Whom to Blame? Automatic Diagnosis of Performance Bottlenecks on Smartphones

Yi Gao, Member, IEEE, Wei Dong, Member, IEEE, Haocheng Huang, Jiajun Bu, Member, IEEE, Chun Chen, Member, IEEE, Mingyuan Xia, Student Member, IEEE, Xue Liu, Member, IEEE

Abstract—The past decade has witnessed a tremendous growth in the variety and complexity of mobile applications (apps). Although considerable amount of efforts have been spent to improve app performance, smartphones nowadays still face many performance challenges. We discover that the resource contention of multiple running apps, caused by resource bottleneck(s), is a key factor that affects the smartphone performance. In this paper, we present APB, an Automatic tool that detects Performance issues caused by resource Bottleneck(s) on commodity Android smartphones. APB employs an innovative bottleneck-hypersurface model to quantify performance issues given a specific system state. Then based on the model, APB identifies a list of apps that contribute most to the resource contention, which can well inform the end user to take action such as killing background apps to resolve the performance issue. We implement APB on commodity Android platforms and widely evaluate its effectiveness with real user studies. Results show that APB outperforms three baseline approaches and helps users to improve smartphone performance by 10% to 67%, with less than 1% runtime overhead.

Index Terms—Mobile applications, performance diagnosis, resource bottlenecks.

1 INTRODUCTION

The past decade has witnessed a tremendous growth in the mobile application (app) market. As of July 2013, the Google Play store reached over 1 million apps released and over 50 billion downloads [1]. With the growth of app market, the variety and complexity of apps also grow rapidly. Figure 1 shows the sizes of four popular apps (1.46 billion end users in total) grow by 88% to 335% in two years. Although considerable amount of efforts have been spent to improve app performance, smartphones nowadays still face many performance challenges. We discover that performance issues can be caused by performance bugs in an app or the resource contention of multiple running apps.

A recent measurement study [2], [3] reveals several patterns of the performance bugs (e.g., lengthy operations in main threads, computation for invisible GUI) and proposes a static program analysis based tool to detect these performance bugs. In the case of resource contention, one or multiple resources (e.g., CPU, Memory) become performance bottlenecks and cause performance issues. Compared with performance bugs, it is more challenging for an ordinary user to find out the real resource bottleneck and the app(s) causing it. For example, a recent work studies memory leak problem [4] and finds out that the memory leak problem of some apps can cause energy and performance problems. For an ordinary user, it is challenging to diagnose such performance issues without professional knowledge and tools. Therefore, it is desirable that the smartphone itself can automatically diagnose performance issues and give accurate and effective advices to help the end users improve the smartphone performance.

1.1 Bottleneck Behavior of Performance Issues

Figure 2 shows two examples of the bottleneck behavior. Figure 2(a) shows the CPU utilization of a smartphone over about 200 seconds and the corresponding performance metric (i.e., relative frame rate in this example) during the same period. From time 0 to time 190, the CPU utilization grows from about 10% to 40%. However, the performance does not change much. At time $t_1$, the CPU utilization grows rapidly and becomes the performance bottleneck, causing the performance to degrade significantly. Similar behavior is shown in Figure 2(b), which gives the bottleneck behavior of memory. The bottleneck behavior poses...
difficulty to determine which app is the root cause when an end user is experiencing performance issues. First, it is challenging to determine that which resources are the performance bottlenecks. The bottleneck resources could be various in smartphones with different operating system versions, different hardware and different installed apps. Further, a smartphone with performance issues could be caused by multiple performance bottlenecks. Identifying these bottlenecks and their contributions to the performance issue is challenging. Second, even if we know the bottleneck resources and the apps which are occupying these resources, it is still challenging to quantify the impact of each app to the overall performance metrics. In order to address the above challenges, we first need to know the bottleneck resources for each smartphone. Then we need a mechanism to accurately determine which apps occupying the bottleneck resources are the root cause of performance issues. Finally, we need an qualitative approach to rank the apps which cause the performance bottleneck and give advice to the users.

1.2 Our Contribution

In this paper, we present APB, which automatically diagnoses performance bottlenecks on smartphones and helps the end users catch the resource intensive apps which cause performance issues. APB is a stand-alone user space application running with other applications. It monitors the resource utilizations as well as the performance metrics of the smartphone at runtime. Then based on an innovative model, APB calculates an ordered list of apps and informs the user to take actions (such as closing some resource-intensive apps) for solving the performance issues. APB has two salient features, First, APB requires zero changes of the system or other apps, making it easier to deploy. Second, APB is lightweight in terms of runtime overhead. It is important since we do not want the performance diagnosis tool APB to introduce potential performance issues itself. The basic idea of APB is as follows.

