研究背景

在这个科技日新月异发展的时代，人们已经越来越离不开手机了。在生活中，不少人衣食住行都需要依赖于手机来完成。比如，在通讯方面，人们依赖微信， QQ 的程度远大于人们使用信件的程度。在购物方面，人们使用手机淘宝的频率也是一天天在增加。在出行方面，手机上的各种地图软件也是层出不穷，给人们带来了很大的便利。这种手机市场上百花齐放的现象也证明了现代科技的发展，以及人们生活逐渐变得更加多彩。

然而，市场上的手机软件多种多样，在使用起来也会有各种各样的体验。比如说，同样都是地图软件，百度地图、谷歌地图、高德地图等著名的地图软件使用起来的感觉可能会远远超过其他的手机地图软件，尤其是小公司生产的软件。这种使用体验上的差异可能体现在方方面面。查询结果的不准确和不全面可能是一个重要的因素。人们在出游的时候，难免会遇到找不到目的地的场合，这时候打开手机地图软件，要是无法正确进行查询的话，会导致很多的问题。还有一些其他的因素，也就是本文希望讨论的，就是手机软件在性能上的缺陷。依然拿手机地图举例，某些粗制滥造的地图软件往往更容易出现性能上的缺陷，比如延迟卡顿现象。尽管在搜索的结果上可能并无太多问题，但是用户体验会很差。想象一下，当用户点击查询之后，如果手机出现了卡顿，这会多么影响用户内心的感受啊。在一定的程度上，这是一个相对更重要的有待研究的因素。

另一方面，目前市场上运行的移动设备系统类型比较少。几年前流行的塞班系统已经几乎完全退出了历史舞台，安卓系统成为了目前移动设备系统的一大巨头。因此，在安卓系统上进行对移动设备应用的性能缺陷的分析就变得格外的重要。由于安卓系统的灵活性非常强，基于安卓系统的性能缺陷研究也就变得更加有意义和有挑战性。

不少人都有过安卓开发的经历。如今的情况下，进行安卓开发也变得越来越容易了。只要在自己的电脑上安装一个安卓的 SDK 和一个文本编辑器，就可以快速上手安卓开发，成为一名入门的安卓开发者了。因此，成为一个安卓开发者的门槛越来越低，这也同时导致了市场上的安卓应用质量更加参差不齐。反过来看，对于开发者来说，尽管开发应用变得简单了，但是开发出高质量的应用依旧是一个不小的挑战。新手开发者由于经验不足，缺少相应的知识，很可能在应用的性能上掌控能力较弱，致使开发出的应用存在大量的性能上的缺陷。最常见的可能就是应用的卡顿。比如开发者不小心把大量的计算任务放在了 UI 线程这种事情，对于一个新人开发者来说还是很容易出现的。再有就是资源上浪费。一个是在内存上，比如开发者使用了不合理的数据结构或者算法，使得应用程序使用了不需要的内存。另一个可能是在能源上，具体来说就是使用了过量的电量，比如应用程序在某些时候进行了无用的计算等等。

因此，对于开发者而言，解决掉上文所说的性能缺陷就是一个至关重要的任务。新人开发者比较容易导致性能缺陷，但是这并不意味着经验充足的开发者不会遇到这样的问题。并且，性能缺陷的解决经常格外复杂。有数据表示对于性能缺陷来讲，解决所需要的时间和代码量都要多于普通的 BUG。而且，调查对象大多数有经验的开发者。

研究意义

对于用户而言，使用存在性能缺陷的应用程序会显著影响使用者的心情，甚至导致其烦躁不安，乃至做出更加激进的行为，比如删除应用等，最终导致各种各样的后果。这种情况确实比较常见。每个用户都希望自己使用的应用流畅节能，使用舒适。

对于开发者而言，一个很好的发现和解决性能缺陷的方法的意义就更大了。如果开发者开发了一款拥有性能缺陷的应用并且投入市场，那么这个应用用户会很不满意并且给这个应用很低的评价。这直接导致潜在用户们不去下载使用这个开发者的应用，而转而投向其他开发者的怀抱，进而导致开发者遭受各种时间上和金钱上的损失。

这时候，开发者会意识到一个没有性能缺陷的应用是多么的重要。可是，即便是开发者自己认识到了自己开发出的应用存在着性能缺陷，比如卡顿或者消耗异常大小的内存或者电量，也很难找到这些性能缺陷的根源。性能缺陷的 Root Cause可能潜藏在某一句不起眼的代码里面，难以辨别。因此开发者可能需要投入大量精力来寻找是什么导致了性能缺陷的发生。

找到性能缺陷的 Root Cause 之后，就要对这个缺陷进行修复。有时候，性能缺陷的原因可能是对某个函数方法的理解就有问题，或者是自己的算法逻辑有不少需要修改的地方。这时，修复这个性能缺陷就会很麻烦。

因此我们可以看到，对智能手机应用性能缺陷分析迫在眉睫。这样的话，寻找一个好的检测智能手机缺陷的方法至关重要。这可以服务于整个智能手机应用市场，使得用户和开发者都能获得最大收益。

背景知识与工具介绍

程序分析

程序分析是指通过一系列的方法，对已有的一段程序进行分析，理解该程序的行为，并且找出办法对其进行优化或者验证其结果正确性。程序分析可以按照分析方法分为两大类：静态分析和动态分析。

