清 华 大 学

综 合 论 文 训 练

题目：GUI 程序卡顿现象研究

系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：赵 涵

指导教师：陈 渝　副教授

2016 年 6 月 3 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签　名：_____导师签名：_____日　期：_____

中文摘要

GUI 程序的卡顿现象一直是现实生活中经常遇到的一个问题，而这个问题在智能手机应用上尤为明显。这些现象的出现使得无论是对用户还是开发者，都要承受不小的损失。

因此，本文以安卓手机应用性能缺陷为出发点，分析研究各种类型的性能缺陷的特点。同时，本文从静态分析和动态分析两个维度提出一定解决方案，并且加以测试分析。

关键词：性能缺陷，静态分析，动态分析

ABSTRACT

It is a common problem that software with GUI often encounters hangs and delays, and on smartphones, the case is more serious. As a result, both users and developers have to suffer from this circumstance.

In this article, by starting from the performance bugs in Android applications, we analyze and study the characteristics of various kinds of performance bugs. Meanwhile, we propose solutions by using static analysis and dynamic analysis, which are carefully tested later in this article.

Keywords: Performance bug, Static analysis, Dynamic analysis

目 录

第 1 章　研究背景与研究意义

1.1　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。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，解决掉上文所说的智能手机应用的性能缺陷就成了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。

1.2 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。

1.3 研究现状

近十年来，人们越发意识到性能缺陷问题的严重性，因此也在很多方面对其进行了探索。以下列出一些成果：

How Developers Detect and Fix Performance Bottlenecks in Android Apps：作者在这篇文章中用调查的方法对如何检测和修复安卓应用的性能缺陷进行了研究。在这篇文章中，作者对将近 500 个安卓开发者进行了调查，并且分析了在 github 上面的有关性能缺陷的 issue。这篇文章最后认为，开发者十分依赖用户的反馈和评价才能有效地发现自己的性能缺陷。同时，开发者使用的工具也不能自动检测和修复这些性能缺陷。

A Study on the Performance of Android Platform：作者对开发者常用的一些评测安卓性能的工具进行了分析和讨论。作者对安卓 SDK 中的 DDMS 进行了分析，尤其是对其中的 TraceView 进行了讨论。作者详细介绍了 TraceView 的使用和特性。作者对其给了较高的评价。

Understanding and Detecting Real-World Performance Bugs：作者研究了很多现实中的性能缺陷的例子，并且给出了一定的解决方案。作者找到了 109 个实际发生的性能缺陷，包括 Apache，Chrome，GCC 等等知名的应用。最后作者给出了一种叫做 Rule-Checking 的方法用于检测性能缺陷。

Characterizing and Detecting Performance Bugs for Smartphone Applications：这篇文章中，作者对安卓系统的一些应用性能缺陷进行了分析，然后设计出了 PerfChecker 工具，能对其中的主线程中复杂操作和 ViewHolder 模式进行检测。

Hang Analysis: Fighting Responsiveness Bugs：作者主要是针对卡顿和延迟现象进行了分析。作者设计出了 HANGWIZ 工具，使用静态分析的方法对源代码中的卡顿现象进行检测。同时，作者进行了大量的真实世界中的软件应用进行了测试，证实了自己算法的正确性和有效性。

此外，还有一些其他的针对安卓系统的性能监测工具值得介绍：

Intel GPA System Analyzer – 这是一款 Intel 推出的性能监测工具。这个工具

可以连接到开发者正在开发的应用，监测多种数据类型，比如帧速率和 CPU 使用情况。比如，可以通过观察 CPU 的负载率来理解帧速率的变化情况。

Android Debug Bridge (ADB) – 这是 Android SDK 里面的一个工具，用这个工具可以管理并且操作真机或者 Android 设备模拟器。

1.4 本文结构

本文结构清晰易懂。

第一章描述了研究背景和意义，以及研究现状。第二章对本文依赖的一些基本知识和工具进行了介绍。第三章，本文对智能手机应用的性能缺陷进行了分析。第四章和第五章分别用静态分析和动态分析的放来给出了一定的解决方案，并且进行了测试。第六章进行了实验总结和展望。

第 2 章　背景知识与工具介绍

2.1　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。

2.1.1　静态分析

静态分析是程序分析中的一种常见方法。静态分析不需要将程序运行起来，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。坏处在于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。