By tracking the resource utilizations (e.g., CPU, memory, I/O) as well as the user perceived performance metrics (e.g., frame rate, which directly affects the responsiveness), APB models the correlation of these resource utilizations and the performance metrics. Then, a resource utilization state can be represented by an n-dimensional resource vector, where n is the number of different resources. Based on the model, APB further obtains a hypersurface with n − 1 dimensions. The hypersurface separates the n-dimensional resource utilization space into two sub-spaces, one represents normal performance state and the other represents system state with performance issues. In other words, this hypersurface can be viewed as a high dimensional change point or bottleneck. We refer to it as bottleneck-hypersurface in this paper. Figure 3 shows an example, in which there are two kinds of resources (i.e., n=2), CPU and memory. The resource utilization space is two dimensional and a point in the space represents a resource utilization state of the smartphone. The bottleneck-hypersurface is one dimensional, i.e., a plane curve. When the resource utilizations are lower than the bottleneck-hypersurface, there is no performance issue. When the resource utilizations are higher than the bottleneck-hypersurface, the performance drops quickly. In the figure, s₁ and s₂ are two resource utilization states, which represent a normal state and a state with performance issue. Based on the performance model, APB is able to find out the apps which cause the performance issues and give accurate advices to help the end users improve the performance of their smartphones.

We implemented APB on commodity Android platforms and evaluated APB by in-lab experiments and user study. 1) We manually install apps with performance issues (based on reported performance bug patterns in [2], [3]) on multiple smartphones with various hardware capabilities. Then we use APB to diagnose performance issues. The detection and diagnosis accuracy are reported by comparing the results with the ground truth. 2) User study. Ten volunteers with various smartphones use APB to improve the performance of their smartphones. Results show that APB outperforms three baseline approaches and helps users to improve smartphone performance by 10% to 67%. We also evaluate the overhead of APB. On average, APB utilizes about 0.91% of the CPU and about 8.13MB memory. The storage overhead and network overhead are negligible (4.53KB-5.91KB per hour).

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 describes the design and implementation of APB in detail. Section 4 presents the evaluation of APB, and finally, Section 5 concludes this paper.

2 RELATED WORK

The performance of smartphones has been attracted a great attention in both academia and industry, since it is directly related to the end user experiences [5], [6], [7]. Depend on the targeted users of the proposed tools/approaches, these works can be divided into two categories, developer oriented and end user oriented (or both).
There are a large number of developer oriented tools/approaches [8] proposed in the recent years, which focus on helping mobile app developers improve the performance of their apps. Several commercial tools [9], [10], [11] are dedicated to provide assistance for app developers to design apps with better performance. In academia, there are more approaches from different perspectives to help developers design better apps. Appinsight [12] instruments app binaries to trace the internal behaviors of user transactions. Then Appinsight uses the traced data to analyze the critical execution paths during the user transactions, which disclose possible root causes for performance issues. Panappticon [13] modifies the Android system code to trace even more low level interactions and is able to reveal more kinds of performance issues. A recent measurement study [2] examines 70 real world performance bugs and analyzes the bug types, impacts, manifestation, fixing effort, and common patterns. This work also provides an automatic source code checker based on the found common pattern. ARO [14] exposes the cross-layer interaction among various layers and detects radio resource and energy inefficiencies. Mantis [15] can predict the execution time of Android apps on given input, by using program analysis and machine learning techniques. This helps identify possible inputs which may slow down the app, so that the developers can improve the design of their app. E3 [16] analyzes the impact of scrolling operations to the power consumption of smartphones. By using smaller frame rate under relatively slow scrolling speed, E3 reduces the CPU power consumption significantly. EnTrack [17] is a fine-grained energy tracing scheme for Android system services. These tools/approaches are proposed to be mainly used by developers. APB, however, is designed to be used by end users to improve the performance of their smartphones.

Besides the above developer oriented tools/approaches, there are also end user oriented tools. The most common tool related to performance improvement in Android is the task manager. It makes end users be able to kill some background apps to release the resource for better performance. Its effectiveness, however, highly depends on whether the end user knows exactly the real root cause of the performance problem, which is usually not the case. eDoctor [18] is an end user oriented tool for diagnosing abnormal battery drain issues. By using the execution phases to capture an app’s time- varying behavior, eDoctor is able detect and diagnose abnormal battery drain issues. Powertutor [19] estimates energy consumption of smartphone apps based on power models for end users. Similar to APB, the above approaches are also proposed to be used by end user directly. Different with APB, these approaches focus on improving the energy efficiency of smartphone apps. APB focus on improving the end user experience of app performance (e.g., frame rate, response time).

There are also a large number of performance issues detection and diagnosis approaches for PC or server-side applications [20], [21], [22], [23], [24], [25], [26], [27], [28]. We discuss some representative approaches in recent years. Jin et al. [20] conducted a comprehensive study of 110 real-world performance bugs from five representative software suites, and identified efficiency rules for detecting these performance bugs. Xu et al. [25] introduced a run-time analysis to discover low-utility data structures, which have high-cost and bring little benefit to a program’s output. Xiao et al. [26] proposed a tool called DeltaInfer to predict workload-dependent loops under large workloads via inferring iteration counts of workload-dependent loops using complexity models for the workload size. Nistor et al. [27] studied how performance bugs were discovered, reported to developers, and fixed by developers. Then they obtained several interesting findings, such as fixing performance bugs were more difficult than fixing non-performance bugs. Log2 [28] is a cost-aware logging system for performance diagnosis. Given a budget, it makes the “whether to log” decision through a two-phase filtering mechanism.