静态分析

静态分析是程序分析中的一种常见方法。静态分析不需要讲程序运行起来，而是让分析工具去分析程序的源代码，可执行程序或者是中间代码，从而给出分析结果。静态分析的好处在于不用真正将程序运行起来，比较简便。坏处再有由于没有将程序实际运行起来，导致无法判定程序实际运行路径，从而使得分析结果过于模糊，或者给出大量根本不会出现的结果。

在静态分析中，控制流和数据流是两个需要考虑的重点。

控制流指的是程序的控制走向。通过对控制流进行分析可以得到程序的控制流图。在控制流图中，节点代表一个程序块，边表示程序可能的运行路径。

数据流指的是程序中数据的值在程序中的变化。通过对数据流分析可以得到在程序可能的各种运行过程中数据的值的行为。

通过对控制流和数据流的分析，我们可以得到很多有助于优化程序的信息。

动态分析

动态分析是程序分析中的另一种常见方法。动态分析需要将程序运行起来。程序可以被运行起来一次到多次，然后从中提取有效的运行信息。

在动态分析中，监控是一种经常会被使用的手段。监控是指在程序运行的过程中，监控记录下程序的具体行为，比如内存的使用，时间的开销， CPU 的占用，函数调用关系，栈的情况等等。根据这些信息，开发者或者分析者可以得出有用的结论，进而找到程序不正常行为的原因。

控制流图

控制流图（CFG）是在程序的静态分析的时候经常会用到的一种表示方法。在这个图中，节点表示一段代码基本块，边表示可能的流向。通过静态分析，可以得到一段代码的控制流图。

以如下的代码段为例：

0: a = 1

1: if (a == 1)

2: b = 0

3: goto 5

4: b = 1

5: print b

在这个简单的例子中，尽管只6行代码，但是也要分成4个基本块。其中语句0和语句1构成基本块 A，语句2和语句3构成基本块B ，语句4构成基本块C，语句5构成基本块D。该段代码的 CFG 图大致如下：

图cfg1.png

利用控制流图，我们可以得到很多有用的信息。比如可到达性和支配关系。

可到达性

当我们得到了一段程序的 CFG 之后，我们可以很容易的看出每一条指令的可到达性。当存在一条有向通路从指令m所在的基本块出发，到达指令n所在的基本块的时候，则称指令n是从指令m出发可到达，否则为不可到达。

根据程序代码的可到达性，如果一些代码段，或者基本块，是从初始基本块开始不可到达的话，则可以直接从代码中删除。因为这些代码是永远都不会被执行到的。

举例而言，比如下图的代码以及对应的 CFG：

0: a = 1

1: goto 3

2: b = 1

3: print b

图cfg2.png

我们可以看到从起点开始，语句3是可到达的，语句2是不可到达的。因此语句2可以被删除，因为永远不会被执行到。

支配关系

支配关系是可以从 CFG 中得到的又一大有用信息。

若存在一个基本块A和基本块B，到达B的路径都必须经过A，则我们定义基本块A支配基本块B。如果A是B的直接祖先，则称基本块A直接支配基本块B。

可以看出，起始基本块支配任何其他的基本块。

相反的, 若存在一个基本块A和基本块B，从A到程序出口的路径都必须经过B，则我们定义基本块B后支配基本块A。如果A是B的直接祖先，则称基本块B直接后支配基本块A。

可以看出，出口所在的基本块吼支配任何其他的基本块（程序只有一个出口的情况下）。

举例而言，比如下图的代码及对应的 CFG：

0: a = 1

1: if (a == 1)

2: b = 0

3: goto 5

4: b = 1

5: print b

图cfg3.png

在这幅图中，代码块A支配B，C，D。代码块D后支配A，B，C。

程序依赖图

程序依赖图是一类有向图，表示了程序中变量的一系列依赖关系。在程序依赖图中，一个节点表示一个变量，一条边表示一个依赖关系。

比如有如下的等式关系：

a = b + c

b = 1

c = d

d = 2

这样，在程序依赖图中就会存在四个节点a，b，c，d。并且，

a依赖于b，c

c依赖于d

根据上述信息得到的程序依赖图如下：

图pdg1.png

根据程序依赖图，我们可以得到元素的计算顺序。比如上述例子中，一个合法的计算顺序是d，c，b，a。

另外，如果在计算过程中发现程序依赖图存在环，则说明不存在一个合法的计算顺序，也就是说这些元素无法顺序被依次计算出来。

安卓系统内的 Activity 及其生命周期

在安卓系统中，一个 Activity 是种类型的类，表明了一件用户执行的单一操作，是一个应用程序的组件。几乎所有的 Activity 都会和用户有交互的过程，所以在 Activity 中，有着对图形用户界面内容的管理。一个应用程序通常包含多个 Activity。

一般而言，一个按照的应用程序可能包含以下这些组成元件： Activity，Service，Broadcast Receiver 和 Content Provider。每一种元件都有一个自己的生命周期。

一个完整的安卓 Activity 生命周几可能会涉及到以下这些函数。这些函数都可以被重写以便于实现用户在切换 Activity 状态的时候想实现的内容：

protected void onCreate(Bundle savedInstanceState);

protected void onStart();

protected void onRestart();

protected void onResume();

protected void onPause();

protected void onStop();

protected void onDestroy();

