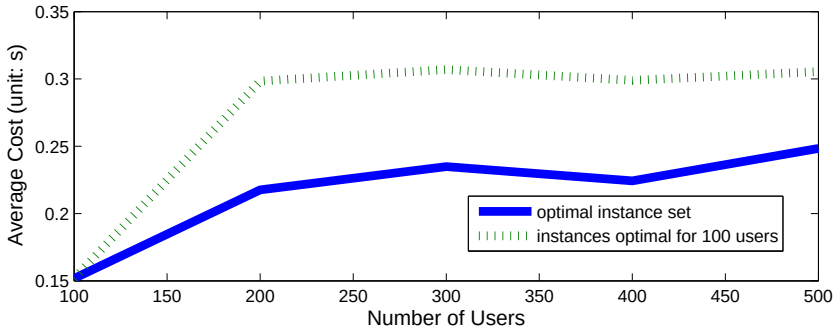


Figure 5.2: Worst case without redeployment

the first 100 users. With the number of users growing, if we do not redeploy the instances the average cost may decrease below 70% of the optimal average cost. In worse cases, far away users would make the average cost much larger. Without redeployment the network performance may discourage the new users. To avoid the worst case from happening, redeployment or at least a performance checking is necessary. Whereas, redeployment is a costly operation which we should not do too frequently.

5.5.3 Weakness of Auto Scaling

As discussed before, current cloud service would apply auto scaling in responding to the change of user scale. We should answer whether it is good enough to simply apply this service without redeployment. We indicate there are two main drawbacks of current auto scaling approaches which we show as followings.

Current auto scaling is limited within one region. We simulate this by selecting out only a part of data centers as potential data centers to deploy instances. Applying the same algorithm to deploy instances, we can see that the average user cost is higher to choose instances from partial data centers than that of choosing from all data centers. Figure 5.3 shows the result. We pick out 50, 100, 150 data centers from totally 303 data centers and deploy 10 instances in these data centers. We apply the *greedy init + single swap* algorithm

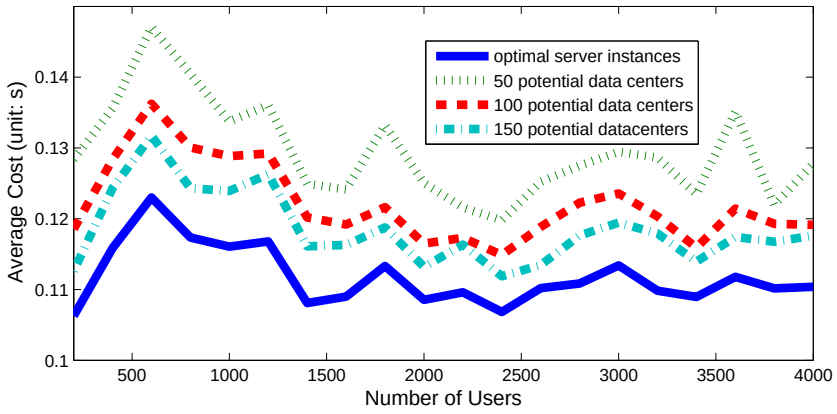


Figure 5.3: Deploy in limited data centers

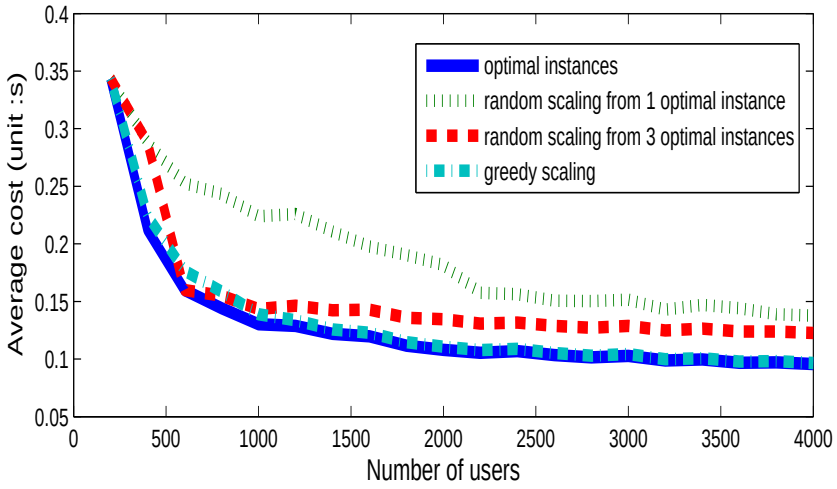
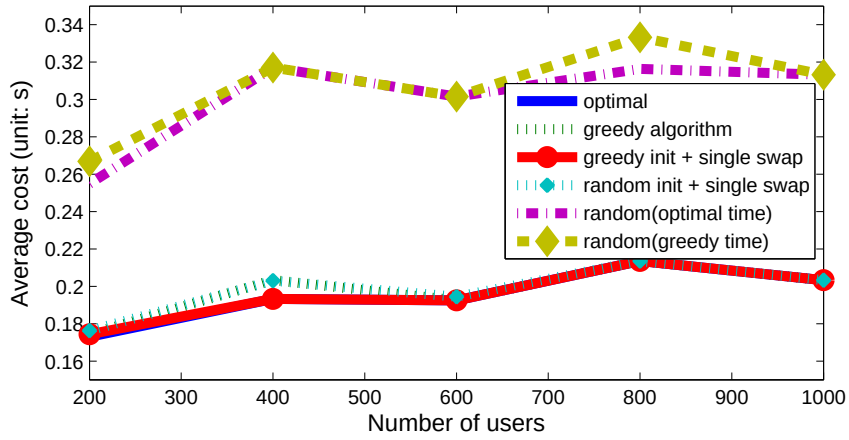
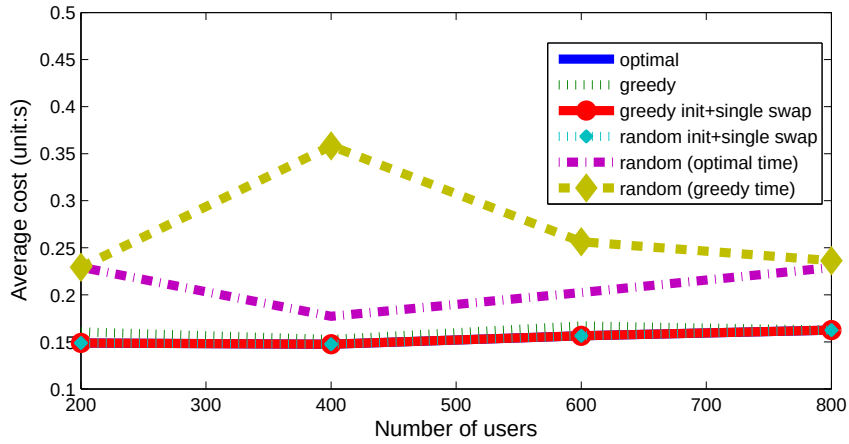


Figure 5.4: Auto scaling algorithms

and the result of using total 303 potential data centers is employed as the baseline. The performance decreases greatly if we choose only 50 potential data centers. Moreover, we find the performance is not affected much by the number of users. Note the distance between users and data centers is better preserved if we choose 100 (33%) or more from the total data centers, and thus the performance gets greatly improved in these cases. It is due to the fact that there are several data centers which are close to many users. The average distance would decrease greatly if we could pick up these data centers.



(a) Selecting 2 data centers by redeployment algorithm



(b) Selecting 3 data centers by redeployment algorithm

Figure 5.5: Selecting data centers by redeployment algorithm

Another disadvantage of current auto scaling approach is that it launches new instances in the data center containing least instances. It is like randomly picking up a new data center to grow. We simulate randomized scaling and the result is in Figure 5.4. On a small scale (*e.g.*, select no more than 4 instances), we consider all possible combination and find the optimal choice. On a larger case, we use again the *greedy init + single swap* algorithm as our best choice. We start from one optimal instance and 200 users. Then we randomly add 200 users at each step and select one more instance.[2] We see

[2]We notice there is a decreasing trend in the figure. As we randomly pick up users, we use

the performance is bad in this situation. If we defer the random selecting after we find 3 optimal service instances for 600 users, we would obtain a better choice than doing so in the beginning.

As we see in this case, if we could do the scaling a little more intelligently by using a simple greedy algorithm, we can get a very good result. However, there still exist some problems by using greedy algorithm, such as some instances would get extremely heavy load. We would discuss this issue later.

5.5.4 Comparing the Redeployment Algorithms for k-Median

The previous experiment confirms the need for redeployment. As we discussed in Section 5.4.1, we formulate the redeployment as the *k-median* problem and compare the algorithms we proposed. First we set k to be 2,3 and M to be 303. We compare the algorithm output to the optimal output generated by the brute force algorithm.

We utilize the optimal average cost as the baseline and compare the greedy algorithm output with the baseline. For initialization of local search with single swap (of selected data center) algorithm, we use the output obtained by the greedy algorithm or a random vector. From Figure 5.5a and 5.5b we know in such a small scale, the greedy algorithm is good enough. The local search algorithm would be even better. Moreover, these algorithms are better in terms of time complexity as they perform better than the random selection algorithm.

When selecting more than 10 instances the brute force algorithm would run in an unacceptably long time, and we use the *greedy init + single swap* algorithm as the baseline to observe the relative performance of the greedy and the *random init + single swap* algorithms. Once more we compare them with the random selection

expected value to illustrate this trend. Suppose the expected average cost for one instance to serve 200 users is c. Then expected average cost for 2 instances serve 200 users each independently should also be c. Since we get the optimal 2 instances serve 400 users the value should be less than c. Similar analyses hold for the rest of the curve.

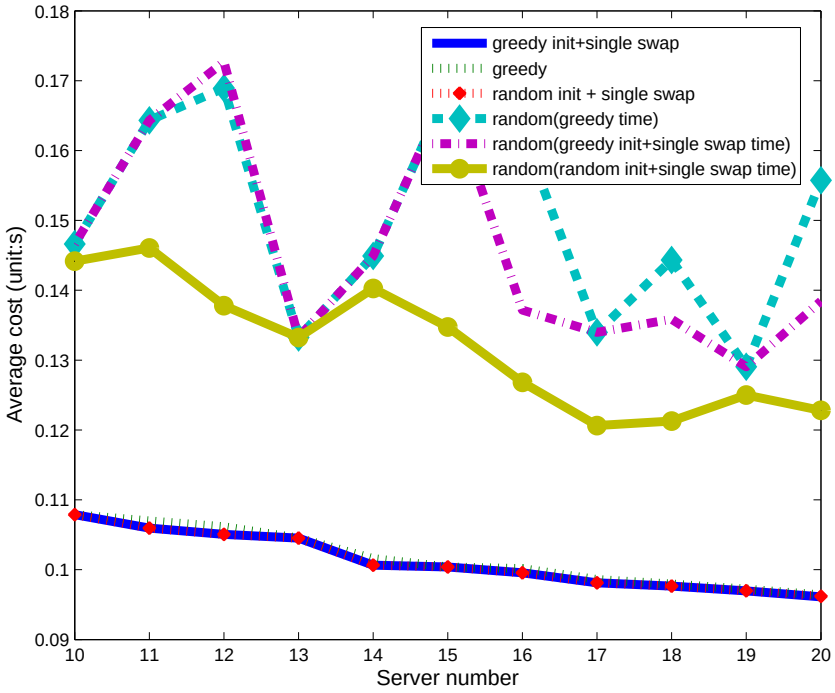


Figure 5.6: Selecting 10 - 20 data centers for 4000 users

Table 5.2: Execution time of the algorithms (unit: ms)

	Brute Force	**Greedy**	**Greedy init + single swap**	**Random init + single swap**
(2,200)	203	< 1	< 1	16
(2,400)	375	< 1	16	31
(2,600)	547	< 1	15	47
(2,800)	735	< 1	31	31
(3,600)	78969	< 1	31	63
(10,4000)	-	94	328	2641
(15,4000)	-	172	500	13109
(20,4000)	-	203	1906	25469

algorithm. We select 10 to 20 data centers to deploy instances so as to satisfy 4000 users. The result is shown in Figure 5.6.