2.1.2　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。

2.1.3　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：

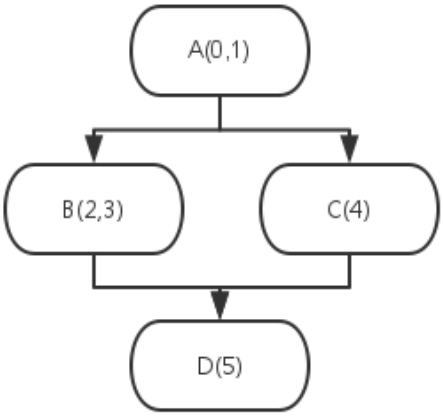


Figure 1 - CFG1

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 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
```

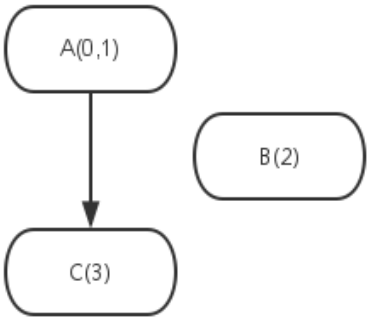


Figure 2 - CFG2

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，出口所在的基本块后支配任何其他的基本块（在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

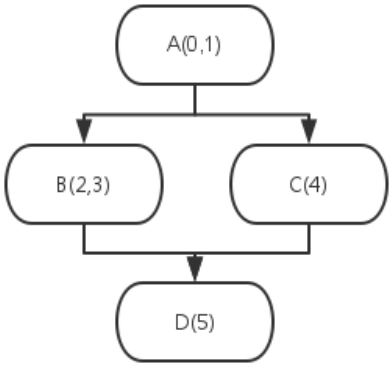


Figure 3 - CFG3

在这幅图中，代码块 A 支配 B，C，D。代码块 D 后支配 A，B，C。

2.1.4 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：

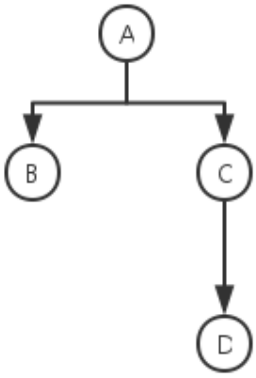


Figure 4 - PDG1

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。

2.2 Soot 工具

2.2.1 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

2.2.2 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，但是需要存在才能使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。

2.2.3 Soot 的一些功能简介

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 函数为根的函数调用图。

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。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计算前继，可以用 cg.edgesInto(target) 来获得。

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。假定一个方法为 me，我们可以先得到这个方法的 ExceptionalUnitGraph，然后再将其转换为 HashMutablePDG 的形式

2.3 DDMS 工具

2.3.1 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。

2.3.2 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。

2.3.3 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内的方法的调用过程，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。这个工具在后文的基于动态分析的性能缺陷分析方法中会被用到。

2.3.4 TraceView 工具

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着自己的位置。时间从左到右递增。

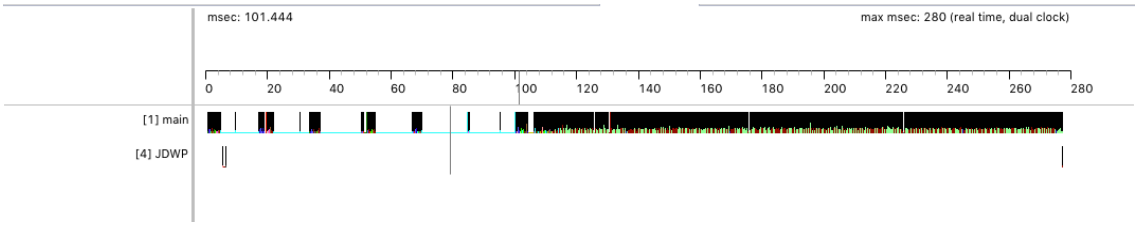
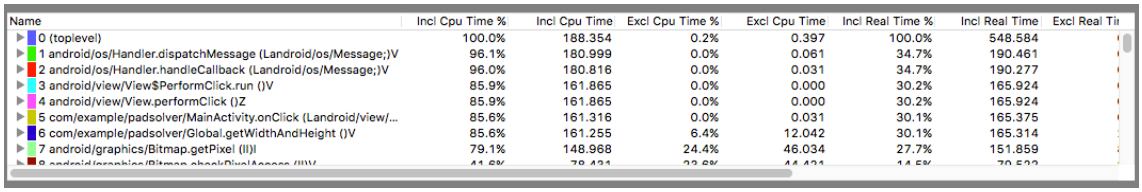


Figure 5 - TraceView1

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。



Name	Incl Cpu Time %	Incl Cpu Time	Excl Cpu Time %	Excl Cpu Time	Incl Real Time %	Incl Real Time	Excl Real Ti
0 (toplevel)	100.0%	188.354	0.2%	0.397	100.0%	548.584	
1 android/os/Handler.dispatchMessage (Landroid/os/Message;)V	96.1%	180.999	0.0%	0.061	34.7%	190.461	
2 android/os/Handler.handleCallback (Landroid/os/Message;)V	96.0%	180.816	0.0%	0.031	34.7%	190.277	
3 android/view/View$PerformClick.run ()V	85.9%	161.865	0.0%	0.000	30.2%	165.924	
4 android/view/View.performClick ()Z	85.9%	161.865	0.0%	0.000	30.2%	165.924	
5 com/example/padsolver/MainActivity.onClick (Landroid/view/...	85.6%	161.316	0.0%	0.031	30.1%	165.375	
6 com/example/padsolver/Global.getWidthAndHeight ()V	85.6%	161.255	6.4%	12.042	30.1%	165.314	
7 android/graphics/Bitmap.getPixel (II)I	79.1%	148.968	24.4%	46.034	27.7%	151.859	

Figure 6 - TraceView6

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 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，因此不推荐。

第 3 章　智能手机应用性能缺陷

3.1　安卓系统内的 Activity 类及其生命周期

3.1.1　Activity 简介

在安卓系统中，一个 Activity 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();
```