Activity 的四种状态

一个 Activity 会有四种可能的状态：Running，Paused，Stopped，Destroyed。

Running：如果一个 Activity 正在被显示在屏幕上并且具有焦点，那么这个 Activity 处于 Running 态。

Paused：如果一个 Activity 失去了焦点，那么这个 Activity 就处于 Paused 状态。当系统资源严重不足的时候，比如内存紧张，安卓系统可能会杀死一个处于 Paused 状态的应用程序。

Stopped：当一个 Activity 完全被另一个 Activity 掩盖掉，进入后台的时候，这个 Activity 就进入了 Stopped状态。这时候这个 Activity 不再被显示出来，所以一旦内存紧张，就很容易被安卓系统杀死。

Destroyed：如果一个 Activity 因为各种可能的原因被系统杀死，那么这个 Activity 就进入了 Destroyed 状态。如果一个处于 Destroyed 状态的 Activity 想要重新执行，那么它必须被彻底的从一开始执行。

图 activity.png

几个重要的 Activity 的寿命

完整寿命：一个 Activity 的完整寿命是指从 onCreate 到 onDestroy 之间的时间。

可见寿命：一个 Activity 的课件寿命是指从 onStart 到 onStop 之间的时间。

前景寿命：一个 Activity 的前景寿命是指从 onResume 到 onPause 之间的时间。

Android 的单线程模型

在安卓应用中，存在一个重要的线程，叫做主线程，也被称作 UI 线程。这个线程在应用程序开始的时候被建立，用于创建和初始化应用程序所需的部件。

默认情况下，这个主线程承包了所有用户和图形用户界面上面的元件交互的工作。每当一个图形用户界面上面的部件被调用时，回调函数都会在这个主线程中去执行。比如这个例子：当用户按下屏幕上图形用户界面中的一个按钮的时候，主线程会把这个按键时间分发给这个按钮部件，然后处理之后开发者设计好的处理函数。这些操作都会在主线程里面被执行。

因此，如果开发者在主线程里面进行大量复杂操作的话，由于上述单线程模型的存在，这个应用程序的表现效果可能会变得非常糟糕。比如，在主线程里面执行网络访问，或者是数据库查询的话，这些操作会阻塞整个主线程。而当主线程被阻塞之后，接下来的各种事件就无法被继续分发（此时连绘制都无法继续进行）。而对于用户来说，用户会感到界面卡住了，这影响了用户的体验。同时，如果卡顿时间过长，应用程序甚至会导致安卓系统的提示用户该应用停止响应，让用户体验雪上加霜。

很多开发者在刚刚接触安卓应用开发的时候，并没有意识到安卓具有这个单线程模型，在主线程中进行了复杂操作，然后导致了程序卡顿现象的发生。当这些开发者学会了将这些复杂操作写到别的线程之后，他们开发出的应用程序就会变得更加流畅。

Android 单线程模型的优势

安卓系统在图形用户界面上使用单线程模型有不少优点。首先，使用单线程模型之后，所有的图形用户界面操作都在同意而线程里面，这样可以让图形用户界面的代码实现变得简单快速，并且更容易被预测。这是因为图形用户界面的状态不会由于线程的切换而随时任意变化。而且，由于实现了统一的消息队列，开发者们更容易保证应用程序的线程安全性。应用程序的开发者只需要关心不要将复杂操作放到这个主线程里面就可以了。如果需要的话，开发者可以让后台线程发送消息给回主线程。这样，主线程就可以变得流程不卡顿了。

Soot

基本介绍

Soot是一个由 McGill 大学的 Sable 研究小组开发 Java 程序静态分析工具。这个工具异常强大，能够从多个角度来处理 Java 程序，给出全面的分析。

在 Soot 中，有四个基本的类，他们分别是：

Scene：保存了 Soot 分析目标代码时运行的环境以及各种额外的信息

SootClass：表示 Java 代码中的一个类

SootMethod：表示类中一个方法（函数）

SootField：表示一个类的成员域

Body：表示类中一个方法的内容

使用方法

Soot可以有两种使用方法，一种是在命令行中直接使用，另一种是作为一个库在Java程序中调用。

在命令行中使用

在命令行中，Soot可以用以下的命令执行：

java [javaOptions] soot.Main [sootOptions] classname

其中，javaOptions 是一系列的java命令，这点请参照java的使用文档，在此不再赘述。SootOptions 是 Soot 的一系列指令。如果想知道Soot都有什么样的千奇百怪的指令的话，可以运行如下指令来获得帮助：

java soot.Main –h

作为库使用

Soot 可以在Java代码中作为第三方的库引用。首先，将 Soot.jar 添加到 Java 的 Library 路径。然后使用以下代码来调用 Soot 进行分析：

SootClass sootClass = Scene.v().forceResolve(className, 3);

Soot 中对于 Java 类的分类

在 Soot 中，Java的类被分为以下三类：

Argument Class：Argument Class 是作为参数传给Soot。每个 Argument Class 同时都是 Application Class。

Application Class：Application Class 是用户希望 Soot进行分析的类。Soot可以对这些类进行分析处理并且给出分析结果。

Library Class：Library Class 是用户不需要分析，但是需要存在才能是得程序正常运行的类。这些类Soot会进行处理，但是不会输出分析结果。

通过 Soot 得到程序的控制流图

Soot提供了获得程序控制流图的方法，实现在 soot.toolkits.graph包里面。Soot 中的 CFG 基于 Directed Graph，提供了一些基本的操作，比如获得图的大小，获得图的起始点和终止点，获得一个节点的前后继等等一类的操作。