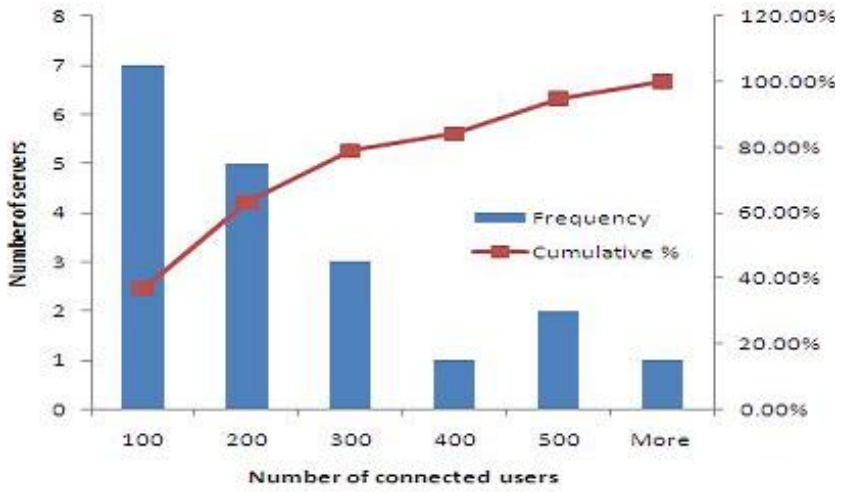


Figure 5.7: Histogram on number of connected users for each server

The *random init + single swap* algorithm is slightly better than the greedy algorithm, but it takes longer to compute. Again, the random selection algorithm fails to give a satisfying answer, which means both the greedy and the local search algorithms are well suited for this problem.

Table 5.2 lists some typical computing times for running these algorithms. The pair (s, u) in the table (*e.g*, $(2, 400)$) stands for selecting s service instances to satisfy u users. Values in the table are the executing time and the unit is millisecond. We are interested in their comparative relation. We can see that the brute force algorithm does cost with an exponential time growth. The greedy as well as the *greedy init + single swap* algorithms run quite fast comparing to the *random init + single swap* algorithm because normally we expect more swaps to find a local optimizer by a random initial vector.

5.5.5 Redeployment Algorithms for Max k-Cover

Through experiments we confirm the *greedy + single swap* algorithm is good enough in terms of both result and time complexity to solve *k-median* problem. This makes the redeployment simple; however, there is a limitation of *k-median* model. Recall on formulating

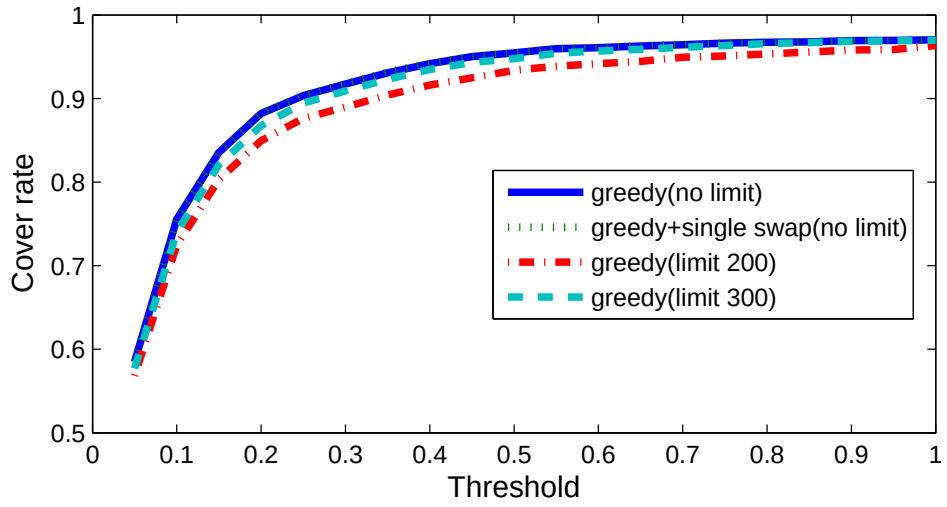


Figure 5.8: Max k-cover using greedy approach

redeployment in the *k-median* problem we have assumed users are evenly distributed and all users can request service from their nearest service instance. In practice this condition does not always hold.

To motivate the construction of the *max k-cover* model, Figure 5.7 shows a typical distribution of user connections on the service instances. 20 instances are selected to provide service for 4000 users. We expect all instances to connect to about 200 users each, while the real case is that about 20% instances connect to up to 400 or more users. Therefore these instances get very heavy load and the performance may become unsatisfiable.

Virtually the cloud has infinite resource, however, in reality the budget is restricted. Service providers can only launch limited (say, k) instances in the cloud. If we would like to ease the burden of some instances we are required to put more than one instances in one data center and to distribute the service instances in k' (where $k' < k$) data centers.

To attack this, we use another model to formulate the problem which is *max k-cover* problem. As discussed in Section 5.4.2, the threshold to cut edges is a key parameter we should pick up carefully. We employ the entire 303×4302 matrix in our experiment

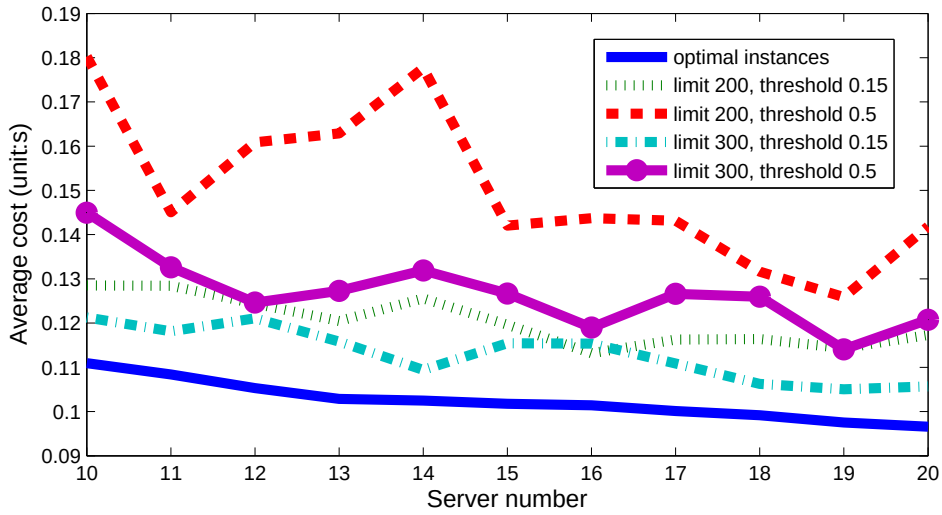


Figure 5.9: Average cost by max k-cover model

and test the value of the threshold ranging from 0.05 to 1. We select data centers to deploy 20 service instances. Two typical values of limitation are applied. The first one is 215, with $20 * 215 = 4300$, which means we can cover at most 4300 users. The second one is 300 which is a loose limitation. The result is illustrated in Figure 5.8. The local search (with single swap) has no much improvement over the greedy algorithm under this setting. A tight limitation (*e.g.*, 200) will affect the performance of coverage; on the other hand loosen the limitation a little bit, the performance will be unaffected by this limitation (close to the result of no limitation).

Since we cut the edges with a weight less than the threshold, the average cost of the covered users is less than that of the threshold. So the threshold to some extent is related to the average cost of the users. To this end, we compared the average cost of users by the instances we selected using the greedy algorithm for the *max k-cover* problem to the cost by instances we selected using the algorithms for the *k-median* problem. We use the result generated by the *greedy init + single swap* algorithm of the *k-median* model as a baseline. The result is illustrated in Figure 5.9. During the experiment we changed k, the number of instances to choose, and accordingly we

set the total number of users to $200 * k$. We pick up two typical values of limitation and threshold. On setting the limitation we use a tight limitation of 200 and a loose one of 300 as done previously. As for the threshold, from previous experiment we get the average cost of above 0.1 using *greedy init + single swap* algorithm. So we use 0.15 as the threshold to ensure the cost of the covered users is below 0.15. We note using the threshold of 0.5 can cover most users in *max k-cover* model. So we employ these two values in the experiment. The result shows a lower threshold would be better than a higher one. Although this limitation decreases the performance, the resulting model is more realistic.

5.6 Discussion

The main contribution of this work is to model the cloud service deployment problem instead of proposing new algorithm. Therefore, in this work we use quite simple but shown to be effective algorithms to solve the proposed model. We choose greedy algorithm and local search algorithm because greedy algorithm is proven to be one of the best polynomial time algorithms for the set cover problem [76], and local search provides the current best known bound for approximating *k-median* problem [40]. Some other heuristic algorithms to solve optimization problems can also apply but that is not our contribution.

In this work, we collect a dense distance matrix in a controlled environment. However, in real cloud environment, the distance matrix is sparse with many missing values since users may not visit all data centers in advance. Therefore we use the missing value prediction technique proposed by Ma et al. [131] to recover the dense matrix as collected in this work. Other missing value prediction techniques can also be applied as long as it could recover the distance matrix with full connection.

5.7 Summary

In this chapter, we highlight the problem of hosting the cloud services. Our work consists two parts. First we propose a framework to address the new features of cloud. In the cloud we can get all possible connection information on potential locations that the service instances could be deployed. The second work is that we formulate the redeployment of service instances as *k-median* and *max k-cover* problems. By doing so, they can be solved by fast approximate algorithms.

We employ a large data set collected via real-world measurement to evaluate the algorithms. The experimental results show our algorithms work well on this problem. The data set and the source codes for the experiment are public released.

□ **End of chapter.**

Chapter 6

Deployment of Multiple Cloud Services

Chapter Outline

In cloud computing, multiple services tend to cooperate with each other to accomplish complicated tasks. Deploying these services should take the interactions among different services into consideration. We model and solve the multi-services co-deployment problem in this chapter. The key points of this chapter are as followings:

- Introduces the framework of cloud-based multi-services and the data processing procedure.
- Model cloud-based multi-services co-deployment problem.
- Evaluate the proposed model on real-world dataset; Public release the dataset and simulation code for repeatable experiments.

6.1 Background and Motivation

We have witnessed the rapid growing of cloud-based services. The tendency of cloud-based services is to deliver computation, software, and data access as services located in data centers distributed over the world [39, 155, 182]. Typically, cloud provides three layers of services namely infrastructure as a service, platform as a service and software as a service (*i.e.*, IaaS, PaaS and SaaS). The cloud-based services are generally deployed on virtual machines (VMs) in the cloud data centers (*e.g.*, Amazon EC2) [186]. Since there are usually a large number of virtual machines in a cloud, making optimal deployment of cloud-based services to suitable virtual machines is an important research problem.