3 APB: DESIGN AND IMPLEMENTATION

In this section, we give the design and implementation of APB in detail. Figure 4 shows the overall architecture of APB. At the device side, a data collector records system states of the smartphone, including various resource utilizations, app upgrade/install events and performance metrics. The collected data are transmitted to the PC/Cloud side. Then the model builder at the PC/Cloud side generates a model that exploits the correlation among the resource utilizations and the performance metrics, for each smartphone. The advisor at the device side takes two kinds of data as input, the collected system states from the data collector and the model from the PC/Cloud side. Then the advisor identifies a list of apps and ranks them by their contributions to the resource contention, which can well inform the end user to take action such as killing background apps to resolve the performance bottleneck(s).

3.1 Data Collector

APB collects three kinds of data of the smartphone: 1) resource utilizations, including the resource utilization of each background app and the overall resource workload; 2) app change events, such as app updates and installation; 3) performance metrics, such as frame rate. In order to protect the user privacy, all kinds of data are sent to the PC/Cloud after anonymization.

Resource utilizations. APB monitors the resource utilizations of the smartphone, including both per app utilization and the overall utilization. The collected resources are CPU utilization, memory utilization, I/O frequency, and etc. These resource utilizations are related to the overall performance of the smartphone. In our implementation, APB takes advantage of the resource utilization tracking mechanism in Android. For example, APB reads the “/proc/stat” file to track the realtime CPU utilization of the overall system and each app. For memory utilization, APB uses the “dumpsys meminfo” command to obtain the realtime memory utilization of each app.

App change events. Performance issues are sometimes due to the latest app change event, such as installation or upgrading. Therefore, APB keeps tracking the all change events...
to further analysis. In our implementation, APB listens to the BroadcastReceivers related to app installation and upgrading.

Besides app installation and upgrading, some configuration changes can also cause a performance issue. Most apps use components (e.g., SharedPreferences) provided by the Android system to manage the configurations. Therefore, modifying these components can track most configuration changes. However, in order to keep zero change of the system, APB does not track these configuration changes directly. Instead, based on the historical behaviors (i.e., resource utilization pattern) of the app, APB infers possible configuration changes and provides this information to end user in the advisor.

**Performance metrics.** The performance metrics are those the end user can experience, such as response time and frame rate. Response time is an important performance metric for user experience. In order to measure response time, there have been many different approaches proposed in the literature. For example, AppInsight [12] instruments the app binary to track the response time of each user transaction. In [2], a source code analysis tool is proposed to detect lengthy operations in the UI thread. However, without modifying the binary or source code of an app, it is very challenging to inspect the response time of a foreground app by a background app due to process protection. Therefore, in our implementation, APB uses the atrace tool to track the realtime frame rate of the smartphone for fast performance issue detection and further diagnosis. However, in practice, the frame rate cannot not be directly used as a performance metric. An intuitive explanation is that the frame rate can be smaller than 60Hz due to two different reasons, there is no new frame to draw and there are frames failed to be drawn. Only the second case indicates performance issues. In order to exclude the impact of the first case, APB traces both the frame rate and the vsync frequency.

The vsync signal is used to synchronize the frame generation and the screen refreshing, which is an important mechanism in Android to improve its GUI performance. Figure 5 is a conceptual figure of the collected frame rate and vsync frequency. A key component in this figure is the SurfaceFlinger which is at the center of Android’s GUI system. The SurfaceFlinger includes a vsync source which is based on an original vsync source. When the foreground app needs to generate a new frame, it sends a request to the vsync source in the SurfaceFlinger. Then the vsync source broadcasts a vsync signal to enable frame generation and the following frame drawing. Therefore, the vsync frequency broadcasted by the vsync source in the SurfaceFlinger represents the frame rate which the foreground app requires at that time. Then APB uses the ratio (i.e., normalized frame rate) of the actual frame rate and the vsync frequency as the performance metric. The maximum normalized frame rate is 1. When this normalized frame rate is below 1, there is a performance issue (i.e., frame drop). Figure 6 gives an example, which shows the frame rate, the vsync frequency and the normalized frame rate over time. At time $t_1$, both the frame rate and the vsync frequency drop and the frame rate drops more significantly. Therefore, the normalized frame drops, indicating that there is a performance issue. At time $t_2$, although the frame rate drops, the normalized frame rate is still 1, indicating that there is no user perceived performance issue.

Before sending the collected data to the PC/Cloud side, the collector filters out some data polluted by the garbage collection (GC) operations in the Android system. GC operations are triggered automatically to retrieve memory occupied by unreferenced Java objects. During the GC operations, the memory utilization and the performance metrics change rapidly in a short period of time, which will introduce inaccuracies to the model builder. Figure 7 shows an example. At time $t$, we open two apps and the CPU utilization grows as expected. However, the memory utilization drops when we open these two apps, which is caused by the GC operations. The GC operation causes the performance drops significantly at that time. In order to avoid the inaccuracies introduced by the GC operations, the resource utilization samples with recent GC operations will be filtered out before being sent to the PC/Cloud side.

Then the collected and filtered data is sent to the PC/Cloud side. In the current implementation, APB prefers using WiFi connection to avoid incurring high cellular data usage.