通过 Soot 得到程序的函数调用图

在 Soot 对目标代码进行分析的时候，会生成函数调用图（Call Graph）。这部分的实现代码可以在 soot.jimple.toolkits.callgraph 里面找到。Soot 在执行 Class Hierarchy Analysis 的时候会进行函数调用图的生成。

我们可以用如下代码得到函数的调用关系图：

CHATransformer.v().transform();

SootMethod src = Scene.v().getMainClass().getMethodByName("tempMethod"); CallGraph cg = Scene.v().getCallGraph();

这样，我们就得到了cg作为以Main类中的 tempMethod 函数为根的甘薯调用图。

函数调用图的遍历

我们若想只要一个函数调用了哪些其他函数，需要调用上一步得到的函数调用关系图。Soot 提供了一些方法，可以让我们很轻易的对其进行遍历。比如如下示例代码：

ReachableMethods rm = new ReachableMethods(CG, (Collection)listMe);

rm.update();

QueueReader<MethodOrMethodContext> reader = (rm).listener();

while (reader.hasNext()) {

SootMethod tgtMethod = (SootMethod)(reader).next();

// do something

}

可以看到，我们调用 ReachableMethods 函数就可以轻松得到一个函数调用图的内容并且进行方法的遍历。

我们还可以换一种方法进行前继，请看如下示例代码：

Iterator sources = new Sources(cg.edgesInto(target));

while (sources.hasNext()) {

SootMethod src = (SootMethod)sources.next();

// do something

}

通过 Soot 得到程序的依赖图

使用 Soot 工具可以很方便地获得一段代码的程序依赖图。方法如下：

ExceptionalUnitGraph EUG = new ExceptionalUnitGraph(me.retrieveActiveBody());

HashMutablePDG PDG = new HashMutablePDG(EUG);

依赖图的遍历

通过如下命令就可以对依赖图进行遍历：

Iterator<Object> itNodes = (PDG.getNodes()).iterator();

while (itNodes.hasNext()) {

// do something

}

DDMS

DDMS 简介

DDMS 全称为 Dalvik Debug Monitor Server。Dalvik 是 Google公司开发的可以运行安卓程序的虚拟机，而 DDMS 是一个用于安卓平台的调试工具。在下载安装安卓 SDK 的时候，DDMS 就被集成在里面一起安装了。DDMS 提供了不少有用的功能，比如端口转发，截屏，线程和堆栈监视等等。

DDMS 既可以调试真机，也可以调试模拟器中的安卓程序。比如，一个常用的功能，LogCat。无论是从真机还是模拟器中发出的 Log 信息，LogCat都能将其显示出来，提供给开发者。

DDMS 工作原理

在 DDMS 启动之后，它首先会去连接 ADB。当设备连接之后，在 ADB 和 DDMS 之间会建立一个 VM 监控服务。这个监控服务会告知 DDMS 设备上 VM 的启动情况和终止情况。DDMS 可以得到 VM 的进程 ID，并且建立到 VM 调试器的连接。

DDMS 指定一个端口和 VM 通信。在安卓设备上，每个的应用程序都有一个属于自己的 VM 和一个唯一的端口号用来通信。

DDMS 使用

在Eclipse 中或者 Android Studio 中，我们都可以找到 DDMS 的标示。点击之后可以切换到 DDMS 的界面。DDMS 会判断现在是否连接真机来决定使用真机还是安卓模拟器。

DDMS 也可以通过命令行启动。

查看进程的堆栈信息

DDMS 可以用来查看在设备中运行的进程的堆栈信息。当我们需要确定某个时刻的堆栈使用情况的时候，这个功能就变得十分重要。

跟踪内存分配情况

我们经常会需要确认应用程序使用内存的情况。DDMS 提供了丰富的方法来跟踪类和线程中分配内存给对象的情况。这个功能可以实时的进行，因此这是一个十分重要的功能，可以帮助我们检查出内存分配的问题来提升程序的性能。

文件系统管理

在 DDMS 中，我们可以使用它提供的一个文件系统进行文件的操作，包括查看、复制、粘贴等等。这是一个非常方便的接口。

线程信息检查

DDMS 提供了检查目前在设备中运行了哪些线程的工具。

方法描述工具

这是 DDMS 中的一个很有用的工具。这个工具可以检测一个线程或者方法的调用过程，执行时间。这个工具可以通过两种方式使用。一个方法是在 DDMS 界面中，点击 Start Method Profiling开始监控，之后点击 Stop Method Profiling停止监控。另一个方法是可以在代码中添加 startMethodTracing() 和 stopMethodTracing 来监控目标代码。

TraceView

TraceView 简介

TraceView 是 DDMS 中提供的一个图形化调试工具。这个工具可以打开上文提到的方法描述工具输出的.trace文件，并且给出图形化的输出给用户。TraceView 界面分为两部分，时间线面板和描述面板。

时间线面板

在时间线面板中，每一个线程有一个自己的时间轴。每个方法在这个时间轴中占用这自己的位置。时间从左到右递增。

图traceview1.png

描述面板

在描述面板中，每一行是一个方法。所有在监控时间中被调用的方法都会在这个面板中出现。这个面板显示了每个方法的运行时间，包括 inclusive 和 exclusive 的两种时间以及对应的占总时间的白分比。