To provide good service performance for users, auto scaling and elastic load balance are widely studied [4, 13]. Currently, different cloud services are typically deployed independently by their own providers. Let's take Facebook and Google as examples. Facebook provides social network service in its own cloud; Google also provides email services and document services in its own cloud. These companies have their own users. They make optimal deployment of their services independently. Our previous work [110], which is shown in Chapter 5, proposes a redeployment mechanism to make optimal deployment for such kind of independent cloud services, considering the distribution of users and workload of the servers.

However, in reality, different cloud-based services may cooperate with each other to complete complicated tasks. For example, when watching a video on YouTube, you can click the share button to share it to Facebook or Twitter by invoking the services provided by Facebook or Twitter; when editing a file on Google Doc you can call Gmail service to shard the document with your colleagues by email; when purchasing commodities on Taobao[1], you can call Alipay[2]

[1] http://www.taobao.com
[2] http://www.alipay.com

service to pay the money (both Taobao and Alipay are affiliated entities of Alibaba Group). The number of various kinds of cloud-based services increasing, whereas the giant cloud providers in the market are not so many, thus multiple services deployed by the same provider cooperate together to fulfill user requests becomes even more common.

Compared with making optimal deployment for a single service, making optimal deployment for multiple correlated services is a more challenging research problem. The situation becomes very complicated when there are tons of services and their deployments affect each other. We could not simply treat the request from other services as the same as those from users since the service VMs of other services are under deployment at the same time and could be migrated to other places. Therefore it is critical to make a global decision for deploying multiple services together, especially for companies (*e.g.*, Google and Alibaba) which host many services or for companies that would like to cooperate with each other. We call this the co-deployment problem of cloud-based services.

This chapter points out the new research problem of multi-services co-deployment in cloud computing and provides comprehensive investigations. For ease of discussion, we assume the cloud-based services are deployed on virtual machines. Replacing VMs with physical servers does not affect our model. We model the multi-services co-deployment problem formally as an integer programming problem which minimizes the latency of user requests. To evaluate the effectiveness of our proposed multi-services co-deployment model, large-scale real-world experiments are conducted, involving 307 distributed computers in about 40 countries, and 1881 real-world Internet-based services in about 60 countries. The comprehensive experimental results show that our proposed latency-aware co-deployment mechanism can provide much better performance than traditional independent deployment techniques. To make our experiments reproducible, we publicly release [21] our

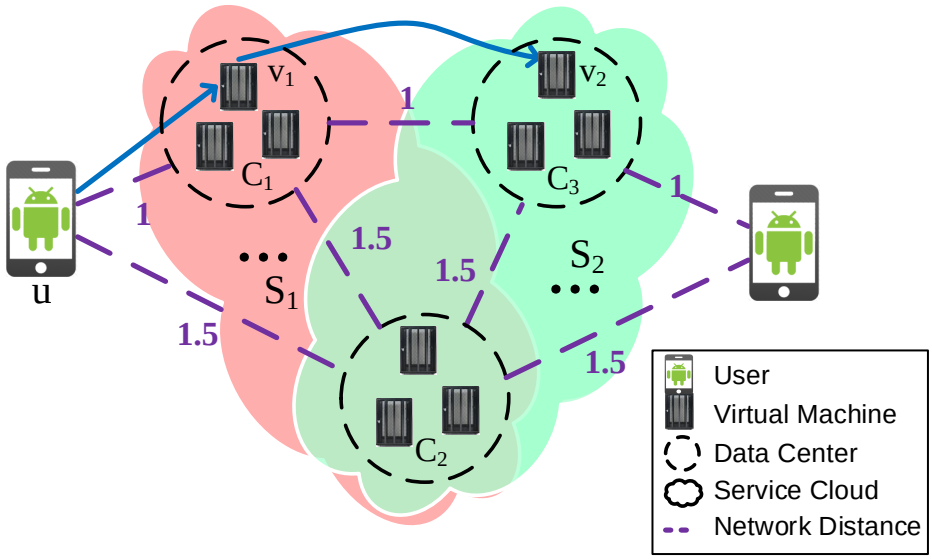


Figure 6.1: The framework of cloud-based multi-services
(The two user icons in the figure actually represent the same user u.)

reusable research dataset, which includes about 577,000 accesses of Internet-based services and 94,000 pings of computers over the world as well as our source codes for experiment.

6.2 Framework of Cloud-based Multi-services

Figure 6.1 shows the general framework of cloud-based multi-services. As shown in the figure, multiple services (*e.g.*, S_1 and S_2) are to be deployed in different clouds. Suppose S_1 (*e.g.*, Google Doc service) and S_2 (*e.g.*, Gmail service) are correlated services (*i.e.*, one request of S_1 would involve S_2 and vice versa). There are three data centers. The network distances between these data centers are shown in the figure (the unit of network distance is second). Among these data centers, C_2 (*e.g.*, Amazon EC2) allows public access while C_1 and C_3 are private clouds (*e.g.*, provided by Google Doc and Gmail, respectively). In this example, S_1 can be deployed in C_1 and C_2 while S_2 can be deployed in C_2 and C_3.

Network distances between users and data centers are marked in

Model 6.1 Single Cloud-based Service Deployment Model

$$\text{minimize} \sum_{\substack{i \in U \\ j \in C}} r_i d_{ij} x_{ij}$$

subject to:

$$\sum_{j \in C} x_{ij} = 1, \qquad \forall i \in U, \tag{6.1}$$

$$x_{ij} \leq y_j, \qquad \forall i \in U, \forall j \in C, \tag{6.2}$$

$$\sum_{j \in C} y_j \leq k, \tag{6.3}$$

$$x_{ij} \in \{0, 1\}, \qquad \forall i \in U, j \in C,$$

$$y_j \in \{0, 1\}, \qquad \forall j \in C.$$

the figure. Firstly, we apply the traditional independent deployment technologies for each service, where the requests from end users and these from other service VMs are not distinguished. It is not hard to see that C_1 and C_3 would be chosen separately for S_1 and S_2. This deployment makes the average latency of requests from both users and other services to be $1s$. If we deploy S_1 (or S_2) in C_2, the average network distance of S_1 (or S_2) would increase to $1.5s$. The deployment decision seems optimal for both S_1 and S_2 when making independent deployment. However, if the providers of S_1 and S_2 cooperate with each other to seek a better deployment solution (e.g., development teams of Google Doc and Gmail try to making co-deployment of these two services together, since there are a lot of invocations between these services and these two teams belong to the same company), the objective becomes to reduce the overall request latency for the users of both services instead of independent services.

For example, a request of S_2 (S_1 similarly) is actually a calling sequence user $\rightarrow S_2 \rightarrow S_1$. To finish this sequence, the latency $2s$ consists two parts, namely $1s$ from user to service VM of S_1 in C_1

and $1s$ from service VM of S_1 in C_1 to service VM of S_2 in C_3. If the deployment problem would be considered globally, deploying both services in C_2 would be a better choice. Therefore the overall request latency of the calling sequence would be $1.5s$ rather than $2s$.

6.3 Independent Deployment of Single Service

First we consider a simple case of deploying a single cloud-based service. Suppose a service is to be deployed on k VMs with that we have already obtained the distances between users and service VMs. We apply the long existing k-median model [130] in the integer programming formulation as Model 6.1.

In Model 6.1, U and C are the sets of users and potential service VMs. r_i is the count of requests from user i. d_{ij} is the network distance between user i and j-th candidate service VM. x_{ij} and y_j are decision variables. y_j indicates whether service VM j is chosen or not to deploy the service. x_{ij} indicates whether user i would request j-th service VM. This model aims at selecting a set of service VMs to deploy service ($y_j = 1$ if selected). User i could connect to the closest available service VM ($x_{ij} = 1$ if connected). The objective function is to minimize the total distance of all requests. Constraint 6.1 ensures one user would connect to one service VM to fulfill the requests. Constraint 6.2 ensures requests of the users would only be processed on the selected service VM. Constraint 6.3 guarantees at most k service VMs are chosen.

The model is in integer programming formulation. We can apply mature techniques to solve the problem very efficiently with an acceptable approximation of the optimal solution. Actually more constraints could be added (*e.g.*, model the computing power decreasing of servers with number of connections). Whereas, to simplify the discussion in this section as a starting point, we use the basic model.

Table 6.1: Alphabet of the multiple cloud-based services co-deployment model

Notation	Descriptions
U	Set of users
m	Number of services
C_h	Candidate VMs set of service h
k_h	Number of VMs service h could deploy
r_{hi}	Number of requests for service h from user i
d_{hij}	Distance between user i and j-th VM of service h
r_{iqs}	Number of times that service q is involved in the requests for service s from user i
d_{pqrs}	Distance between p-th VM of service q and r-th VM of service s
x_{hij}	(Indicator variable) whether user i would request j-th VM for the service h
y_{ipqrs}	(Indicator variable) for user i whether r-th VM of service s is chosen to respond the requests from p-th VM of service q
z_{hj}	(Indicator variable) whether j-th VM of service h is chosen

6.4 Co-deployment of Multi-services

After introducing the single cloud-based service deployment, we turn to the more complicated problem of deploying multiple services. We extend the model of deploying a single service. An integer programming model is proposed and heuristic approach is provided to solve the problem. The notations in the model are given in Table 6.1.

Model 6.2 Cloud-based Multi-services Co-deployment Model

$$\text{minimize} \sum_{\substack{i\in U\\ 1\le h\le m\\ j\in C_h}} r_{hi} d_{hij} x_{hij} + \sum_{\substack{i\in U\\ 1\le q,s\le m, q\neq s\\ p\in C_q, r\in C_s}} r_{iqs} d_{pqrs} y_{ipqrs}$$

subject to:

$$\sum_{j\in C_h} x_{hij} = sign(r_{hi}), \qquad 1\le h\le m, \forall i\in U, \tag{6.4}$$

$$x_{hij} \le z_{hj}, \qquad 1\le h\le m, \forall j\in C_h, \forall i\in U, \tag{6.5}$$

$$\sum_{j\in C_k} z_{hj} \le k_h, \qquad 1\le h\le m, \tag{6.6}$$

$$\sum_{\substack{1\le s\le m\\ s\neq h\\ r\in C_s}} y_{ijhrs} \le x_{hij}, \qquad 1\le h\le m, \forall j\in C_h, \forall i\in U, \tag{6.7}$$

$$y_{ipqrs} \le z_{rs}, \qquad 1\le q,s\le m, q\neq s, \forall p\in C_q, \forall r\in C_s, \forall i\in U, \tag{6.8}$$

$$\sum_{\substack{p\in C_q\\ r\in C_s}} y_{ipqrs} = sign(r_{iqs}), \qquad 1\le q,s\le m, q\neq s, \forall i\in U, \tag{6.9}$$

$$x_{hij}\in\{0,1\}, \qquad 1\le h\le m, j\le C_h, \forall i\in U,$$

$$y_{ipqrs}\in\{0,1\}, \qquad 1\le q,s\le m, q\neq s, \forall p\in C_q, \forall r\in C_s,$$

$$z_{hj}\in\{0,1\}, \qquad 1\le h\le m, \forall j\in C_h.$$

6.4.1 Multiple Cloud-based Services Co-deployment Model

As we have introduced the motivation of deploying multiple services simultaneously, a model is designed. Model 6.2 optimizes the

Algorithm 6.1 Iterative sequential co-deployment algorithm

```
1: tempS ← φ, S ← φ
2: temp ← MAX, tempmin ← MAX, min ← MAX
3: for all service h do
4:    Select a set S_h of k_h service VMs randomly among the candidate set C_h
5:    tempS ← tempS + S_h
6: end for
7: for i = 1 → n do
8:    tempmin ← Evaluate the solution tempS
9:    repeat
10:      temp ← tempmin
11:      for all service h do
12:         Select a set S'_h of k_h service VMs according to Model 6.1 with decided
            tempS
13:         tempS ← tempS − S_h + S'_h
14:      end for
15:      tempmin ← Evaluate the solution tempS
16:      if tempmin < min then
17:         min ← tempmin
18:         S ← tempS
19:      end if
20:   until |tempmin − temp| ≤ ε
21:   Disturb the solution set tempS
22: end for
23: return S
```

latencies of both the requests from users and the cross-services requests.