3.1.2　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。

具体的转换关系如下图所示：

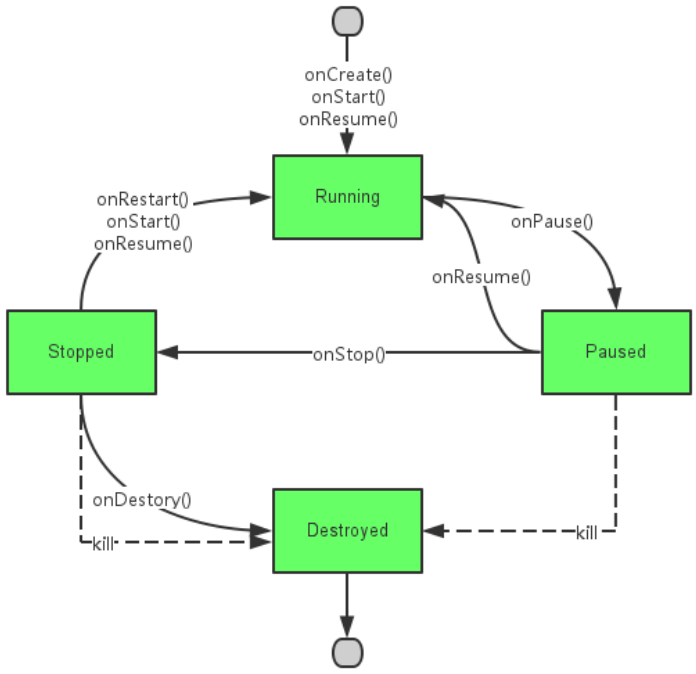


Figure 7 - Activity

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 的可见寿命是指从 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。

3.1.3 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。

3.2 有关性能缺陷的若干问题讨论

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：

常见的智能手机应用性能缺陷具有什么样的类型和后果？

这些性能缺陷有什么表现特点？

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。

3.2.1 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。

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。

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。

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。

3.2.2 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：

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。

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。

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。

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。

3.2.3 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。

3.2.4 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。

3.3 常见的智能手机应用性能缺陷的模式

我们发现，大部分的智能手机应用性能缺陷，都是属于以下的几种模式。

3.3.1 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。

3.3.2 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，然后它仍然继续响应 GPS 位置返回的变化，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。

3.3.3 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。

第 4 章　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。

4.1 简介

在生活中，人们越来越多地接触智能手机。智能手机中有各种各样的应用程序，可是智能手机的软件大多或多或少地存在一些性能缺陷。尽管这样的性能缺陷经常影响人们对软件的使用，然而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 是在论文一个智能手机应用性能缺陷静态分析工具。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性能缺陷的检测。

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 模式的使用问题。

4.2 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。

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。

4.3 主线程中的复杂操作的检测

4.3.1 ClassFinder 的设计

首先，在对主线程中复杂操作进行检测之前，很基础的一步是要先找到所有的 class 文件。因此，我们设计了一个类叫做 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。

我们设计 find 函数拥有三个参数，fileClassPath，prefix 和 classesFound。

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。

4.3.2 设法找到需要分析的类以及对应的入口

在上一步中，我们利用设计出的 ClassFinder 函数已经将目录下面所有的 class 文件都找到了，接下来要想办法找到所有真正需要分析的类。

基本上，我们需要找到能让 Soot 进行分析的入口，因此，我们要找到处理图

形用户界面事件的类以及继承 Activity 的类。对于这两个问题，我们设计了两种不同的策略：

寻找处理 GUI 事件回调函数的类

首先，遍历每一个上文找到的 class 文件。接下来判断这个类的文件名。由于所有包含 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，观察其是否真的包含 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。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。

一些常见的 GUI 回调函数是：android.view.View$OnClickListener，android.view.View$OnKeyListener 等等。我们整理出来了一个列表，里面包含了几十种常见的这类函数名称。