同时，每个方法的父方法和子方法也都被显示了出来。父方法是指调用自己的方法，子方法是指自己调用的方法。调用次数也都被记录了下来。

图traceview2.png

Dmtracedump 工具

Dmtracedump 是一个工具，可以将 trace 格式的文件以另一种形式展现出来。具体的表现形式类似于一棵树，每个节点都是一个方法，边表示调用关系。这个工具依赖 Graphviz 的库。

这种方式显示出来结果比较乱，看起来不直接，而且图片庞大。

火焰图工具

火焰图是一种监控信息的可视化表示方法。使用这种可视化的表示方法之后，可以非常清晰得看到占用时间最长的调用路径。这样可以有效地看出应用程序阻塞的瓶颈函数。

火焰图工具输出 .svg 格式的火焰图图像，这是一种可以交互的图像格式，使用起来很方便。

火焰图的表示

图flamegraph1.png

火焰图要从横轴和纵轴两个维度来观察。

横轴：横轴是栈里面的方法的名称，按照字母来排序。横轴总长度为总时间，里面每个小块的长度为对应的这个函数执行的总时间，因此一个小块越长，这个函数出现的时间就越长。

纵轴：纵轴是函数调用栈。纵轴里面的上下关系表示函数的调用和被调用的关系。最上面的小块就正在被执行的函数，下面的小块都是它的祖先函数。

火焰图中的颜色不重要，都是随机产生的，只是为了形成一种火焰的感觉。

火焰图来源

火焰图的输入来源可以是各种监控工具产生的输出文件。比如：

Linux：perf，SystemTap，ktap

Solaris，illumos，FreeBSD：DTrace

Mac OS：DTrace，Instruments

Windows：Xperf

事实上，只要我们有有意义的栈记录，就可以使用火焰图工具生成火焰图。比如上文提到的 DDMS 生成的 trace 文件就是一个很好的输入来源。

火焰图的使用

首先使用监控工具，比如 perf、DTrace 等，生成栈记录文件，然后转变成火焰图工具可以识别的格式。

然后使用以下命令来生成火焰图（需要perl）：

./flamegraph.pl out.kern\_folded > kernel.svg

智能手机性能缺陷介绍

需要先回答的问题

想要实现一个对于智能手机应用的性能缺陷进行静态分析的工具，我们首先要对智能手机性能缺陷有一些基本的认识。我们将从以下几个问题入手，研究智能手机性能缺陷的各种特性：

1. 常见的智能手机应用性能缺陷具有什么样的类型和后果？
2. 这些性能缺陷有什么表现特点？
3. 修复这些性能缺陷的过程有什么特点？

接下来我们对这些问题进行逐一的讨论。

常见的智能手机应用的性能缺陷的类型和后果

大多数的智能手机应用的性能缺陷都涉及到以下三种类型和后果。

1. 图形用户界面的卡顿和延迟

这是常见的性能缺陷现象之中，出现概率很大的一种类型。这种类型的 bug 非常影响应用程序的反应速度的平滑性，从而严重影响用户体验 。

实例：在以前版本的 Firefox 的浏览器，如果用户进行标签页的切换，就有可能造成 Firefox 浏览器用户界面卡顿。这会导致浏览器无法继续正常使用，造成各种后果。

1. 电量的过度使用

由于程序的不当设计或者用户的不当操作，导致只能手机的电量下降速度异常。这也是一种很严重的缺陷，可能使得手机的电量过快地下降，影响用户进行其他操作。

实例：在以前版本的 Zmanim 应用中，这个应用程序可能会去绘制一些不被显示的 GUI 。这种操作很明显地会影响电量的使用，使得电量下降变得更快。

1. 内存泄漏现象

内存泄漏是应用程序经常可能会出现的一种 bug。这种类型的 bug 可能导致大规模的内存爆炸，从而影响应用程序甚至系统的正常运行。

实例：在以前版本 chrome 浏览器中，经常会出现内存泄漏的现象。一旦内存泄漏被多次的触发，就会多次调用系统的自动垃圾回收机制，从而显著影响应用程序的性能。

常见的性能缺陷表现特点

大多数的智能手机应用的性能缺陷具有如下的特点：

1. 促发性能缺陷可能仅仅需要很少量的输入数据

这是和 PC 应用程序的性能缺陷相比很大的一点差别。PC 应用程序的性能缺陷触发可能需要很大量的数据输入，比如几十M，几百M的输入数据。而智能手机性能缺陷可能由于非常简单的一个输入引发，比如触发一次屏幕事件。

1. 促发性能缺陷可能需要特定的用户操作顺序

根据分析，相当一部分的性能缺陷需要用户以特定的序列操作才能引发。这个对于开发者重现bug造成了相当大的困难。开发者经常无法预料到用户的特定操作序列，因此难以进行有效的 debug 操作。

1. 缺乏自动化的衡量策略

我们衡量一个问题是否是一个性能缺陷bug没有一种很通用的自动化衡量标准。我们可能依赖以下几点来判断性能缺陷：人类报告、同类产品比较、开发者的共识等有限的信息来源。因此，在衡量一个bug是否是性能bug的时候，难免会占用很多复杂的人类劳动。