![Fig. 5. A conceptual figure of the collected frame rate and vsync frequency.](image)

**3.2 Model Builder**

In order to describe the model builder and the advisor (in the following subsection) formally, we introduce the notations used as follows.

- $n$ kinds of resources, e.g., $u^i_t = [u^i_{1t}, ..., u^i_{nt}]$ are all resource utilizations of app $i$ at time $t$;
- $m$ apps are running in the system, e.g., $u^i_{1t} \sim u^i_{mt}$ are the $m$ apps’ utilizations of resource $i$ at time $t$;
- $u^i_j$, for app $i$, its utilization of resource $j$, at time $t$;
- $x^j_t$ is the overall utilization of resource $j$, at time $t$. It is also the resource utilization state of the smartphone.
the model builder calculates the model frame rate, since multiple performance metrics can be modeled to the performance of the phone. The model builder takes the resource utilization vectors and the performance metric as input. To evaluate its performance. Results show that it takes more than reason of not implementing it on smartphones is mainly due to performance issue.

The model builder is implemented at the PC/Cloud side. The model builder first eliminates performance samples whose frame rate $F^t$ is the maximum value, i.e., no performance issue. For example, Figure 8 shows a real trace of the frame rate and the CPU utilization. There will be inaccuracy if we use all data points to build the model (green line). As shown in the figure, using all data points will locate $c_1$ as the bottleneck, which is not accurate. Therefore, APB eliminates the data points when there is no performance issue (i.e., data points when the normalized frame rate is equals to one) and builds the model (blue line) by the rest of the data points, which is more accurate to locate the real bottleneck (i.e., $c_2$).

In order to improve the accuracy of the model, the model builder uses least square fitting to generate the model $\hat{F} = \mathcal{F}(X)$ based on the performance samples. In the current implementation of APB, we use a quadratic function to present the model. Let $\alpha_1, \alpha_2, \ldots$ be the parameters of the function $\mathcal{F}$. The following equation uses a number of performance samples $S^t = \{X^t, F^t\}$ to calculate the parameters $\alpha_i$ of $\mathcal{F}$.

$$[\alpha_1, \alpha_2, \ldots] = \arg\min_{[\alpha_1, \alpha_2, \ldots]} \sum_k (\mathcal{F}(X^{t_k}; \alpha_1, \alpha_2, \ldots) - F^t)^2.$$  \hspace{1cm} (1)

Since the maximum normalized frame rate is 1, the bottleneck-hypersurface can be represented by $\mathcal{F}(X) = 1$. It is actually the $n - 1$ dimensional intersection of two n-dimensional surfaces, $\hat{F} = \mathcal{F}(X)$ and $\hat{F} = 1$. Figure 9 shows a real example of a Nexus 4 smartphone, where the two resources are CPU and memory (we cannot plot the model when there are more than two resources). Every data point shown in the figure represents a performance sample at time $t$, including the normalized frame rate $F^t$ and the resource utilization vector $X^t$ at time $t$. Using a number of samples $(S^{t_1}, S^{t_2}, \ldots, S^{t_k})$, we can calculate the model $\mathcal{F}(-)$;

- $X^t$ is the overall resource utilization vector at time $t$, i.e., $X^t = (x^t_1, x^t_2, \ldots, x^t_k)$;
- $F^t$ is the normalized frame rate at time $t$;
- $\hat{F}^t$ is the estimated normalized frame rate at time $t$, which can be modeled by the resource utilization vector $X^t$, i.e., $\hat{F}^t = \mathcal{F}(X^t)$;
- $S^t = \{F^t, X^t\}$ is a performance sample at time $t$, including the normalized frame rate $F^t$ and the resource utilization vector $X^t$ at time $t$. Using a number of samples $(S^{t_1}, S^{t_2}, \ldots, S^{t_k})$, we can calculate the model $\mathcal{F}(\cdot)$;
- $\mathcal{H}$ is the bottleneck-hypersurface which represents the performance bottleneck of the smartphone.

The model builder is implemented at the PC/Cloud side. The reason of not implementing it on smartphones is mainly due to the performance overhead of the model builder. Concretely, we have also implemented the model builder on a typical phone and evaluated its performance. Results show that it takes more than 5 seconds to complete, which indicates a non-negligible impact to the performance of the phone. The model builder takes the resource utilization vectors and the performance metric as input. In this paper, we only consider one performance metric, i.e., the frame rate, since multiple performance metrics can be modeled separately. Then the model builder calculates the model $\mathcal{F}(\cdot)$ and the bottleneck-hypersurface $\mathcal{H}$ for the smartphone.