寻找继承 Activity 的类

首先，遍历每一个上文找到的 class 文件，接下来迭代判断他的父类。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，我们将其统一保存在一个列表里面。

找到该类的入口

根据经验，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。比如，onCreate，onPaused 等等。可以事先准备一个列表，保存这一类方法名。如果遍历上文找到需要分析的类中，某个函数名是上面的格式，那么可以认为这是一个潜在的函数入口。

4.3.2 判断从该入口出发可能到达的复杂操作

如果一个函数入口出发，通过某种路径，能够到达一个复杂操作，那么这个地方就会有性能缺陷的隐患。

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。

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。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。一些常见的复杂操作签名有：

java.net.URL, <java.net.URL: java.lang.Object getContent()>，

android.database.sqlite.SQLiteDatabase, <android.database.sqlite.SQLiteDatabase: void execSQL(java.lang.String)>等等。

4.3.3 整体流程总结

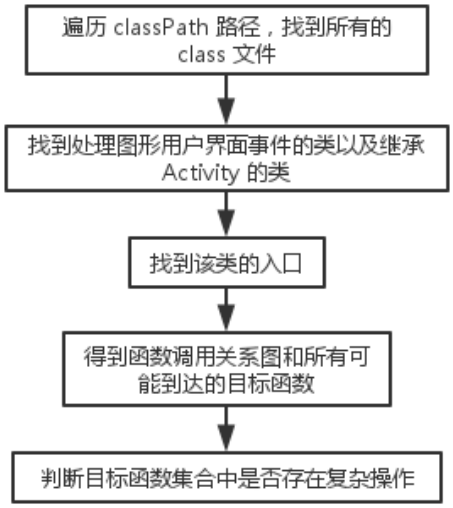


Figure 8 - Static1

4.4 view holder 模式的使用检测

4.4.1 设法找到需要分析的类以及对应的待分析函数

由于这次我们要判断开发者是都使用了 ViewHolder 模式，所以我们经过分析，明确了我们的目的是找到所有开发者编写的 getView 函数。经过研究，我们确认了需要所有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基于如下三个类中的某一个类：

android.widget.BaseAdapter

android.widget.ListAdapter

android.widget.SpinnerAdapter

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。

找到了这些类之后，对应的待分析函数也就很明确，就是其中的 getView 函数。

4.4.2 获得函数的程序依赖图并进行分析

在找到该类的 getView 函数之后，我们为了得到这个函数的程序依赖图，首先要得到这个函数的 ActiveBody。然后使用 Soot 提供的 ExceptionalUnitGraph 作为中间结果，最终得到对应的 HashMutablePDG，也就是这段代码的程序依赖图。

接下来设计算法来判断这一个 getView 函数有没有使用 ViewHolder 模式。一个基本的想法就是去判断这段函数有没有使用 recycledview 进行加速。

已知，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。

4.4.3 整体流程总结

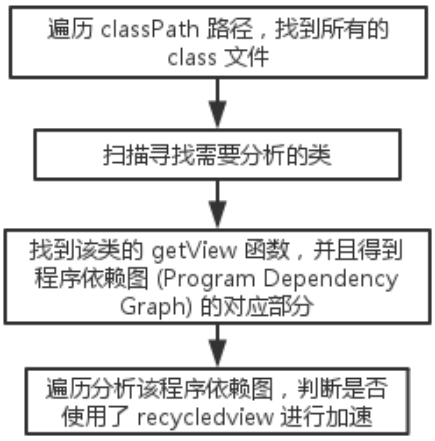


Figure 9 - Static2

4.5 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，设计一个函数叫做 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
```

4.6 测试结果

4.6.1 测试对象和测试内容

我们使用的被测试的应用程序为 Omnidroid。这是一个 NYU 开发的一个 android 的事件管理工具。该应用程序不大，并且依赖的 android10.jar 也不大，因此很适合作为测试工具。

我们从三个方面测试该应用的 Performance Bug：Activity 生命周期中的复杂操作、GUI 事件回调函数中的复杂操作、viewholder 模式使用检测。

4.6.2 运行结果

Activity 生命周期中的复杂操作检测

测试结果发现在 6 个类的中可能会调用复杂操作。这些复杂操作均为数据库访问。