1. 性能缺陷可能依赖于实际使用平台

一个性能缺陷可能在某一个平台上出现，而再另一个平台上不出现。由于智能手机平台的多样性，导致只能手机的性能缺陷的发现与解决存在很大的困难。

性能缺陷修复过程的特点

智能手机的性能缺陷和非性能缺陷的 bug 相比，修复时间更长，用户提供的 bug 报告更多，并且解决 bug 需要的补丁包更大。

总结

综上所述，无论是诊断还是修复一个性能缺陷，尤其是在智能手机的应用里面的性能缺陷，都存在很大的困难。

常见的智能手机应用性能缺陷的模式

我们发现，大部分的只能手机应用性能缺陷，都是属于以下的几种模式。

1. 主线程中的复杂操作

上文提到了安卓系统中的单线程模型。这个模型中提出了一个主线程，在应用程序的执行过程中，默认所有的图形用户界面操作都是在这个线程中完成的。因此，如果应用程序开发者在这个主线程中添加大量的复杂操作，就会导致图形用户界面的卡顿，从而产生性能缺陷。常见的复杂操作可能会涉及数据库查询，网络访问等等。

比如下面的例子。用户在应用程序中按下一个按键，这个按键调用了一个onClick函数，然后这个函数的功能是去查询一个数据库中的表项，然后将结果显示在屏幕上。如果这个应用程序的开发者直接将查询数据库的操作写在onClick函数里，那么由于数据库查询时间可能会很长，这样就会导致程序卡住。当数据库查询之后，结果被显示在屏幕上，程序界面才能够变得正常。

正确的做法应该是，当用户点击onClick之后，创建一个临时的线程用来处理这个数据库访问。这样onClick就能迅速返回，不会继续阻塞主线程。当数据库查询完成之后，想办法让这个线程发消息给主线程，然后将查询结果显示在屏幕上。这样就避免了性能缺陷的发生。

1. 隐身图形用户界面的操作和相关计算

当一个安卓的应用程序被放到后台的时候，它在后台仍然可以继续运行，处理各种事件。然而由于这个时候图形用户界面并没有被显示出来，所以这时候图形用户界面并不需要进行变化的计算。这些更新可以等到这个应用程序重新被放回前台的时候一并计算。

比如下面的例子。一个安卓应用程序在被切换到后台了，然后它仍然继续相应 GPS 的变化，并且执行一些函数。在这些函数中有可能会去对图形用户界面进行一些操作。可是这些操作并不会被显示出来，所以这些操作会浪费额外的资源。

正确的做法应该是，当程序在后台接受到 GPS 的变化时，不应该立即对图形用户界面进行修改，而是应该想办法保留这些变化。当这个应用程序再次被放回到前台的时候，一并计算这些变化，并且实际操作图形用户界面。这样可以节省不少计算资源，同时不会影响程序的表现效果。

1. 经常使用的复杂回调函数

在实际开发中，开发者经常会写出一些病态的、效率低下的回调函数。比如 listView 的回调函数。当开发者自行设计一种 listItem 的时候，需要实现 getView 函数。当一个 item 由于 list 的滚动导致出现在屏幕上的时候，系统会调用这个 getView 函数。在这个函数中需要访问并且生成该 item 的 layout。生成 layout 的过程可能会访问额外的xml文件，从而造成时间上的浪费。因此在 android 开发中，开发者应该使用安卓系统提供的 recycledView 和 view holder 模式来加速 getView 的过程。如果开发者没有这么做，则可能造成卡顿现象的出现。

基于静态分析的性能缺陷分析方法

在本部分中，我们重现了一个名叫 PerfChecker 的安卓性能缺陷分析工具。

PerfChecker 简介

在生活中，人们越来越多地接触智能手机。然而，智能手机的软件大多或多或少地存在一些性能缺陷，也就是performance bug。尽管这样的性能缺陷影响人们对软件的使用，人们在解决这种性能缺陷上的进展一直不是很顺利。我们对性能缺陷的理解很少，并且缺乏有效的解决方法。

PerfChecker是在论文“Characterizing and Detecting Performance Bugs for Smartphone Applications”中提出的一款静态分析工具。它可以用来检测智能手机软件能性能缺陷。在这篇论文中，作者分析了常见的智能手机软件性能缺陷的特征，比如性能缺陷的类型以及它们产生的过程。然后针对每一种特征，这篇论文提出了对应的解决方案。最终，整合实现出来的性能缺陷分析工具就是PerfChecker。

在实际操作上，PerfChecker 输入的是智能手机应用的 .class 文件或者源代码，在分析过程中会使用 java 的静态分析库 Soot 来得到必要的中间过程，然后执行一定的算法来进行上述两种性能缺陷的检测。

PerfChecker 基本原理

PerfChecker 作为一个智能手机应用性能缺陷的静态分析工具，使用了 Soot 工具进行 java 源代码的静态分析。

PerfChecker 可以检测可以检测以下两种性能缺陷的类型：主线程中的复杂操作，view holder模式的使用检测。

在两种缺陷的检测中，首先 PerfChecker 都要先遍历 classPath 路径，找到所有的 class 文件。接下来，PerfChecker 要从这些 class 文件中，找到自己感兴趣代码入口。对于主线程中的复杂操作来说，PerfChecker 要找到所有图形用户界面的事件处理函数入口，以及所有基于 Activity 类的 class 的入口；对于view holder模式的使用检测来说，PerfChecker 要找到所有继承 android.widget.BaseAdapter，android.widget.ListAdapter 或者 android.widget.SpinnerAdapter 这三个类的 class 的入口。