Model 6.2 decides the co-deployment of service VMs in a candidate set for all services ($z_{hj} = 1$ if the VM is selected for Service h). User i could connect to the closest available service VM for service h ($x_{hij} = 1$ if connected). The objective function aims at minimizing the total distance of all requests (including requests from all users and other cooperative services).

The Constraints 6.4 to 6.6 are similar to those in single cloud-based service deployment model. The function $sign(x)$ in Equation 6.4 returns 1 for $x > 0$ and 0 for $x = 0$. Equation 6.4 constrains user i would connect to one service VM to fulfill the requests if the user has a request of service h. Inequality 6.5 ensures the

requests of the users would only be processed on the selected service VM. Constraint 6.6 ensures at most k_h service VMs are chosen for service h. The Constraints 6.7 to 6.9 are introduced for cross-service requests. Inequality 6.7 means for user i the cross-service requests from service h to s should be initiated from the j-th VM of service h, in which j-th service VM is chosen to serve user i for requests of service h. Inequality 6.8 constrains the cross-service requests could only be sent to selected service VMs. Constraint 6.9 ensures one link would be set if and only if there are cross-service requests.

6.4.2 Iterative Sequential Co-deployment Algorithm

We have acquired a co-deployment model in Section 6.4.1, whereas, there are already too many decision variables. It is infeasible to apply general methods to obtain an approximate solution of the model. Therefore we take the special properties of this problem and get a heuristic approach. Algorithm 6.1 shows the approach.

Algorithm 6.1 first generates a random deployment (Line 3 to 6) as there is no information about where to deploy. After deciding a temporary deployment, the algorithm tries to sequentially improve the deployment of the services one by one (Line 11 to 14). It treats the requests from other services the same as those from the users in Line 12. By iteratively doing the improvement procedure in Line 9 to 20, the deployment of all services would converge. The best deployment of the iteration does not necessary appear at last, so we record the best ever deployment in Line 16 to 19. Since the iterative sub-procedure would fall into a local minimum, we disturb the solution (usually change one or two chosen service VMs) and run n loops to find a good enough co-deployment.

It is worth mentioning that this iterative computing algorithm is different from doing unsupervised single service deployment. The goal of this algorithm is to acquire a feasible suboptimal solution for Model 6.2. It is true that if we do not participate in the deploying

schedule of all the services, after some time these services would also evolve to an overall balanced suboptimal deployment. We call this a natural evolution approach. There are several disadvantage of natural evolution approach compared to our algorithm. We list three of them.

1. For the natural evolution approach, we have to wait for quite a long time until the deployment performing well globally since the redeployment of services would not be done frequently. On contrary, our algorithm can compute the result in a short time.

2. The natural evolution approach would incur many real migrations of service VMs which costs a lot. The migration of one service could cause the migrations of some other services and a chain reaction continues, but the order is random and hard to predict. Our algorithm exploits the changes in a more systematic way and does not trigger any real migrations.

3. Last but the most important reason that our algorithm outperforms the natural evolution approach is we are not necessary to fall into local optimum. We disturb the result in Line 21 and repeat our iterative algorithm to jump out of the local minimum, while the natural evolution approach has no choice but to be stuck in the local optimum.

6.5 Experiment and Discussion

We first collect a large real-world latency data set. Based on the data set we conduct the experiment to evaluate methods of deploying multiple cloud-based services. We compare our algorithm to the independent deployment method. The independent deployment method deploys the service VMs randomly at the beginning. After the logs have been collected, all services redeploy the VMs applying single service deployment model without differentiating requests

from users and from other services. Ilog Cplex 9.0 [17] is applied to give a good enough approximate solution of the integer programming problems (Model 6.1). Both the dataset and the related source codes are public released on our website [21].

6.5.1 Latency Data Collection

We use a similar assumption as in our previous work [110] described in Chapter 5. Generally, latency of a service request (the request either from user or another service) contains three parts: the Internet delay between the request sponsor and the gateway of cloud data center, the Intranet delay inside the cloud, and the computing time on the service VM. While inside a data center, usually gigabyte switches are used, the Intranet delay can be ignored. Moreover, assuming two service VMs in the same data center have the same computational capability. Thus their processing time of a service can be viewed as the same. Therefore, the dominant part of the latency is the Internet delay. Under this assumption, there is no difference between two VMs in the same data center. We use *distance* between a user u and a data center C to denote the latency between u and *any* VM v in C. Similarly, *distance* between a pair of data centers C_i and C_j represents the latency between *any* pair of VMs v_i and v_j deployed in C_i and C_j separately.

Since the service VMs are running in the cloud, the service providers can collect the latency data when fulfilling user requests. The calling sequences of services are recorded as well. One log entry contains the user ID, the service ID it requests, a sequence of service IDs involved and a sequence of latencies of every service requests.

For any user u, if u has ever requested a service VM v in data center C, the recorded latency between (u, v) is used to represent the distance between u and C. When more than one VM in C is requested by u, we take the average latency. Distance between two

pairs of data centers is retrieved by the same way. If there exists a latency record from a service VM v_i in data center C_i to v_j in C_j, this value is taken as the distance between C_i and C_j. If there is no historical information about a specific distance, we could use similar user/data center to predict the missing distance value (applying the method described in the work [131]).

We use again the example in Figure 6.1 to illustrate this. The user u requests the VM v_1 followed by the VM v_1 requests the VM v_2. The calling sequence is recorded together with the latency between the pair (u, v_1) and (v_1, v_2). These two values are regarded as distances between pair (u, C_1) and (C_1, C_2).

After this period of data processing, we obtain two distance matrices. One matrix records the distances between every $(user, data\ center)$ pair. Another matrix records the distances between every pair of data centers.

6.5.2 Dataset Description

To obtain real-world response time values of diverse users and service VMs, we implement a Crawler using Java. Employing our Crawler, we get the addresses of 1,881 openly-accessible services. We deploy shell script codes to 307 distributed computers of PlanetLab, which is a distributed test-bed with hundreds of computers all over the world. During the experiment, each PlanetLab computer pings all the Internet services as well as all the other PlanetLab computers once and records the round trip time (RTT). The ping operations are done simultaneously in a randomized way to avoid overflow on one computer. By this real-world evaluation, we obtain two matrices and public release them [21]. One is a 307×1881 matrix containing RTT values between PlanetLab computers and web services (P2W Matrix). The other one is a 307×307 matrix with RTT values between every pair of PlanetLab computers inside (P2P Matrix). These two matrices are used as the two matrices described

Table 6.2: Dataset statistics (unit: ms)

Statistics	P2W Matrix	P2P Matrix
minimum	0.01	0.08
$25th$ percentile	53.69	58.58
median	118.14	133.76
$75th$ percentile	176.00	188.53
maximum	1604.02	5704.04
average	129.41	138.38

Table 6.3: Parameters used in randomized log generation

Notation	Descriptions	Default
n_h	Number of log entries of service h	1880
ρ_h	Ratio of users use service h to total users	0.2
l	Number of services involved in one service request	5

in the previous section. All the service VMs are represented by the containing data center in the experiment.

6.5.3 Experiment Parameters

The experiment is conducted on the data set we obtained (whose statistics is shown in Table 6.2) with randomly generated user logs. The iterative sequential algorithm for solving Model 6.2 is evaluated. If it is not specially explained the parameters of the model in Table 6.1 are set to default values. By the default setting, there are 1881 users, 10 services. Every service would deploy 10 service VMs among a candidate set in 100 data centers. A user of service s would have 5 request logs. One request of a service would involve on average 5 requests of other services. The parameters without default value r_{iqs} together with parameter r_{hi} depend on user logs.

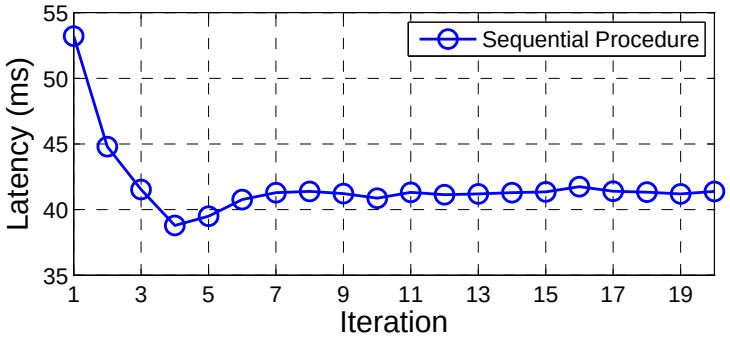


Figure 6.2: Convergence of sequential procedure

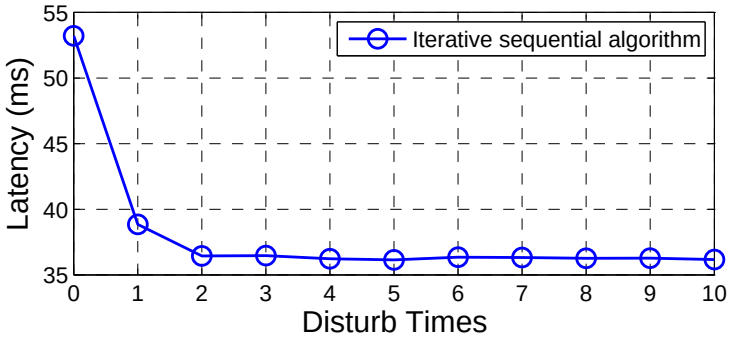


Figure 6.3: Number of disturbs

We generate user logs randomly. Table 6.3 states several parameters related to the randomized log generation.

6.5.4 Algorithm Specifics

Convergence of Iterative Sequential Procedure

There is a sequential procedure in Algorithm 6.1 (Line 11 to 14) that does the service deployment one by one. As shown in Figure 6.2, the procedure would soon converge after 5 iterations. So it is quite safe to limit the iteration number and make the algorithm run faster.

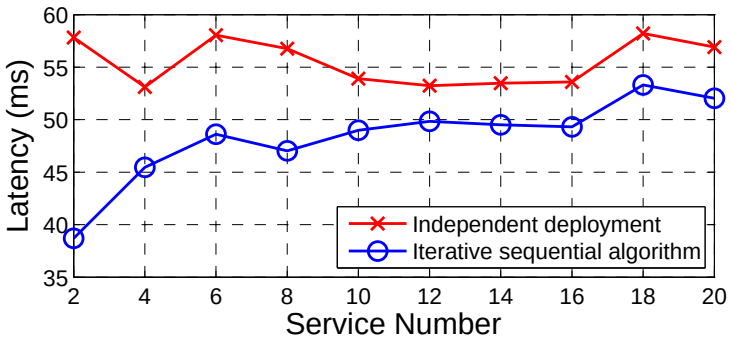


Figure 6.4: Number of services

Number of Disturbs

To avoid falling into a bad local minimum, we disturb the result and run several loops to get a better result in Algorithm 6.1. Figure 6.3 shows the effect of adding disturbs. The latency value of disturb 0 is the average latency of random deployment. We could see as the figure shows the biggest improvement is made on the first iteration. The disturbing would let the algorithm jump out of the local minimum. The result of the first several iterations is good enough that less further improvements are made.