```
Printing Results
edu.nyu.cs.omnidroid.app.view.simple.ActivityChooseFiltersAndActions
        void onActivityResult
```

MAY CALL android.database.Cursor query
boolean onContextItemSelected
MAY CALL android.database.Cursor query
edu.nyu.cs.omnidroid.app.view.simple.ActivityDlgApplicationLoginInput
void onCreate
MAY CALL android.database.Cursor query
edu.nyu.cs.omnidroid.app.view.simple.ActivityDlgLog
void onCreate
MAY CALL android.database.Cursor query
MAY CALL android.database.Cursor query
edu.nyu.cs.omnidroid.app.view.simple.ActivityLogTabs
void onCreate
MAY CALL android.database.Cursor query
void onResume
MAY CALL android.database.Cursor query
boolean onOptionsItemSelected
MAY CALL android.database.Cursor query
edu.nyu.cs.omnidroid.app.view.simple.ActivityMain
void onCreate
MAY CALL android.database.Cursor query
void onResume
MAY CALL android.database.Cursor query
edu.nyu.cs.omnidroid.app.view.simple.ActivitySavedRules
void onCreate
MAY CALL android.database.Cursor query
void onActivityResult
MAY CALL android.database.Cursor query
boolean onContextItemSelected
MAY CALL android.database.Cursor query
MAY CALL android.database.Cursor query
boolean onOptionsItemSelected
MAY CALL android.database.Cursor query

GUI 事件回调函数中的复杂操作

检测测试结果发现在 2 个类中可能会调用复杂操作。这些复杂操作同样均为数据库访问。

```
Printing Results
edu.nyu.cs.omnidroid.app.view.simple.ActivitySavedRules$1
        void onItemClick
                MAY CALL android.database.Cursor query
edu.nyu.cs.omnidroid.app.view.simple.ActivitySavedRules$RuleListAdapter$1
        void onClick
                MAY CALL android.database.Cursor query
```

ViewHolder 模式使用检测

测试结果发现有 9 个 getView 函数没有利用给定的 recycledView，也就是说没有利用 ViewHolder 模式。