接下来，对于主线程中的复杂操作，和view holder模式的使用检测两种类型的性能缺陷检测就有了不同的算法。对于主线程中的复杂操作，需要找到上一步中得到的类里面，所有由 on 开头的函数。这些函数可能是 Activity 类的 onCreate 函数，或者是图形用户界面的 onClick 函数等等。对于每一个找到的函数，PerfChecker 都要利用 Soot 工具，得到一个从这个函数出发的调用关系图。之后遍历这个调用关系图，观察其中是否包含数据库查询、网络访问一类的复杂操作。如果有则记录下来，之后统一输出。对于view holder模式的使用检测来说，PerfChecker 要找到这些类中的 android.view.View getView函数，然后得到这个函数的程序依赖图。得到这个程序依赖图之后，PerfChecker 要遍历这个程序依赖图的每个节点，观察代码是否利用的安卓系统提供的 recycledView。如果没有利用的话，就记录下来，之后统一输出。

主线程中的复杂操作的检测

1. 遍历 classPath 路径，找到所有的 class 文件

首先我们得到所有的 class 文件，这些文件里面包含我们需要需要分析的 class。这里我们实现了一个类叫做 ClassFinder，其中的 find 函数用来遍历文件路径，并且返回所有寻找到的 class文件：

public static void find(File fileClassPath, String prefix, ArrayList<String> classesFounded) {

if (fileClassPath.exists()) {

File[] Files;

Files = fileClassPath.listFiles();

for (int i = 0; i < Files.length; i++) {

fileClassPath = Files[i];

if (fileClassPath.isDirectory() && (!fileClassPath.getName().contains("."))) {

if (prefix == null)

find(fileClassPath, fileClassPath.getName(), classesFounded);

else

find(fileClassPath, prefix + "." + fileClassPath.getName(), classesFounded);

} else if (fileClassPath.getName().endsWith(".class"))

if (prefix != null)

classesFounded.add(prefix + '.' + fileClassPath.getName().substring(0, fileClassPath.getName().length() - 6));

else

classesFounded.add(fileClassPath.getName().substring(0, fileClassPath.getName().length() - 6));

}

}

}

可以看到，find 函数的工作流程如下：

参数File fileClassPath 是目前待操作的文件路径。String prefix 是目前记录下来的一个前缀，初始为空，每次进入一个文件夹的时候都会添加当前文件夹的名称和一个”.”，使得之后记录下来的 class 文件路径可读可用。ArrayList<String> classesFound 是一个列表，每当找到一个 class 文件，都要添加到这个列表里。

接下来遍历当前路径中的每一项。

如果该项是一个文件目录，并且目录名里面不包含”.”，那么判断当前状态下的 prefix 是否为空。递归调用 find 函数进入下一层文件夹。

如果该项是一个文件的话，则判断当前状态下的 prefix 是否为空，然后将自己这个 class 文件去掉后缀名后，拼接上 prefix 然后记录到 classesFound 列表里。

1. 找到处理图形用户界面事件的类以及继承 Activity 的类。

接下来，PerfChecker 遍历上一步找到的所有 class 文件，一次对其分析，是否是处理图形用户界面事件的类以及继承 Activity 的类。

Class cl = Class.forName(className);

boolean toAnalyse = false;

if (checkGUI != 0 && className.contains("$")) {

for (Class clInterface : cl.getInterfaces()) {

if (ListenerList.list.contains(clInterface.getName())) {

toAnalyse = true;

}

}

} else if (checkLifecycle != 0) {

Class tmp = cl;

while (tmp.getSuperclass() != null) {

if (tmp.getName().contains("android.app.Activity")) {

toAnalyse = true;

}

tmp = tmp.getSuperclass();

}

}

if (toAnalyse) {

toAnalyseList.add(className);

}

1. 判断一个类是否是处理 GUI 事件回调函数的类

对于一个找到的类，首先判断这个类的文件名，所有包含GUI事件回调函数的类，都会被编译成XXXXX$YYYYY.class的形式。因此我们找出所有这种形式的类，观察弃是否真的包含GUI回调函数。为此，我们事先准备好了一系列GUI回调函数的列表（见附录ListenerList）。得到并且遍历改类的所有接口，如果存在上面提到的列表里面的函数，则认为这个类需要分析。

1. 判断一个类是否是继承 Activity 的类

对于每一个类，迭代判断他的父类。迭代的终止条件就是该类的 getSuperClass 返回 null。如果这个类的某一级父类包含 android.app.Activity 字段，则认为这个类是继承 Activity 的类，因此需要分析。

对于所有标记为需要分析的类，PerfChecker 将其统一保存在一个列表里面。

1. 找到该类的入口

上一步中，我们得到了所有需要分析的类。接下来，我们需要找到这些类可能的入口。

List<SootMethod> methods = sootClass.getMethods();

ArrayList<SootMethod> entrypoints = new ArrayList<SootMethod>();

Iterator<SootMethod> itMethods = methods.iterator();

while (itMethods.hasNext()) {

SootMethod method = itMethods.next();

if (method.getName().startsWith("on")) {

entrypoints.add(method);

}

}

我们假定，对于一个函数，如果它以on作为前缀，则他很可能是一个入口方法。

1. 得到函数调用关系图和所有可能到达的目标函数。

CallGraph CG = Scene.v().getCallGraph();