6.5.5 Number of Services

We change the total number of services and conduct the experiment. The result is shown in Figure 6.4. We can see our algorithm has the improvement of 10% to 20%. Since we set the number of services involved in one request to be the default value, the average latency of one request does not vary a lot. The variations of two curves are caused by the random generation of the candidate set of service VMs of different services in different service number setting.

An observation is that the improvement of our algorithm decreases for big service numbers. It is because with the growing number of services there are more service VMs selected. Therefore more information is known when deploying one service. Every

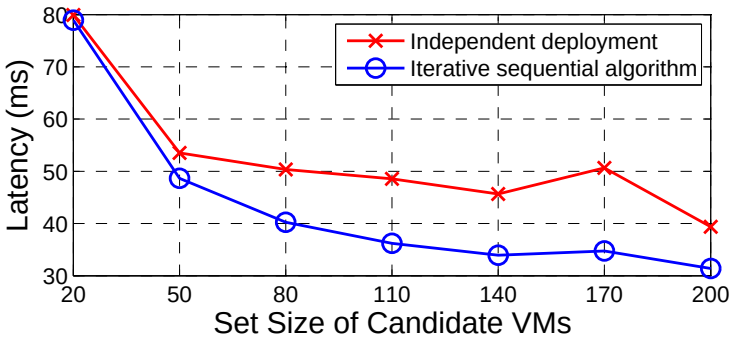


Figure 6.5: Set size of candidate VMs

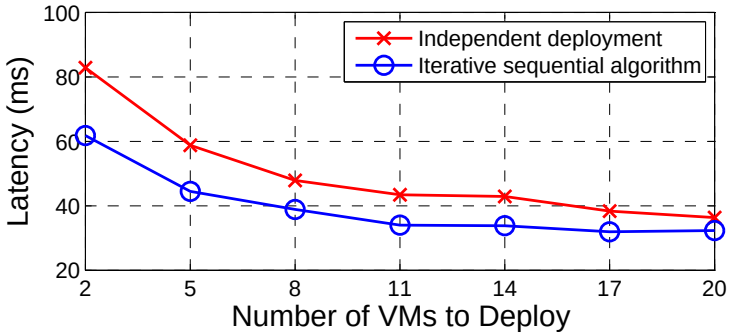


Figure 6.6: Number of service VMs to deploy

service could choose overall better service VMs for many users and other service VMs independently. Thus, the improvement of our algorithm decreases. While it is more common situation that there are not many services, our algorithm is effective.

6.5.6 Number of Service VMs

Size of Candidate Set of Service VMs

We set different size of candidate service VM set. The result can be found in Figure 6.5. The decreasing tendency of the curve is not hard to understand. Since there are more potential service VMs to be chosen from the result should be better. The abnormal increasing on set size 170 is caused by randomized initial deployment phase.

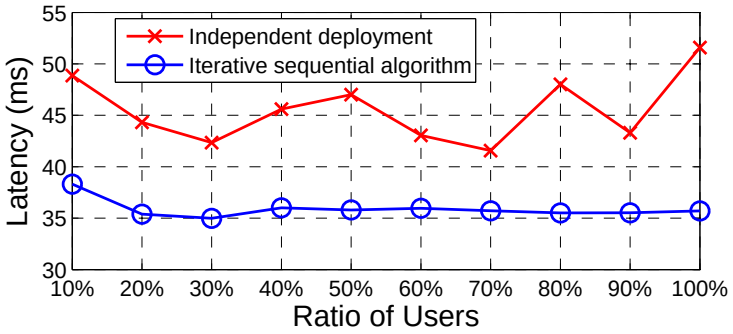


Figure 6.7: Number of service users

It is worth mentioning that our algorithm has greater improvement with bigger set size. The reason is with more candidates to be chosen from, our algorithm could do a much better decision than considering the deployment independently.

Number of Service VMs to Deploy

Figure 6.6 is generated by modifying the number of total service VMs of every service. We can see our algorithm outperforms the independent deployment method by at most 25 percentages. Especially deploying a smaller number of VMs, we can make a much better decision. If we can deploy many service VMs, independently decision could find several outstanding VMs that perform well. Therefore in these cases our algorithm does not have many advantages.

6.5.7 Services Logs

We tried different log generation parameters and see the behavior of our algorithm under different use cases of the services.

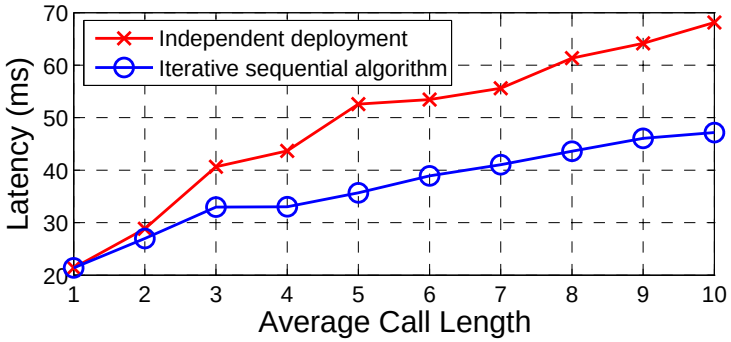


Figure 6.8: Average call length

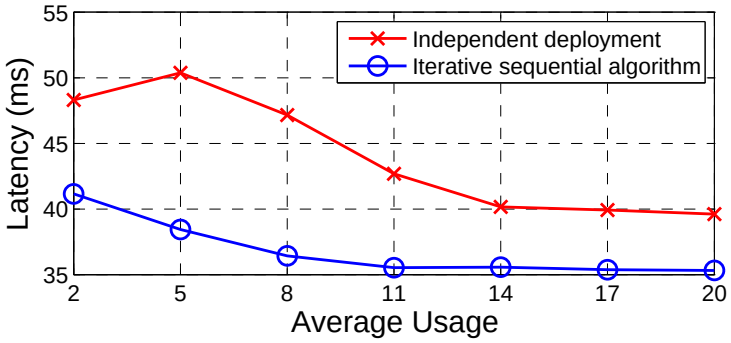


Figure 6.9: Average usage

Number of Service Users

In Figure 6.7 we change the number of service users. Since we have totally 1881 sample users we change the ratio of users to use one service. For example 10% means we have logs of 188 users that use a specific service. The result shows that due the randomization property the performance of independent deployment varies a lot, while our algorithm is quite stable under all size of users. Moreover, our algorithm reduces the average latency up to 30%.

Average Call Length of Service

We define the call length as the number of service requests involved in one user task. For example, a call length of 5 means a user

request of a specific service s would involve 4 consequent requests of other services from the VM of service s. As shown in Figure 6.8, our algorithm outperforms the independent deployment method especially when the service call length is large. It can also be found that the increasing curve of our algorithm is more stable.

Number of Logs

We modify the number of total logs on randomized generation. The result is shown in Figure 6.9. Average usage of a user means the average number of a user requesting a specific service. We can expect the more records of users the more intelligent we can deploy the service. As we can see in the figure, the average latency decreases with more user logs. An observation is that the gap between our algorithm and independent deployment method becomes narrower with the growing number of the user logs. This could be explained by that there are numerous user logs providing enough information even for deploying service independently. We can see the tendency of two curves. Our algorithm converges when there are more than 10 average usages, while independent deployment method converges after 14. Therefore our algorithm could do a better deployment when the number of logs is insufficient.

6.6 Summary

In this chapter, we investigate the latency-aware cloud-based multiple services co-deployment problem. Motivating examples are given to illustrate the problem. We offer the method to show what and how the user logs could be collected. The co-deployment problem is abstracted in a general framework. Integer programming formulation is employed to model the problem and a new heuristic algorithm is given to obtain an approximate solution. Extensive experiments are conducted on hundreds of computers over the world. The dataset

and the source codes of the experiment are public released. The experimental results show the effectiveness of our model.

□ **End of chapter.**

Chapter 7

Conclusion and Future Work

Chapter Outline

In this chapter, we give a brief conclusion of this thesis and summarize our contributions. We also provide several interesting future directions.

7.1 Conclusion

Performance is a critical criterion for mobile applications and cloud services. Users are very intolerant to poor performance in mobile apps (*e.g.*, laggy, slow responses). However, due to a lack of handy tools and clear guidelines, developers struggle to tune the performance. In this thesis, we discuss performance-tuning mechanisms for both mobile applications and cloud services. Open-source tools and optimization guides are offered. Real-world performance issues are found by our tools.

More specifically, in Chapters 3 and 4 we discuss tuning the performance of mobile apps. In Chapters 5 and 6, we investigate mechanisms for tuning the performance of cloud services. We first study mechanisms for diagnosing the performance of mobile apps in Chapter 3. In Chapter 4, we discuss the UI design that could

make the users more tolerant of long latency. Finally, cloud service deployment, which is a critical aspect of performance, is studied. We study single cloud service deployment in Chapter 5 and multiple services co-deployment in Chapter 6.

In Chapter 3, we introduce the design and implementation of `DiagDroid`, a handy, open-source tool that allows developers to diagnose the performance of mobile apps. It is the first work that can be easily applied to Android mobile devices with different OS versions. We first categorize the asynchronous tasks of Android apps. Then we propose the Android framework instrumentation method for profiling the asynchronous tasks. Then, the suspicious asynchronous tasks are identified according to some pre-defined criteria. The suspicious tasks together with their related piece of source codes are then reported to the developers which help them a lot on performance diagnosis. We have successfully applied our tool to real-world Android apps and found 27 previously unknown performance issues. The whole diagnosis procedure can be done by ourselves, who are not familiar with the apps being tested. This result is convincing that our tool can be useful to developers.

In Chapter 4, we discuss the problem of responsive UI design in Android apps. The need to detect poor-responsive operations is confirmed by our user survey. The survey results show that users' patience is correlated with UI responsiveness. We design and implement a tool called `Pretect` that can detect poor-responsive operations. The tool is shown to work correctly on synthetic benchmarks and on open source apps. We further verify `Pretect` with real-world case studies. The results demonstrate the effectiveness of `Pretect`.

In Chapter 5, we propose a model for optimizing cloud service redeployment. The users and cloud VM instances communication costs are a critical component of operation delays in mobile app. We first model and extract the major part of such communication delays. Then we propose methods for measuring and predicting such delays.

We propose two optimization problems for reducing the total delay. Several algorithms are put forth to solve the problems. To verify our idea and the usefulness of the algorithms, we collect a dataset of communication delays in the real-world and conduct simulation experiments. The results show the effectiveness of our proposed methods. Our collected data and simulation source codes have been public released. Other researchers could repeat our results or use our data for further research.

In Chapter 6, we extend the cloud service deployment problem discussed in the previous chapter to the multiple cooperative cloud services deployment case. In this work, we collect a dataset of not only the communication delays between users and VM instances but also communication delays between VM instances. Then, we propose an optimization model for service deployment that considers both user-service and service-service communication delays. We also collect real-world data and conduct simulations to show the effectiveness of our methods. Again, the data and the source codes for the simulation have been public released.