```
Printing Results
edu.nyu.cs.omnidroid.app.view.simple.ActivityChooseRootEvent$AdapterEvents
        android.view.View getView
                Recycled View Not Used
edu.nyu.cs.omnidroid.app.view.simple.ActivityDlgActionInput$DlgAttributes$Adapter
Attributes
        android.view.View getView
                Recycled View Not Used
edu.nyu.cs.omnidroid.app.view.simple.ActivityDlgActions$AdapterActions
        android.view.View getView
                Recycled View Not Used
edu.nyu.cs.omnidroid.app.view.simple.ActivityDlgApplications$AdapterApplications
        android.view.View getView
```

Recycled View Not Used
edu.nyu.cs.omnidroid.app.view.simple.ActivityDlgAttributes$AdapterAttributes
android.view.View getView
Recycled View Not Used
edu.nyu.cs.omnidroid.app.view.simple.ActivityDlgFilters$AdapterFilters
android.view.View getView
Recycled View Not Used
edu.nyu.cs.omnidroid.app.view.simple.ActivityLogTabs$LogAdapter
android.view.View getView
Recycled View Not Used
edu.nyu.cs.omnidroid.app.view.simple.ActivitySavedRules$RuleListAdapter
android.view.View getView
Recycled View Not Used
edu.nyu.cs.omnidroid.app.view.simple.AdapterRule
android.view.View getView
Recycled View Not Used

4.6.3 结果分析

可以看出，PerfChecker 确实能像论文中声称的，找出特定类型的潜在性能缺陷。这些分析结果对于程序员调试性能缺陷 bug 肯定会有一定的帮助。

然而，PerfChecker 有如下的缺点：

1. PerfChecker 的原理是基于静态分析，因此就会存在普遍的静态分析可能导致的各种问题。静态分析仅仅是从代码，或者是从中间代码以及二进制码的层面上来分析程序，没有实际将程序运行起来，因此就会存在相当的局限性。比如，程序真正运行起来的时候所产生了实际的控制流和数据流，静态分析是无法获得的。例如，应用程序出现性能缺陷那一次执行中，程序代码运行的通路是无法从静态分析中获得的。然而我们并没有去除掉那些应用程序正常运行时程序代码经过的通路。

2. PerfChecker 给出的结果，可能确实是在动态分析中发生了的现象，可是仅仅是得到了这些发生的现象，我们也无法判断出这是否真正造成了性能缺陷。比如，在主线程中，我们对数据库进行了非常小规模访问操作，可是这个访问的代价可能会很小，因此这个对性能并不会有什么影响，然而 PerfChecker 还是会给出警

告来提示开发者注意。

3. PerfChecker 给出的结果并不能直接转变为解决方法。比如我们可能的确存在在主线程中访问数据库的需求。即使我们将访问数据库这个过程封装在一个新的线程里面，主线程也可能要等待数据库操作执行完毕之后才能继续下一步的工作。这时候，主线程依然不能先执行之后的操作。因此这种时候的性能缺陷有时会很难避免。

第 5 章　基于动态分析的性能缺陷分析方法

由于智能手机应用性能缺陷的静态分析方法具有不少的局限性，我们试图寻找一些能解决这个问题的动态分析方法。

在这里，我设计并且实现了一种基于火焰图的智能手机应用性能缺陷的动态分析方法。

5.1　火焰图及火焰图工具简介

5.1.1　火焰图及其特点

静态分析过程由于不具备应用程序实际运行的信息，因此失去了很多能够分析的内容。这些方面，在动态分析的过程中都能得到补偿。当程序运行起来之后，由于接受了用户的输入，程序实际运行的轨迹才能得以确定。这样，很多没有实际运行到的分支都可以不用被重视，而只需要关心实际发生的情况就可以了。这是动态分析的一些优点。

比如说，只有在应用程序实际运行起来之后，应用程序的堆栈信息才能够被捕获。而一个应用的堆栈信息往往能从很多角度反映一个程序是否在正常运行。函数的调用问题，时间的开销问题等等，都会在动态分析的过程中暴露出来。于是，在这一个章节中，我们就从函数调用栈的实际运行记录出发，来进行智能手机应用性能缺陷的静态分析。

在前面的章节里面，我们引入了一种数据的表现形式，叫做火焰图。火焰图可以清晰的看出应用程序实际运行的瓶颈所在之处，具体而言就是所消耗时间最长的地方。举个例子来说，一个应用程序从 main 函数出发，依次执行了 A，B，C 三个函数，然后在 A，B，C 三个函数里面还存在各种复杂的调用，调用深浅也不一。这时候开发者发现在程序执行 main 的时候发生了卡顿。这时候如果使用火焰图的形式来展示程序的运行过程，会很明显地看出究竟是哪个函数占用了过多的时间。比如可能得到了下面这样的火焰图。

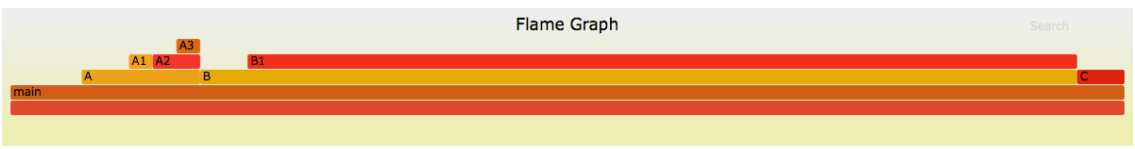


Figure 10 - Dynamic1

看到这个图之后，开发者可以很明确地意识到是函数 B 出了问题，更有可能是函数 B1 内部发生了什么不好的事情。这时候，开发者就可以仔细看看函数 B1 里面是不是有什么地方写错了，或者有什么算法需要改进，或者先尝试直接把函数 B1 删除观察变化。

可以看到，火焰图是观察函数调用栈的一种非常直观的形式，如果有哪个函数占用了异常多的时间，就会在火焰图上异常显眼。

要注意的是，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，都是从暖色系里面随机产生的，只是为了形成一种火焰的感觉。

5.1.2 火焰图工具

火焰图工具是一个开源的工具，该工具可以将一定格式的输入文件转换为 svg 格式的火焰图图像的输出文件。svg 格式一种可以交互的图像格式，使用起来很方便。

火焰图工具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

然而事实上，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。

5.2 整体操作流程简介

开发者可能从多种消息来源得知自己编写的程序的确拥有一些性能缺陷，比如自己的测试结果，再比如用户提供的 Bug 报告。这时候，开发者可以采用下面的方法来进行动态分析。

1. 开发者要记录下导致性能缺陷的操作流程。这是由于前文提到的，智能手机应用的性能缺陷非常依赖用户的操作顺序，否则很可能无法触发性能上的缺陷。

2. 开发者要在自己的开发环境上（很可能是 Android Studio 或者 Eclipse）调试自己的程序，使用上文提到的 DDMS 工具，点击 Start Method Profiling，重现可能引发性能缺陷的操作，点击 Stop Method Profiling。这时候，DDMS 会进入 TraceView 界面。

3. 开发者可以导出 trace 记录到磁盘上。这一份 trace 记录就是我们算法所需的输入文件。

4. 调用 parse.py，可以将 trace 格式的文件转换为一种中间文件格式，暂时叫做 result.txt。

5. 使用 FlameGraph 库，将上面的 result.txt 渲染成 svg 格式的火焰图文件。这个文件可以用各种浏览器打开查看。

6. 通过观察火焰图，开发者能够直观得看出是哪个函数占用了过多的时间。

5.3 两种文件格式

5.3.1 Trace 文件的格式

Trace 文件由两个不同的文件拼接而成：第一部分是文本格式的 Key 文件，第二部分是二进制格式的 Data 文件

Key 文件

第一部分是文本格式的 Key 文件，描述了应用程序运行环境的基本信息、线程信息和函数信息：