ArrayList<SootMethod> listMe = new ArrayList<SootMethod>();

listMe.add(method);

ReachableMethods rm = new ReachableMethods(CG, (Collection)listMe);

rm.update();

通过调用 Soot 工具，我们可以轻松得获得从某一个函数出发的函数调用关系图。然后 Soot 工具还提供一个函数叫做 ReachableMethods，可以直接得到从这个函数出发的所有可以到达的函数的集合。

1. 判断目标函数集合中是否存在复杂操作

QueueReader<MethodOrMethodContext> reader = (rm).listener();

while (reader.hasNext()) {

SootMethod tgtMethod = (SootMethod)(reader).next();

if (LengthyOprList.list.contains(tgtMethod.getSignature())) {

result.addResult(sootClass, method, "MAY CALL " + tgtMethod.getSubSignature().substring(0, tgtMethod.getSubSignature().indexOf('(')));

}

}

我们事先存好一系列复杂操作的函数签名，比如数据库访问、网络通信等等（见附录LengthyOprList）。然后遍历上面得到的目标函数是否属于这个List。若属于，则记录到result中。

view holder模式的使用检测

1. 遍历 classPath 路径，找到所有的 class 文件

这部分和主线程中的复杂操作的检测中基本一样，不再赘述

1. 扫描寻找需要分析的类

Class cl = Class.forName(className);

boolean toAnalyse = false;

Class tmp = cl;

while (tmp.getSuperclass() != null) {

if (tmp.getName().contains("android.widget.BaseAdapter")

|| tmp.getName().contains("android.widget.ListAdapter")

|| tmp.getName().contains("android.widget.SpinnerAdapter")

) {

toAnalyse = true;

}

tmp = tmp.getSuperclass();

}

我们的目的是找到所有开发者编写的getView函数，因此我们去找需要实现这个函数的类的特征。我们发现，我们所有需要实现这个函数的类都是android.widget.BaseAdapter、android.widget.ListAdapter、android.widget.SpinnerAdapter 之一的子类。因此我们类似在主线程中的复杂操作的检测中寻找基于 Activity 的类的方法，迭代寻找被测试类的父类，判断其是不是这三个类之一。如果是，则认为该类需要分析。

1. 找到该类的 getView 函数，并且得到程序依赖图 (Program Dependency Graph) 的对应部分

me = cl.getMethod("android.view.View getView(int,android.view.View,android.view.ViewGroup)");

ExceptionalUnitGraph EUG = new ExceptionalUnitGraph(me.retrieveActiveBody());

HashMutablePDG PDG = new HashMutablePDG(EUG);

这里，在PerfChecker中可以使用 Soot 提供的 API 来找到一个函数在程序依赖图中对应的部分

1. 遍历分析该程序依赖图，判断是否使用了 recycledview 进行加速

Iterator<Object> itNodes = (PDG.getNodes()).iterator();

boolean checked = false;

while (itNodes.hasNext()) {

Object o = itNodes.next();

if (!(o instanceof PDGNode)) continue;

PDGNode node = (PDGNode)o;

if (node.getType() == PDGNode.Type.CFGNODE) {

if (node.toString().contains("r1 == null") || node.toString().contains("r1 != null")) {

checked = true;

}

}

}

if (!checked) {

result.addResult(cl, me, "Recycled View Not Used");

}

在 Soot 工具产生的程序依赖图中，节点分为两种。一种叫做 CFGNode ，也就是控制程序流的节点，比如 if 判断。另一种叫做 Region ，这个代表其他的程序语句块。

因此，为了判断开发者是否依据 recycledview 是否存在来编写 getView 函数，我们需要先找到 PDG 中所有的 CFGNode 。由于 recycledview 是函数的第二个参数，因此如果开发者判断了 recycledview 是否存在，则一定会出现 "r1 == null" 或者 "r1 != null" 的字样。我们根据这个来判断开发者是否使用了 viewholder 模式。

如若没有出现这样的字样，那么可以认为这里出现了性能缺陷，需要记录到 Result 类里面。

Result 类的实现

Result类是我设计的用来保存性能缺陷记录并且格式化输出的一个类。在Result 类里面有一个重要的列表叫做 list。在 list 中保存了每一个性能缺陷的记录。每条记录以 Entry 类的格式记录。

Entry 类的格式如下：

class Entry {

public SootClass cl;

public SootMethod me;

public String comment;

}

可以看出，Entry 类里面，保存了该性能缺陷发生所在的类 SootClass cl，所在的方法 SootMethod me，以及该性能缺陷的描述 comment。

同时，在 Entry 类里面，我还定义了一个相等比较函数 equals，判定两个 entry 相等，当他们的cl、me、comment 均相等。

在 Result 类里面，有一个函数叫做 addResult，用来将一条性能缺陷记录添加到列表中。首先，我先将这条记录转变为 Entry，然后判断当前列表里面是否已经包含了这条记录。如果不包含，则添加该记录到列表。这是为了防止相同位置发生的性能缺陷记录重复出现，影响阅读。

在输出性能缺陷记录的时候，用到了 printResult 函数。该函数会将当前列表中的记录输出为以下格式：

Printing Results

SootClass1

SootMethod11

Comment111

Comment112

SootMethod12

Comment121

SootClass2

SootMethod21

Comment211