In summary, in this thesis we enhance the user experienced performance of mobile apps and cloud services by reducing latency. We propose comprehensive performance tuning mechanisms for both mobile apps and cloud services. We implement the proposed mechanisms. The data and our tools have been public released online to make our results repeatable, and more importantly, to make our mechanism available to developers.

7.2 Future work

Delays in cloud-based mobile apps can come from three sources: mobile-side delays, cloud-side delays and communication delays. The performance can be tuned in all three parts. Although we have made a number of significant achievements in this thesis, there are still many interesting research problems for future studies.

7.2.1 User Experience Enhancement of Mobile Applicaions

Users interact directly with their mobile apps. Therefore, optimizing the performance of mobile apps is critical to improving user experience. We propose two promising directions for research on user experience enhancement, separately, preloading/precomputing techniques, and delay-tolerant UI design.

Selective Loading, Computing in Advance

Preloading large size resources, and precomputing long-term computations are accepted as effective methods for reducing latency. The idea is to use the mobile devices' hardware resources during idle circles to enhance performance. Unlike PCs, mobile devices have limited energy; therefore, unnecessary preloading and precomputing should not be conducted to avoid wasting energy. Predicting and judging more precisely whether the pre-processing of the resources/computation would be beneficial is the key to saving energy. Our work, which detects the long-term operations during testing, provides some help for this problem. Identifying the frequent use patterns of these long-term operations may help users and developers to select appropriate resources/computations for preloading/precomputing.

Delay Tolerant UI Design

Our study has shown that without feedback, users tend to become impatient with long-term operations. The users may doubt the sensitivity of the touch screen or believe that the UI has frozen. In our work, we focus on offering yes/no feedback to user input. Further research could examine what kind of feedback is better for users. For example, Airbnb [23] shows fancy pictures when loading information. However, no study has shown that this design outperforms a traditional loading bar design and developers may doubt whether it is really a better design. A survey similar to the one

in this thesis could be conducted to verify whether new UI designs outperform established ones in terms of user satisfaction.

7.2.2 Cloud Service Processing Time Optimization

With the powerful computation resource, processing time of cloud services can be reduced a lot. This work focuses on optimizing the deployment of computing instance in clouds. We try to make the instance as close to users as possible. Another direction for optimizing instance deployment is to move the instances to where the data are stored. The instances that store the data in the cloud could be different from the instances that perform computing. This would create delays inside the cloud for retrieving data. The topology of the cloud could be optimized by reducing the distance between the data servers and computation servers.

Another direction is considering application-specific optimization. This work focuses on general applications, but different applications have distinct characteristics. For example, social networking applications may require a lot of inter-server communications inside a cloud for obtaining massive amounts of information (*e.g.*, presenting updates from large groups of friends), whereas, servers for different news applications store similar contents. The cloud server characteristics of different applications could be taken into consideration for further optimization.

7.2.3 Cloud-client Communication Cost Reduction

In addition to tuning the performance on both mobile apps and cloud services, the communication between the two could be optimized to reduce the delays. Our proposed mechanism is one method for reducing the cost per communication, further research could is to reduce the number of communications.

The communication complexity between the two has already been studied in fields such as information theory, but the theoretical

bound has not been achieved in practice. Applying the theorem successfully to real-world cloud-mobile communication is an interesting and practical research problem. What kind of communications can be merged to reduce the number of communications and how to do this are interesting problems. The extent to which the number of communications can approach the bound also deserves further study.

□ **End of chapter.**

Appendix A

List of Publications

1. **Yu Kang**, Yangfan Zhou, Hui Xu and Michael R. Lyu. DiagDroid: Android Performance Diagnosis via Anatomizing Asynchronous Executions. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2016.

2. Hui Xu, Yangfan Zhou, Cuiyun Gao, **Yu Kang**, and Michael R. Lyu. SpyAware: Investigating the Privacy Leakage Signatures in App Execution Traces. In *Proc. of the International Sysposium on Software Reliability Engineering (ISSRE)*, pages 348-358, 2015.

3. Jiaojiao Fu, Yangfan Zhou, and **Yu Kang**. Dependability Issues of Android Games: A First Look via Software Analysis. In *Proc. of the International Workshop on Internet-based Virtual Computing Environment (iVCE)*, pages. 291-296, 2015.

4. **Yu Kang**, Zibin Zheng, and M.R. Lyu. A Latency-aware Co-deployment Mechanism for Cloud-based Services, In *Prof. of the IEEE International Conference on Cloud Computing (Cloud)*, pages 630-637, 2012.

5. Jieming Zhu, **Yu Kang**, Zibin Zheng, and Michael R. Lyu. WSP: A Network Coordinate based Web Service Positioning Framework for Response Time Prediction. In *Proc. of the*

IEEE International Conference on Web Services (ICWS), pages 90-97, 2012.

6. Jieming Zhu, **Yu Kang**, Zibin Zheng and Michael R. Lyu. A Clustering-based QoS Prediction Approach for Web Services Recommendation. In *Proc. of the International Workshop on Internet-based Virtual Computing Environment (iVCE)*, pages 93-98, 2012.

7. **Yu Kang**, Yangfan Zhou, Zibin Zheng, and M.R. Lyu. A User Experience-based Cloud Service Redeployment Mechanism. In *Proc. of the IEEE International Conference on Cloud Computing (Cloud)*, pages 227-234, 2011.

□ **End of chapter.**

Bibliography

[1] 12306 mobile app download. https://kyfw.12306.cn/otn/appDownload/init.

[2] 2 billion consumers worldwide to get smart(phones) by 2016. http://www.emarketer.com/Article/2-Billion-Consumers-Worldwide-Smartphones-by-2016/1011694.

[3] The aicache website. http://aicache.com/.

[4] Amazon auto scaling developer guide. http://aws.amazon.com/documentation/autoscaling/.

[5] Amazon elastic compute cloud (amazon ec2). http://aws.amazon.com/ec2/.

[6] Apache hadoop. http://hadoop.apache.org/.

[7] Application Performance Solutions | Akamai. http://uk.akamai.com/aps.

[8] Average daily time spent online via mobile by internet users worldwide in 2015, by age group (in hours). http://www.statista.com/statistics/428425/daily-time-spent-online-mobile-age/.

[9] AWS | High Performance Computing - HPC Cloud Computing. http://aws.amazon.com/ec2/hpc-applications/.

[10] Best practices for performance — android developers. http://developer.android.com/training/best-performance.html.

[11] Chinese government launches app for booking train tickets, and it already sucks. https://www.techinasia.com/chinese-government-launches-app-booking-train-tickets-sucks/.

[12] Cloud computing technology market and mobile cloud industry strategic focus research reports. http://www.prnewswire.com/news-releases/cloud-computing-technology-market-and-mobile-cloud-industry-strategic-focus-research-reports-286988271.html.

[13] Elastic load balancing developer guide. http://aws.amazon.com/documentation/elastic-load-balancing/.

[14] Google app engine. http://code.google.com/appengine/.

[15] Google cloud endpoints. https://cloud.google.com/appengine/docs/java/endpoints/.

[16] Growth of time spent on mobile devices slows. http://www.emarketer.com/Article/Growth-of-Time-Spent-on-Mobile-Devices-Slows/1013072.

[17] Ibm cplex optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

[18] Microsoft's windows azure platform. http://www.microsoft.com/windowsazure/.

[19] Number of apps available in leading app stores as of july 2015. http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[20] Parallel Programming and Computing Platform | CUDA | NVIDIA | NVIDIA. http://www.nvidia.com/object/cuda_home_new.html.

[21] Resources of multiple services co-deployment. http://appsrv.cse.cuhk.edu.hk/~ykang/cloud/multi_service_deploy.zip.

[22] Resources of single service deployment. http://appsrv.cse.cuhk.edu.hk/~ykang/cloud/cloud_redeploy.zip.

[23] Vacation rentals, homes, apartments & rooms for rent - Airbnb. https://www.airbnb.com/.

[24] abhi2rai/RestC. https://github.com/abhi2rai/RestC.

[25] S. Abolfazli, Z. Sanaei, E. Ahmed, A. Gani, and R. Buyya. Cloud-based augmentation for mobile devices: Motivation, taxonomies, and open challenges. *IEEE Communications Surveys Tutorials (CST '14)*, 16(1):337–368, 2014.

[26] Activity Testing — Android Developers. http://developer.android.com/tools/testing/activity_testing.html.

[27] S. Alimadadi, A. Mesbah, and K. Pattabiraman. Understanding asynchronous interactions in full-stack JavaScript. In *Proc. of the ACM/IEEE International Conference on Software Engineering (ICSE '16)*, pages 1169–1180, 2016.

[28] N. Alon, B. Awerbuch, and Y. Azar. The online set cover problem. In *Proc. of the Annual ACM Symposium on Theory of Computing (STOC '03)*, pages 100–105, 2003.

[29] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*, pages 739–753, 2010.

[30] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE '12)*, pages 258–261, 2012.

[31] C. Amrutkar, M. Hiltunen, T. Jim, K. Joshi, O. Spatscheck, P. Traynor, and S. Venkataraman. Why is my smartphone slow? on the fly diagnosis of underperformance on the mobile internet. In *Proc. of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, pages 1–8, 2013.

[32] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proc. of the Joint Meeting on Foundations of Software Engineering (FSE '12)*, pages 1–11, 2012.

[33] G. Anderson, R. Doherty, and S. Ganapathy. User perception of touch screen latency. *Design, User Experience, and Usability: Theory, Methods, Tools and Practice (DUXU '11)*, 6769:195–202, 2011.

[34] M. Andreolini, S. Casolari, M. Colajanni, and M. Messori. Dynamic load management of virtual machines in cloud architectures. In *Lecture Notes of the Institute for Computer*

Sciences, Social Informatics and Telecommunications Engineering (LNICST '10), volume 34, pages 201–214. 2010.

[35] Android drawer. `http://www.androiddrawer.com/`.

[36] Android Execute Multiple AsyncTask Parallely. `http://stackoverflow.com/questions/30011054/android-execute-multiple-asynctask-parallely`.

[37] Pretect: Poor-responsive ui detection of android applications. `http://www.cudroid.com/pretect`.

[38] Apktool: A tool for reverse engineering Android apk files. `http://ibotpeaches.github.io/Apktool/`.

[39] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *ACM Communication (CACM '10)*, 53(4):50–58, 2010.

[40] V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Munagala, and V. Pandit. Local search heuristic for k-median and facility location problems. In *Proc. of the Annual ACM Symposium on Theory of Computing (STOC '01)*, pages 21–29, 2001.

[41] AsyncTask. `http://developer.android.com/reference/android/os/AsyncTask.html`.

[42] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proc. of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*, pages 641–660, 2013.

[43] A. Banerjee. Static analysis driven performance and energy testing. In *Proc. of the ACM SIGSOFT International Sym-*

posium on Foundations of Software Engineering (FSE '14), pages 791–794, 2014.

[44] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury. Static analysis driven cache performance testing. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS '13)*, pages 319–329, 2013.

[45] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for opencl based heterogeneous computing on clusters with many gpu devices. In *Proc. of the IEEE International Conference on Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS '10)*, pages 1 –7, 2010.

[46] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. The impact of api change- and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering (TSE '15)*, 41(4):384–407, 2015.

[47] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proc. of the annual conference on Internet measurement (IMC '10)*, pages 267–280, 2010.