```
*version      //环境信息
3
data-file-overflow=false
clock=dual
…
```

*threads// 线程信息

1 main

10 Binder_2

9 Binder_1

…

*methods // 函数信息

0x6d1df560 android/ddm/DdmHandleProfiling handleChunk

(Lorg/apache/harmony/dalvik/ddmc/Chunk;)Lorg/apache/harmony/dalvik/ddmc/Chunk; DdmHandleProfiling.java 73

0x6d1df3d0 android/ddm/DdmHandleProfiling handleMPRQ

(Lorg/apache/harmony/dalvik/ddmc/Chunk;)Lorg/apache/harmony/dalvik/ddmc/Chunk; DdmHandleProfiling.java -1

…

*end

在这里我们比较关心 Thread 部分和 Method 部分。Thread 部分的每一行描述了一个线程，由两部分组成：Thread ID 和 Thread Name。Method 部分的每一行描述了一个函数，由四部分组成：Method ID，Class Name，Method Name，Signature

Data 文件

第二部分是二级制格式的 Data 文件，包含很多条记录。每条记录描述了一次进入函数或者退出函数的事件。对于 64 位系统来说，产生的每一条记录有 14 字节。其中：

0 – 1 字节是一个 Byte。其内容是 Thread ID。

2 – 5 字节是一个 Int。其内容高 30 位是 Method ID，低 2 位如果是 0 则表示进入函数，是 1 则表示退出函数。

10 – 14 字节是一个 Int。其内容是当前事件发生的时刻，单位为 0.001 毫秒。

5.3.2 FlameGraph 工具的输入文件格式

如果要使用 FlameGraph 工具，必须给出满足一定格式的输入文件。对于刚刚那个示例火焰图，对应的输入文件是这样的：

```
main    3
main;A 2
main;A;A1  1
main;A;A2  1
main;A;A2;A3   1
main;B 2
main;B;B1  35
main;C 2
```

每一行包含两个部分：函数调用栈的状态 和 所持续的时间。

函数调用栈的状态又一系列函数名组成，左边的是外层函数，右边的是内层函数。也就是说，最左边的是栈底函数，最右边的是栈顶函数。函数名之间用分号隔开。所持续的时间指的是 exclusive time，下文会详细讲解。

上面的例子里面可以看出，最外层的函数是 main 函数，总执行时间为第二列所有数字的和。要注意，main 函数的总运行时间并不是最上面的“3”。3 表示的是，除去 A，B，C 三个子函数以外，停留在 main 函数中的执行时间。同样的道理，函数 A 总共的执行时间是 3，其中 1 个时间单位执行 A1，2 个之间单位执行 A2（其中 A2 里面有一个时间单位执行 A3）。函数 B 和函数 C 分别执行了 37 和 2 的时间。

另一个要注意的是，上面的执行时间是指总执行时间，比如在 main 里面，先用 1 时间执行函数 A，1 时间执行函数 B，1 时间执行函数 A，1 时间执行函数 B。一个合法的输入文件是：

```
main;A 2
main;B 2
```

还有一点要注意的是，输入文件的行与行之间的顺序毫无关系，可以任意交换。比如下面的写法和实力火焰图的效果完全一致：