### 3.3 Performance Improvement Advisor

Based on the received model $F = \mathcal{F}(X)$ and bottleneck-hypersurface $\mathcal{F}(X) = 1$, the advisor at the device side first calculates the performance gain $g_i$ of each app $i$. The performance gain of an app indicates the performance improvement if the end user takes actions (downgrade/uninstall or close) about that app. Then the advisor provides the end user with an ordered list of apps according to their performance gain.
The performance gain of an app is affected by the resource vector of the app. Figure 10 shows an illustrative example. Each point in the plane represents a resource utilization state in terms of the CPU and memory utilizations of the smartphone. Assume $H$ is the bottleneck-hypersurface of the smartphone, which divides the resource utilization state into normal (i.e., bottom left part) and with performance issues (i.e., upper right part). The current resource utilization state $s_c$ (i.e., a vector of overall resource utilizations of all resources $[x_1, x_2, ..., x_n]$) is in the upper right part of the plane, indicating that there is a performance issue. Suppose there are two apps, app A and app B, which are CPU intensive and memory intensive, respectively. The resource utilization state $s_A$ in the figure represents the new state if app A is closed and all its resources are released. Similarly, the resource utilization state $s_B$ is the state if app B is closed. Since app A is CPU intensive, closing app A will make the new resource state move left (i.e., the direction of CPU). Similarly, closing app B will make the new resource state move down (i.e., the direction of memory). From this example, we can see that the new resource utilization state after closing app A is still in the upper right part, indicating that the performance issue still exists. The reason is that the bottleneck-hypersurface in this example suggests that the performance bottleneck of this smartphone is mainly the CPU, instead of the memory. Therefore, closing the memory intensive app B is not able to improve the performance much.

Then we introduce how to calculate the performance gain $g_i$ of an app $i$ formally. As given in the previous subsection, we use $u_{ij}$ to represent app $i$’s utilization of resource $j$, and $n, m$ to represent the number of resources and apps, respectively. Let $Dist(s_k, H)$ be the distance of a resource utilization state $s_k$ to the bottleneck-hypersurface $H$. Then the performance-gain of the app can be represented by the distance change if the app is closed. Let $Dist(s_k, H)$ be positive when $s_k$ is in the normal state and be negative when $s_k$ is a state with performance issues. In the example shown in Figure 10, $Dist(s_A, H)$ is positive, and $Dist(s_c, H)$ and $Dist(s_B, H)$ are negative. The distance change (i.e., the performance gain $g_A$) after closing app A can be calculated as the following:

$$g_A = Dist(s_c, H) - Dist(x - u_A, H),$$

where $s_c$ is the current resource utilization state, and $x - u_A$ is the resource utilization state after closing app A (i.e., $s_A$).

Calculating $Dist(s, H)$, given the resource utilization state $s = [x_1, x_2, ..., x_n]$ and the bottleneck-hypersurface $H$, is generally a computation intensive task. In order to avoid introducing high computational overhead, APB uses an efficient approximation. The approximation is based on the following two observations. First, a resource utilization state $s$ with performance issues is adjacent to the bottleneck-hypersurface. One reason is that when $s$ is far away above the bottleneck-hypersurface, the performance of the smartphone will be extremely poor, making the smartphone unusable. Second, the bottleneck-hypersurface is typically monotonic. When more resources are occupied, the performance degrades. Then the approximation works as follows.

1) The advisor calculates $n$ points $\{s_1^H, s_2^H, ..., s_n^H\}$ on the bottleneck-hypersurface. For each $s_j^H$, it is the intersection of $H$ and one n-dimensional line which satisfies the following conditions. First, the line includes $s$ and intersects with the axis of resource $j$. Second, the line is perpendicular to the axis of resource $j$. According to these two conditions, each line with $s_j^H$ can be represented as follows.

$$\begin{align*}
x_1 &= x_1^s, \\
x_2 &= x_2^s, \\
... &= ... \\
x_i &= x_{i}^s, \\
x_{j} &= x_{j}^s + 1, \\
x_{j+1} &= x_{j+1}^s + 1, \\
... &= ... \\
x_n &= x_n^s.
\end{align*}$$

Then the intersection points can be calculated by combining the above lines and the bottleneck-hypersurface. Figure 11 gives an example when $n = 2$. Two intersection points $\{s_1^H, s_2^H\}$ are obtained.

2) The $n$ points in the n-dimensional resource utilization space determine a hyperplane $H^*$ which includes all these points. Let the hyperplane be $f(x) = w^T x + b$, where $w^T$ is an n-dimensional coefficient vector and $b$ is a constant vector, which can be calculated from the $n$ intersection points. Then the approximated distance $Dist^*(s, H)$ between $s$ and $H$ is the distance between $s$ and $H^*$. The approximated distance can be calculated as follows.

$$Dist^*(s, H) = \frac{w^T x^s + b}{|w||}.$$
closing it will improve the performance more significantly. Since performance issues are often caused by apps recently installed/upgraded/reconfigured, the advisor also shows the apps which are recently installed/upgraded/reconfigured, leaving the decision to the end user. As mentioned in the introduction section, APB uses multiple BroadcastReceivers to capture the app installed/upgraded. In addition, in order to capture the app reconfiguration events without modifying the OS, the advisor uses the resource utilization vector of the app to infer possible app reconfigurations.

4 Evaluation

In this section, we present the evaluation methodology and results of APB. We first use in-lab experiments to evaluate the detection accuracy of APB. Then we compare the performance of APB and three baseline approaches. Finally, we use user studies to evaluate the performance improvement effectiveness of APB, including three concrete case studies.