[48] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. In *Proc. of the ACM workshop on Research on enterprise networking (WREN '09)*, pages 65–72, 2009.

[49] I. Brandic. Towards self-manageable cloud services. In *Proc. of the Annual IEEE International Computer Software and Applications Conference (COMPSAC '09)*, pages 128–133, 2009.

[50] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *Proc. of the ACM SIGSOFT*

Symposium on The Foundations of Software Engineering (FSE '09), pages 3–12, 2009.

[51] R. Buyya, C. S. Yeo, and S. Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *Proc. of the IEEE International Conference on High Performance Computing and Communications (HPCC '08)*, pages 5–13, 2008.

[52] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems (FGCS '09)*, 25(6):599–616, June 2009.

[53] Y. Cai and L. Cao. Effective and precise dynamic detection of hidden races for java programs. In *Proc. of the Joint Meeting on Foundations of Software Engineering (FSE '15)*, pages 450–461, 2015.

[54] B. Cao and G. Uebe. An algorithm for solving capacitated multicommodity p-median transportation problems. *The Journal of the Operational Research Society (JORS '93)*, 44(3):259–269, 1993.

[55] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *Proc. of the Annual Symposium on Foundations of Computer Science (FOCS '99)*, pages 378–388, 1999.

[56] M. Charikar, S. Khuller, D. M. Mount, and G. Narasimhan. Algorithms for facility location problems with outliers. In *Proc. of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '01)*, pages 642–651, 2001.

[57] Q. A. Chen, H. Luo, S. Rosen, Z. M. Mao, K. Iyer, J. Hui, K. Sontineni, and K. Lau. Qoe doctor: Diagnosing mobile

app qoe with automated ui control and cross-layer analysis. In *Proc. of the Conference on Internet Measurement Conference (IMC '14)*, pages 151–164, 2014.

[58] J. P. Chin, V. A. Diehl, and K. L. Norman. Development of an instrument measuring user satisfaction of the human-computer interface. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '88)*, pages 213–218, 1988.

[59] W. Choi, G. Necula, and K. Sen. Guided GUI testing of android apps with minimal restart and approximate learning. In *Proc. of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*, pages 623–640, 2013.

[60] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? In *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*, pages 429–440, 2015.

[61] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proc. of the Conference on Computer Systems (EuroSys '11)*, pages 301–314, 2011.

[62] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research (MOOR '79)*, 4(3):233–235, 1979.

[63] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of the 2nd conference on Symposium on Networked Systems Design & Implementation (NSDI'05)*, pages 273–286, 2005.

[64] Configuring ART. `https://android.googlesource.com/platform/frameworks/base`.

[65] M. Creeger. Cloud computing: An overview. *ACM Queue*, 7(5):627–636, June 2009.

[66] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: Making smartphones last longer with code offload. In *Proc. of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*, pages 49–62, 2010.

[67] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of The ACM (CACM '08)*, 51(1):107–113, 2008.

[68] J. Deber, R. Jota, C. Forlines, and D. Wigdor. How much faster is fast enough? user perception of latency & latency improvements in direct and indirect touch. In *Proc. of the ACM Conference on Human Factors in Computing Systems (CHI '15)*, pages 1827–1836, 2015.

[69] DiagDroid - Android Performance Diagnosis via Anatomizing Asynchronous Executions. `http://www.cudroid.com/DiagDroid`.

[70] H. T. Dinh, C. Lee, D. Niyato, and P. Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing (WCMC '13)*, 13(18):1587–1611, 2013.

[71] D. Duis and J. Johnson. Improving user-interface responsiveness despite performance limitations. In *Proc. of the IEEE Computer Society International Conference (Compcon Spring '90)*, pages 380–386, 1990.

[72] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, 2001.

[73] D. Ersoz, M. Yousif, and C. Das. Characterizing network traffic in a cluster-based, multi-tier data center. In *Proc. of the IEEE International Conference on Distributed Computing Systems (ICDCS '07)*, pages 59 –59, 2007.

[74] F-Droid. `https://f-droid.org/`.

[75] R. Farivar, A. Verma, E. Chan, and R. Campbell. Mithra: Multiple data independent tasks on a heterogeneous resource architecture. In *Proc. of the IEEE International Conference on Cluster Computing and Workshops (CLUSTER '09)*, pages 1–10, 2009.

[76] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM '98)*, 45:634–652, 1998.

[77] N. Fernando, S. W. Loke, and W. Rahayu. Mobile cloud computing: A survey. *Future Generation Computer System (FGCS '13)*, 29(1):84–106, 2013.

[78] L. Findlater and J. McGrenere. Impact of screen size on performance, awareness, and user satisfaction with adaptive graphical user interfaces. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*, pages 1247–1256, 2008.

[79] C. Forlines, D. Wigdor, C. Shen, and R. Balakrishnan. Direct-touch vs. mouse input for tabletop displays. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*, pages 647–656, 2007.

[80] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. November 1998.

[81] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Proc. of the Grid Computing Environments Workshop (GCE '08)*, pages 1–10, 2008.

[82] Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna. Clapp: Characterizing loops in android applications. In *Proc. of the Joint Meeting on Foundations of Software Engineering (FSE '15)*, pages 687–697, 2015.

[83] D. F. Galletta, R. Henry, S. McCoy, and P. Polak. Web site delays: How tolerant are users? *Journal of the Association for Information Systems (JAIS '04)*, 5(1):1–28, 2004.

[84] G. Gan, C. Ma, and J. Wu. *Data Clustering: Theory, Algorithms, and Applications (ASA-SIAM Series on Statistics and Applied Probability)*. 2007.

[85] C. Gao, B. Wang, P. He, J. Zhu, Y. Zhou, and M. R. Lyu. Paid: Prioritizing app issues for developers by tracking user reviews over versions. In *Proc. of the International Symposium on Software Reliability Engineering (ISSRE '15)*, pages 35–45, 2015.

[86] Z. Gao, C. Fang, and A. M. Memon. Pushing the limits on automation in gui regression testing. In *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE '15)*, 2015.

[87] S. L. Garfinkel. An evaluation of amazon's grid computing services: Ec2, s3 and sqs. Technical report, Center for Research on Computation and Society, Harvard University, 2007.

[88] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *Proc. of the International Conference on Software Engineering (ICSE '13)*, pages 72–81, 2013.

[89] C. J. Goodwin. *Research in psychology : methods and design*. 1995.

[90] Google. `http://developer.android.com/reference/android/os/StrictMode.html`.

[91] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao. Accelerating mobile applications through flip-flop replication. In *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*, pages 137–150, 2015.

[92] B. D. Harper, L. A. Slaughter, and K. L. Norman. Questionnaire administration via the www: A validation & reliability study for a user satisfaction questionnaire. In *WebNet*, volume 97, pages 1–4, 1997.

[93] B. Hayes. Cloud computing. *Communications of the ACM (CACM '08)*, 51(7):9–11, July 2008.

[94] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *Proc. of the International Conference on Software Engineering (ICSE '15)*, pages 483–493, 2015.

[95] Hierarchy viewer. `http://developer.android.com/tools/help/hierarchy-viewer.html`.

[96] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proc. of the ACM SIGPLAN/SIGOPS*

international conference on Virtual execution environments (VEE '09), pages 51–60, 2009.

[97] J. A. Hoxmeier and C. Dicesare. System response time and user satisfaction: An experimental study of browser-based applications. In *Proc. of the Association of Information Systems Americas Conference (AMCIS '00)*, pages 10–13, 2000.

[98] J. Huang and L. Rauchwerger. Finding schedule-sensitive branches. In *Proc. of the Joint Meeting on Foundations of Software Engineering (FSE '15)*, pages 439–449, 2015.

[99] W. Huang, Q. Gao, J. Liu, and D. K. Panda. High performance virtual machine migration with rdma over modern interconnects. In *Proc. of the IEEE International Conference on Cluster Computing (CLUSTER '07)*, pages 11–20, 2007.

[100] K. Jain, M. Mahdian, and A. Saberi. A new greedy approach for facility location problems. In *Proc. Annual ACM Symposium on Theory of Computing (STOC '02)*, pages 731–740, 2002.

[101] K. Jain and V. V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. *Journal of the ACM (JACM '01)*, 48:274–296, 2001.

[102] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA '13)*, pages 67–77, 2013.

[103] M. Ji, E. W. Felten, and K. Li. Performance measurements for multithreaded programs. *ACM SIGMETRICS Performance Evaluation Review (PER '98)*, 26(1):161–170, June 1998.

[104] J. Johnson. *Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules*. 2010.

[105] M. E. Joorabchi, M. Ali, and A. Mesbah. Detecting inconsistencies in multi-platform mobile apps. In *Proc. of the International Symposium on Software Reliability Engineering (ISSRE '15)*, pages 450–460, 2015.

[106] R. Jota, A. Ng, P. Dietz, and D. Wigdor. How fast is fast enough? a study of the effects of latency in direct-touch pointing tasks. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*, pages 2291–2300, 2013.

[107] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: Performance bug detection in the wild. In *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPLSA '11)*, pages 155–170, 2011.

[108] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proc. of the ACM SIGSOFT Symposium on The Foundations of Software Engineering (FSE '09)*, pages 13–22, 2009.

[109] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *Proc. of the ACM SIGCOMM conference on Internet measurement conference (IMC '09)*, pages 202–208, 2009.

[110] Y. Kang, Y. Zhou, Z. Zheng, and M. Lyu. A user experience-based cloud service redeployment mechanism. In *Proc. of the International Conference on Cloud Computing (CLOUD'11)*, pages 227 –234, july 2011.

[111] G. Kecskemeti, P. Kacsuk, G. Terstyanszky, T. Kiss, and T. Delaitre. Automatic service deployment using virtualisation. In *Proc. of the Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP '08)*, 2008.

[112] Keeping your app responsive. `http://developer.android.com/training/articles/perf-anr.html`.

[113] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM '12)*, pages 945–953, 2012.

[114] D. Kovachev, T. Yu, and R. Klamma. Adaptive computation offloading from mobile devices into the cloud. In *Proc. of the IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA '12)*, pages 784–791, 2012.

[115] S. Kristoffersen and F. Ljungberg. "making place" to make it work: Empirical explorations of hci for mobile cscw. In *Proc. of the International ACM SIGGROUP Conference on Supporting Group Work (GROUP '99)*, pages 276–285, 1999.

[116] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*, pages 151–165, 2015.

[117] S. Lee and S. Zhai. The performance of touch screen soft buttons. In *Proc. of the SIGCHI Conference on Human*

Factors in Computing Systems (CHI '09), pages 309–318, 2009.

[118] Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering (TSE '06)*, 32(6):382–403, 2006.

[119] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm. What's inside the cloud? an architectural map of the cloud landscape. In *Proc. of the ICSE Workshop on Software Engineering Challenges of Cloud Computing (ICSE Cloud '09)*, pages 23–31, 2009.

[120] Libinject – c/c++ code injection library. `https://code.google.com/p/libinject/`.

[121] G. Lin, G. Dasmalchi, and J. Zhu. Cloud computing and it as a service: Opportunities and challenges. In *IEEE International Conference on Web Services (ICWS '08)*, pages 6–7, 2008.

[122] Y. Lin, S. Okur, and D. Dig. Study and refactoring of android asynchronous programming. In *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*, pages 224–235, 2015.

[123] Y. Lin, C. Radoi, and D. Dig. Retrofitting concurrency for android applications through refactoring. In *Proc. of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '14)*, pages 341–352, 2014.