4.1 Methodology

The in-lab experiments to evaluate the accuracy are conducted on four different smartphones with various configurations. We implement several problematic apps in terms of various resource utilization issues. For example, one app with CPU utilization problem does intensive calculation randomly at the background. Then we install these problematic apps and APB on each smartphone. In total, we collect 16 traces from the four smartphones. Table 1 shows more information about the CPU and memory resource of these four phones. For each smartphone, we install the problematic apps in four different combinations. When the problematic apps actually cause performance degradation of the smartphone, we record whether APB can accurately find the problematic apps and report them in the advisor.

We also evaluate the accuracy of the approximation method mentioned in the design section. The approximation method is used to calculate the distance from a resource utilization state to the bottleneck-hypersurface. We use a desktop PC to calculate the ground truth of the distance and compare it with the approximated value.

Ten volunteers install our APB app on their smartphones to detect and diagnose performance issues. APB first collects resource utilization samples and performance metrics and send the data to the PC/Cloud side. The model builder at the PC/Cloud side calculates the bottleneck-hypersurface for each smartphone and sends it back. Then based on the bottleneck-hypersurface and the resource utilization of each app, the advisor of APB displays a list of ranked suspicious apps. The end users may close the apps on the list to improve the performance of their smartphone. Although closing an app at the top of the list can improve the performance most significantly, the end users not always close the app at the top of the list. The reason is that some apps with high resource utilizations will be frequently accessed by the user. The end user does not want to close them in order to avoid losing the context. We report the app list in the advisors, the user actions, and the performance metrics during the user study.

4.2 Accuracy

Accuracy of the model builder. We first evaluate the accuracy of the model builder, which calculates the bottleneck-hypersurface $\mathcal{H}$. Figure 13 shows a trace of the performance, CPU utilization, memory utilization, the bottleneck-hypersurface $\mathcal{H}$ and six resource utilization transitions during the experiment period. For example, in the first transition (i.e., transition (1)), an app with high CPU utilization is opened, causing the performance degradation. In the second transition, the problematic app is closed and high memory utilization in the fifth transition. Then we close that app, we open an app with both high CPU and high memory utilizations. We can observe high resource utilizations at time $t_4$ after the third transition. After the fourth transition, in which we close that app, we open an app with high memory utilization in the fifth transition. Then we close it in the last transition. As shown in the six sub-figures at the bottom of Figure 13, all of these transitions across the bottleneck-hypersurface $\mathcal{H}$. Therefore, the model builder of APB accurately models the bottleneck-hypersurface $\mathcal{H}$.

Accuracy of the advisor. We then evaluate the accuracy of the advisor. Four apps with High CPU utilization (HC),

<table>
<thead>
<tr>
<th>Apps/Smartphone</th>
<th>phone1</th>
<th>phone2</th>
<th>phone3</th>
<th>phone4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCHM</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>HCM</td>
<td>1.3</td>
<td>1.2</td>
<td>1.4</td>
<td>1.2</td>
</tr>
<tr>
<td>MCM</td>
<td>2.1</td>
<td>3.1</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>MCMM</td>
<td>n/a</td>
<td>3.1</td>
<td>1.4</td>
<td>1.2</td>
</tr>
</tbody>
</table>

TABLE 2

Accuracy of the advisor

Fig. 12. Screen snapshots of APB advisor running on an Android device.
Fig. 13. The performance of a smartphone during an experiment, which shows the accuracy of the model builder. At \(t_1, t_3, t_5, t_7\), the performance is satisfactory. At \(t_2, t_4, t_6\), the performance degrades significantly. The CPU and memory utilizations are also shown in the figure. At \(t_2, t_4, t_6\), the resource utilization grows due to the opening of the problematic apps. The performance becomes normal when we close these apps. The six small sub-figures show the resource utilization state transitions. The normal states and the states with performance issues are separated by the bottleneck-hypersurface \(H\).

Medium CPU utilization (MC), High Memory utilization (HM) and medium memory utilization (MM) are installed in the four smartphones. These four smartphones have different CPU and memory capacities (smartphone 1 has the highest CPU frequency and the largest memory). Then we report their rank in the list generated by the advisor in APB. Table 2 shows the results. Each entry of the table represents the rank of the two problematic apps in the list generated by the advisor. For example, the first entry (1,2) in the first row means that the two problematic apps are listed as the most suspicious apps in the advisor. In smartphone 1, which has the highest CPU frequency and the largest memory, the performance does not degrade significantly when app MC and MM are installed (i.e., the first entry “n/a” of the fourth row). The reason is that these two apps do not use too much resources and the resource utilization state of the smartphone does not surpass the bottleneck-hypersurface much. In the rest 15 cases which have performance degradations, the advisor of APB successfully catches the two problematic apps with high accuracy. In all the 15 cases, at least one problematic app is diagnosed as the most suspicious app. And in 10 cases, the advisor successfully catches the two apps as the top two suspicious apps. These results show that APB is able to find the problematic apps accurately.

Figure 14 gives more insights about the above 16 experiments. Each sub-figure shows the resource utilization of one problematic app combination as well as the performance change before and after closing them in the four smartphones. For example, Figure 14(a) shows the results of app HCHM (high CPU utilization and high memory utilization) on four smartphones. The first three columns show the results of that app on phone1. As we can see, the performance of phone1 can be improved by 27.5% when closing this app. The CPU and memory utilization of this app is 40.6% and 16.1%, respectively.

From Figure 14, we can have the following observations. First, app combination MCMM utilizes less resources and causes smaller performance change to the four smartphones (Figure 14.d). Second, for the same app, its resource utilizations are different on different smartphones. One reason could be the different resource management mechanisms on different smartphones. Third, for app combination HCHM (a), MCHM (c) and MCMM (d), the performance change on smartphone 1 is not as significant as on the other three smartphones. This is reasonable since smartphone 1 has the highest frequency CPU and the largest memory, causing it less sensitive to resource utilization change. However, for app combination HCMM (Figure 14(b)), the performance change on smartphone 1 is higher than that on other smartphones. This contradicts to the third observation mentioned above. We look into the trace and find out that the reason is the GC (i.e., garbage collection) operations in Android. The GC operation causes the
4.3 Comparison with Baseline Approaches

We also compare the performance of APB with three baseline approaches. The first two baseline approaches always choose the apps with the most CPU/Memory utilizations as the suspicious apps (i.e., using the task-manager or the “top” command to identify suspicious apps). In the third baseline approach, we use a task managing tool called “Smart Task Manager” to rank the importance of currently running apps. This tool will order the apps according to whether the app is foreground, whether the app is visible, whether the app is a service, and whether the app is in the background.

Given a phone with performance issues, we use APB and these three base approaches to identify suspicious apps and kill these apps. Then we compare the number of killed apps before the performance issue is resolved. The experiments are conducted on the four phones and repeated ten times with different apps running.

Figure 15 shows the results. On these four phones, APB always kill the smallest number of apps to resolve the performance issue. We can also observe that many apps are killed by the User-CPU approach on phone2 and by the User-Memory approach on phone3. The reason is that phone2 is a memory constrained phone. However, the User-CPU always kills apps with high CPU utilization, causing poor performance (large number of killed apps). Similarly, the memory-approach performs poorly on phone3 which is CPU constrained. For the User-Tool approach, it performs better on the memory constrained phone2 compared with it on other phones. The reason is that background apps without running services mainly consume memory, instead of CPU. From these

![Figure 15. Comparison of APB and three baseline approaches. User-Memory: always kill the app with the highest memory consumption. User-CPU: always kill the app with highest CPU utilization. User-Tool: use an existing task managing tool to kill apps according their importance.](image)
results, we can see that the actual performance bottleneck could be very different under different settings, and APB is able to accurately identify the current performance bottleneck and provide useful information for the users to improve the performance of their smartphones.

4.4 Effectiveness

In order to evaluate the effectiveness of APB in terms of improving the performance of smartphones, we conduct user studies by installing APB in the smartphones of ten volunteers. In total, there are 286 apps installed in the smartphones of these ten volunteers. Figure 16 gives an overview of these apps.

We install APB in these smartphones to detect performance bottlenecks. Table 3 shows nine apps, which are either listed as the most suspicious app by the advisor or closed by the end users. The apps are in various categories: security tools, communication, game, personalization and media. Six of them are listed in the advisor due to their high CPU utilizations, five of them are listed in the advisor due to their high memory utilizations and two of them are listed in the advisor due to their high I/O frequencies. Note that an app can have high utilizations of multiple resources. Six apps are listed as the most suspicious app in the advisor and three of them are closed by the end users. The users also close three apps which are not listed as the most suspicious app, but are listed as the second or third suspicious app. No app listed after the third suspicious app is closed by the users.

In the third case study, we further show three case studies. Table 4 shows the results. The top four apps listed in the APB advisor are shown in the table, ranked by the potential performance gain by killing each app. We can see the memory consumption of the app Chrome is the highest, and the CPU consumption of app Minimap is the highest. However, the potential performance gains of these two apps are different. The potential performance gain of killing app Chrome (22.3%) is much larger than that of killing app Minimap (15.4%). These results show that the current main performance bottleneck is the memory. The app Chrome will consume about 20MB memory every time the user opens a new tab, causing the performance degradation of this smartphone. In practice, the user closed Chrome and the performance improved by 20.2%, which is close to the predicted value (22.3%) by the APB advisor. This case study shows that APB is able to accurately detect the current performance bottleneck and gives useful information to the user for performance improvement.

Case study 2. There are 18 different apps installed in a phone of another user. Similar as the previous case study, Table 5 shows the top five apps listed in the APB advisor. We can see that the CPU consumption of the app QQPimSecure is the highest, and the memory consumption of the app MobileTicket is the highest. Specifically, QQPimSecure consumes about 35% of the CPU time even when it is in the background. The reason is that it will scan other apps from time to time when it is in the background. From the table, we can see that the potential performance gain of killing QQPimSecure (19.6%) is much larger than that of killing MobileTicket (6.8%). These results show that the current main performance bottleneck is the CPU, not the memory. In practice, after the user closed QQPimSecure, the performance of the phone improved by 19.9%. This first two case studies show that different phones may have different performance bottlenecks, and APB can accurately identify the performance bottleneck and effectively help the users improve the performance of their smartphones.