[124] Y.-D. Lin, J. Rojas, E.-H. Chu, and Y.-C. Lai. On the accuracy, efficiency, and reusability of automated test oracles for android devices. *IEEE Transactions on Software Engineering (TSE '14)*, 40(10):957–970, 2014.

[125] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proc.*

of the International Conference on Software Engineering (ICSE '14), pages 1013–1024, 2014.

[126] Y. Liu, C. Xu, and S.-C. Cheung. Diagnosing energy efficiency and performance for mobile internetware applications. *IEEE Software*, 32(1):67–75, 2015.

[127] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13(4):383–390, 1975.

[128] Low RAM Configuration. `https://source.android.com/devices/tech/config/low-ram.html`.

[129] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 329–339, 2008.

[130] D. M. Network and discrete location: Models, algorithms and applications. *Journal of the Operational Research Society (JORS '93)*, 48(7):763–763, 1997.

[131] H. Ma, I. King, and M. R. Lyu. Effective missing data prediction for collaborative filtering. In *Proc. of the Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR '07)*, pages 39–46, 2007.

[132] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proc. of the Joint Meeting on Foundations of Software Engineering (FSE '13)*, pages 224–234, 2013.

[133] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proc. of the*

ACM International Symposium on Foundations of Software Engineering (FSE '14), pages 599–609, 2014.

[134] M. Melo, S. Nickel, and F. S. da Gama. Facility location and supply chain management - a review. *European Journal of Operational Research (EJOR '09)*, 196(2):401 – 412, 2009.

[135] R. B. Miller. Response time in man-computer conversational transactions. In *Proc. of the December 9-11, 1968, Fall Joint Computer Conference, Part I (AFIPS '68)*, pages 267–277, 1968.

[136] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. Sigdroid: Automated system input generation for android applications. In *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE '15)*, pages 461–471, 2015.

[137] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes (SEN '12)*, 37(6):1–5, Nov. 2012.

[138] monkeyrunner. `http://developer.android.com/tools/help/monkeyrunner_concepts.html`.

[139] Y. Moon, D. Kim, Y. Go, Y. Kim, Y. Yi, S. Chong, and K. Park. Practicalizing delay-tolerant mobile apps with cedos. In *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*, pages 419–433, 2015.

[140] R. M. L. M. Moreira, A. C. R. Paiva, and A. Memon. A pattern-based approach for gui modeling and testing. In *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE '13)*, pages 288–297, 2013.

[141] F. F.-H. Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, 2004.

[142] A. Ng, J. Lepinski, D. Wigdor, S. Sanders, and P. Dietz. Designing for low-latency direct-touch input. In *Proc. of the ACM Symposium on User Interface Software and Technology (UIST '12)*, pages 453–464, 2012.

[143] D. T. Nguyen, G. Zhou, G. Xing, X. Qi, Z. Hao, G. Peng, and Q. Yang. Reducing smartphone application delay through read/write isolation. In *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*, pages 287–300, 2015.

[144] J. Nielsen. Response times: The 3 important limits. In *Usability Engineering*, 1993.

[145] S. Niida, S. Uemura, and H. Nakamura. Mobile services. *IEEE Vehicular Technology Magazine*, 5(3):61–67, 2010.

[146] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proc. of the International Conference on Software Engineering (ICSE '15)*, pages 902–912, 2015.

[147] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proc. of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID '09)*, pages 124 – 131, 2009.

[148] omer727/FacebookWidget. `https://github.com/omer727/FacebookWidget`.

[149] A. Oulasvirta, S. Tamminen, V. Roto, and J. Kuorelahti. Interaction in 4-second bursts: The fragmented nature of

attentional resources in mobile hci. In *Proc. of the Conference on Human Factors in Computing Systems (CHI '05)*, pages 919–928, 2005.

[150] H. Pirkul and V. Jayaraman. A multi-commodity, multi-plant, capacitated facility location problem: formulation and efficient heuristic solution. *Computers and Operations Research (COR '98)*, 25:869–878, 1998.

[151] I. Poupyrev and S. Maruyama. Tactile interfaces for small touch screens. In *Proc. of the ACM Symposium on User Interface Software and Technology (UIST '03)*, pages 217–220, 2003.

[152] Processes and Threads. `http://developer.android.com/guide/components/processes-and-threads.html`.

[153] Profiling with Traceview and dmtracedump. `http://developer.android.com/tools/debugging/debugging-tracing.html`.

[154] H. Qian, D. Medhi, and K. Trivedi. A hierarchical model to evaluate quality of experience of online services hosted by cloud computing. In *Proc. of the IFIP/IEEE International Symposium on Integrated Network Management (IM '11)*, pages 105–112, 2011.

[155] L. Qian, Z. Luo, Y. Du, and L. Guo. Cloud computing: An overview. In *Proc. International Conference on Cloud Computing (CloudCom '09)*, pages 626–631, 2009.

[156] Z. Qin, Y. Tang, E. Novak, and Q. Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *Proc. of the ACM/IEEE International Conference on Software Engineering (ICSE '16)*, 2016.

[157] L. Qiu, V. Padmanabhan, and G. Voelker. On the placement of web server replicas. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM '01)*, volume 3, pages 1587–1596, 2001.

[158] Railag/rtweel: A lightweight twitter android application. `https://github.com/Railag/rtweel`.

[159] J. Ramsay, A. Barbesi, and J. Preece. A psychological investigation of long retrieval times on the world wide web. *Interacting with Computers (IwC '98)*, 10(1):77–86, 1998.

[160] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '14)*, pages 190–203, 2014.

[161] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *Proc. of the USENIX Conference on Operating Systems Design and Implementation (OSDI '12)*, pages 107–120, 2012.

[162] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 85–100, 2013.

[163] Robospice. `https://github.com/stephanenicolas/robospice`.

[164] Robotium - User scenario testing for Android. `http://www.robotium.org`.

[165] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galan. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):1–11, 2009.

[166] V. Roto and A. Oulasvirta. Need for non-visual feedback with long response times in mobile hci. In *Special Interest Tracks and Posters of the International Conference on World Wide Web (WWW '05)*, pages 775–781, 2005.

[167] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-efficient sampling for performance prediction of configurable systems. In *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*, pages 342–352, 2015.

[168] M. Selakovic and M. Pradel. Performance issues and optimizations in JavaScript: An empirical study. In *Proc. of the International Conference on Software Engineering (ICSE '16)*, 2016.

[169] P. Selvidge. How long is too long for a website to load. *Usability News*, 1(2), 1999.

[170] Z.-J. M. Shen. A multi-commodity supply chain design problem. *IIE Transactions*, 37(8):753–762, 2005.

[171] B. Shneiderman. Response time and display rate in human performance with computers. *ACM Computing Surveys (CSUR '84)*, 16(3):265–285, 1984.

[172] smali: An assembler/disassembler for android's dex format. `https://code.google.com/p/smali/`.

[173] S. Srinivasan, F. Buonopane, S. Ramaswamy, and J. Vain. Verifying response times in networked automation systems using jitter bounds. In *Proc. of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW '14)*, pages 47–50, 2014.

[174] A. Stage and T. Setzer. Network-aware migration control and scheduling of differentiated virtual machine workloads. In *Proc. of the ICSE Workshop on Software Engineering Challenges of Cloud Computing (ICSE CLOUD '09)*, pages 9–14, 2009.

[175] D. D. Suthers, R. Vatrapu, R. Medina, S. Joseph, and N. Dwyer. Beyond threaded discussion: Representational guidance in asynchronous collaborative learning environments. *Computer & Education*, 50(4):1103–1127, 2008.

[176] V. Terragni, S.-C. Cheung, and C. Zhang. Recontest: Effective regression testing of concurrent programs. In *Proc. of the International Conference on Software Engineering (ICSE '15)*, pages 246–256, 2015.

[177] Testing Support Library - UI Automator. `https://developer.android.com/tools/testing-support-library/index.html#UIAutomator`.

[178] P. N. Thanh, N. Bostel, and O. Péton. A dynamic model for facility location in the design of complex supply chains. *International Journal of Production Economics*, 113(2):678 – 693, 2008.

[179] UI/Application Exerciser Monkey. `http://developer.android.com/tools/help/monkey.html`.

[180] H. Van, F. Tran, and J.-M. Menaud. Autonomic virtual resource management for service hosting platforms. In *Proc.*

of the ICSE Workshop on Software Engineering Challenges of Cloud Computing (ICSE CLOUD '09), pages 1–8, 2009.

[181] S. van der Burg, M. de Jonge, E. Dolstra, and E. Visser. Software deployment in a dynamic cloud: From device to service orientation in a hospital environment. In *Proc. of the ICSE Workshop on Software Engineering Challenges of Cloud Computing (ICSE Cloud '09)*, pages 61–66, 2009.

[182] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review (CCR '09)*, 39(1):50–55, January 2009.

[183] vincent-d/BeerMusicDroid. `https://github.com/vincent-d/BeerMusicDroid`.

[184] M. Vouk. Cloud computing - issues, research and implementations. In *Proc. of the International Conference on Information Technology Interfaces (ITI '08)*, pages 31–40, 2008.

[185] E. Walker. benchmarking amazon ec2 for high-performance scientific computing. *Usenix*, 33(5):18–23, 2008.

[186] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *Proc. of the IEEE International conference on computer communications (INFOCOM '10)*, pages 1–9, 2010.

[187] Q. Wang, M. G. Jorba, J. M. Ripoll, and K. Wolter. Analysis of local re-execution in mobile offloading system. In *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE '13)*, pages 31–40, 2013.

[188] Q. Wang, H. Wu, and K. Wolter. Model-based performance analysis of local re-execution scheme in offloading system. In

Proc. of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13), pages 1–6, 2013.

[189] A. Wert, J. Happe, and L. Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *Proc. of the International Conference on Software Engineering (ICSE '13)*, pages 552–561, 2013.

[190] D. Wigdor, S. Williams, M. Cronin, R. Levy, K. White, M. Mazeev, and H. Benko. Ripples: Utilizing per-contact visualizations to improve user interaction with touch displays. In *Proc. of the ACM Symposium on User Interface Software and Technology (UIST '09)*, pages 3–12, 2009.

[191] H. Wu, Q. Wang, and K. Wolter. Tradeoff between performance improvement and energy saving in mobile cloud offloading systems. In *Proc. of the IEEE International Conference on Communications Workshops (ICC '13)*, pages 728–732, 2013.

[192] Xposed Module overview. `http://repo.xposed.info/module-overview`.

[193] Xposed Module Repository. `http://repo.xposed.info`.

[194] D. Yan, S. Yang, and A. Rountev. Systematic testing for resource leaks in android applications. In *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE '13)*, pages 411–420, 2013.

[195] X. Yu, S. Han, D. Zhang, and T. Xie. Comprehending performance from real-world execution traces: A device-driver case. *ACM SIGPLAN Notices*, 49(4):193–206, 2014.

[196] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda. Panappticon: Event-based tracing to mea-

sure mobile application and platform performance. In *Proc. of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '13)*, pages 1–10, 2013.

[197] Q. Zhang, J. Xiao, E. Grses, M. Karsten, and R. Boutaba. Dynamic service placement in shared service hosting infrastructures. In *Proc. of the International IFIP TC6 Networking Conference (NETWORKING'10)*, pages 251–264, 2010.

[198] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki. Performance prediction of configurable software systems by fourier learning. In *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*, pages 365–373, 2015.