Case study 3. In the third case study, we further show
the impact of an important system parameter to the overall performance. This system parameter is the data collection length of the model builder. For the same smartphone of a user, we used two different settings of this parameter, 60 seconds and 300 seconds. When the data collection length is 60 seconds, Table 6 shows the results. We can see that the app Plusgpschina is expected to introduce the largest potential performance gain if it is killed. However, when the user actually kills this app, the performance gain is only 16.4%, which is not very close to the expected value 22.2%. Table 7 shows the results when the data collection length is 300 seconds. We can see that the top five apps are different compared with the case in Table 6. In this case, the app Bocmbci is expected to introduce the largest potential performance gain if it is killed. In practice, the performance gain is 22.3%, which is very close to the expected value 22.1%. This case study shows that the data collection length has non-negligible impact to the overall performance of APB. More specifically, when the data collection length is too short, the performance data and resource usage data are not sufficient for the model builder to build an accurate model of performance bottlenecks. Therefore, in order to improve the accuracy, a sufficiently large number of samples are collected and sent to the PC side for model building. Calculating the model with a large number of samples could consume non-negligible resources, which is not desirable to be implemented on the smartphones.

### 4.5 Overhead

We also evaluate the computational overhead, memory overhead and network overhead of APB. Since the model builder works at the PC/Cloud side, we focus on the overhead of the information collector and the advisor. The information collector of APB runs in the background and collects information periodically. The advisor calculates the suspicious app list in the background and be put to foreground when the end user accesses it. We evaluate the overhead when APB is running in the background, since the overhead of the foreground running advisor highly depends how frequently the user access the advisor and how the user interacts with the advisor. We use APB to measure its resource utilizations. On average, APB utilizes about 0.91% of the CPU and about 8.13MB memory. The storage consumption is negligible. Since the bottleneck model of a smartphone is relatively stable over time, the network overhead is also negligible (4.53KB-5.91KB per hour).

### 5 Conclusion

This paper addresses the performance issues detection and diagnosis problem from the perspective of ordinary end users. We built a practical tool, APB, to help end users find the “criminal” apps which degrade the performance of their smartphones. We evaluated the accuracy of APB through extensive in-lab experiments. Results show that APB is able model the bottleneck behavior of the performance issues and generates the suspicious apps accurately. We also conducted user studies to evaluate the effectiveness of APB in practice. Results show that APB helps the end user improve the performance of their smartphones significantly.

There are multiple dimensions to explore in the future work. First, we would like to investigate more resources as well as more performance metrics. Second, we would like to conduct cross user analysis based on the data collected by APB. Using resource utilizations and performance metrics of multiple smartphones should be able to further improve the accuracy of diagnosing performance issues.

### References


[29] Yi Gao (M’15) received the B.S. and Ph.D. degrees in Zhejiang University in 2009 and 2014, respectively. He is currently a research assistant professor in Zhejiang University, China. From October 2011 to October 2012, he visited McGill University as a research training student. His research interests include network measurement, wireless sensor networks and mobile computing. He is a member of the IEEE and the ACM.

[30] Wei Dong (S’08, M’12) received the B.S. and Ph.D. degrees in computer science from Zhejiang University, Hangzhou, China, in 2005 and 2011, respectively. He is currently an Associate Professor with the College of Computer Science, Zhejiang University. His research interests include networked embedded systems, network measurement and wireless sensor networks.

Haocheng Huang received the B.S degree from Zhejiang University of Technology in 2013 and the M.S degree from Zhejiang University in 2016. His research interests include android performance diagnosis and performance measurement.

Jiajun Bu (M’06) received the B.S. and Ph.D. degrees in Computer Science from Zhejiang University, China, in 1995 and 2000, respectively. He is a professor in College of Computer Science and the deputy director of the Institute of Computer Software at Zhejiang University. His research interests include embedded system, mobile multimedia, and data mining. He is a member of the IEEE and the ACM.

Chun Chen (M’08) received his Bachelor of Mathematics degree from Xiamen University, China, in 1981, and his M.S. and Ph.D. degrees in Computer Science from Zhejiang University, China, in 1984 and 1990 respectively. He is a professor in College of Computer Science, and an academician of the Chinese Academy of Engineering. His research interests include embedded systems, image processing, computer vision, and CAD/CAM.

Mingyuan Xia (S’15) received the B.S. degree from the Shanghai Jiao Tong University in 2011. He is currently a Ph.D. candidate in School of Computer Science, McGill University, Canada. His research focuses on mobile systems, program analysis and distributed file systems. He received IBM Ph.D. Fellowship in year 2015.
**Xue Liu** is an associate professor and William Dawson Scholar in the School of Computer Science at McGill University. He received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 2006. He received his B.S. degree in Mathematics and M.S. degree in Automatic Control both from Tsinghua University, China. He has also worked as the Samuel R. Thompson Associate Professor in the University of Nebraska-Lincoln and HP Labs in Palo Alto, California. His research interests are in computer networks and communications, smart grid, real-time and embedded systems, cyber-physical systems, data centers, and software reliability. Dr. Liu has been granted 4 US patents and published over 200 research papers in major peer-reviewed international journals and conference proceedings, including the Year 2008 Best Paper Award from IEEE Transactions on Industrial Informatics, and the First Place Best Paper Award Security (WiSec 2011). He is a recipient of the Outstanding Young Canadian Computer Science Researcher Prizes from the Canadian Association of Computer Science.