```
main;A;A2  1
main    3
main;A 2
main;A;A1  1
main;B 2
```

main;B;B1 35

main;A;A2;A3 1

main;C 2

这个性质非常好用，由于不需要考虑记录的有序性，这可以让我们在转换文件格式的时候省去不少的麻烦。

5.4 两种文件格式的转换

上文已经给出了 trace 文件的格式和 FlameGraph 输入文件的格式。我们通过 DDMS 获得的 trace 文件很显然不能直接交给 FlameGraph 工具使用，因此要经过转换，变成 FlameGraph 的输入格式。

5.4.1 数据结构的选用

我们需要选用一种数据结构来解决这个格式转换的问题。很容易想到两种备选的数据结构。一个是多叉树，一个是栈。

使用多叉树

要是考虑到维护函数的调用关系，一个很直接的想法就是用多叉树来表示。我们可以让多叉树中的节点表示一个函数，边表示一次调用关系。这样的话，树根表示最开始进入的函数。

如果用多叉树的形式表示上一小节给出的示例调用关系的话，基本结构应该是如下的样子：

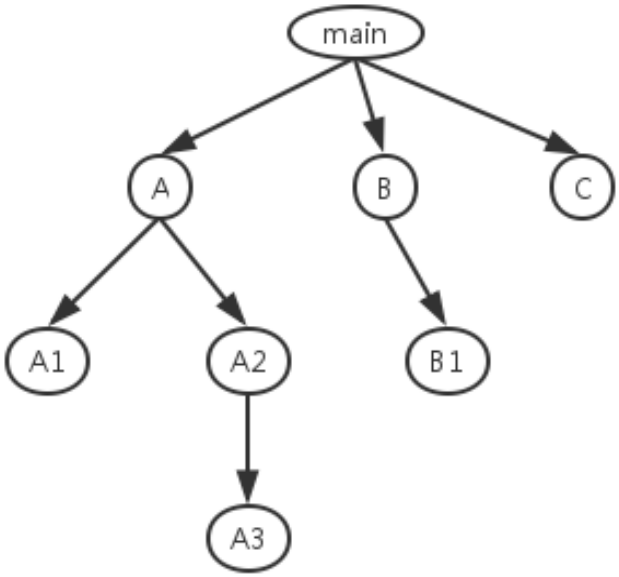


Figure 11 - Dynamic2

在多叉树的结构中，调用关系一目了然。由于保存了整个树的结构，所以进行很多操作都是很轻松的。

比如，每当出现一次进入函数的事件，就可以先记录到这条边上。每当出现一次离开函数的事件，也可以同理记录在这条边上。这些事件都添加到了这些边上之后，只需要递归遍历一次整棵树，就能将数据整理成 FlameGraph 输入格式。

使用栈

由于在内存中，函数的调用关系就是用栈来表示的。因此如果自己实现一个栈的话，应该也可以维护所需要的各种信息。

当一个函数被调用的时候，这个函数入栈；当一个函数离开的时候，这个函数出栈。因此，栈底表示的是最底层的函数，栈顶表示的是当前正在被执行的函数。

多叉树和栈的优劣比较

多叉树：

优点：展现直接，看起来一目了然。由于保存了全部时间的调用关系，因此可以进行跨时间的计算操作。

缺点：实现比栈麻烦一些，占用空间大一些。

栈：

优点：实现简单。占用空间较小，规模是函数调用的深度

缺点：比较抽象不够直观。不能同时表示两个时间点的状态。

经过斟酌比较，最终我们还是决定用栈来实现这个算法。

5.4.2 栈元素的设计

对于栈里面的每一个元素，保存他的四个数据，分别下标为 0-3

0. Method ID
1. 进入栈的时间
2. Exclusive Time
3. 最后一次操作这个元素的时间

5.5 算法整体流程

5.5.1 最初的设计

这里我们先给出算法的最基本版本的流程，之后由于考虑了各种细节问题，还会对立面的一些步骤进行补充和完善。

1. 读入 trace 文件的 Key 部分：这部分包含 Version 段、Threads 段、Methods 段，顺序读入即可。每当读入一个 Method 的时候，记录下函数名和 Method ID 的对应关系。当读到”*end”的时候，表明 Key 部分结束。
2. 读入 trace 文件的 data 部分：data 部分由”SLOW”标记开始，紧跟着是一个 offset。通过 offset 和文件总长度可以计算出记录的数量。之后顺序读入每一条记录，按照上面说的格式来分割记录，暂时全部存在一个列表里面。
3. 利用栈来模拟函数调用过程，并且计算出每个函数执行的 exclusive time。首先将栈初始化为空。然后遍历每一条记录。
   3.1. 如果为进入函数事件：
      a. 那么将这个函数入栈，记录下这个函数的 Method ID 和入栈的时间。初始的 exclusive time 设置为 0，然后将最后一次操作这个元素的时间设置为当前时间
   3.2. 如果为退出函数事件：
      a. 设栈顶为 top，top 的 exclusive time 应该加上当前时间减去最后一次操作这个元素的时间。同时，设最后一次操作这个元素的时间为现在时间，然后调用 add_to_output 函数把这条记录先存到输出缓存里面。
      b. 如果当前栈元素个数大于 1，令 below_top 为栈顶下面的元素。这个元

素的 exclusive time 要加上栈顶元素的入栈时间-该元素的最后访问时间。最后，令该元素的最后访问时间为当前时间。

c. 栈顶弹出。

4. 输出 output 里面的结果

5.5.2 考虑空闲的时间

当所有函数都没有在运行的时候，时间仍在在流逝。在上面的算法中，这一部分的时间完全没有被考虑进来。因此，我们引入了变量 blank 和 last_time，专门处理这一部分空闲的时间。变量 blank 表示当前记录下来的空间时间，last_time 表示最近一次处理到的时间。因此我们对算法进行如下的修改：

3.1. 如果为进入函数事件：

a. 如果当前栈空，那么说明刚刚是空闲的时间，因此要将这一部分时间记录到 blank 里面。所以让 blank 加上 time – last_time。

b. 将这个函数入栈，记录下这个函数的 Method ID 和入栈的时间。初始的 exclusive time 设置为 0，然后将最后一次操作这个元素的时间设置为当前时间。

这样修改之后，空闲的时间就被记录在了 blank 函数里面，最后添加到 output 里面即可。

5.5.3 考虑截断效应

上面给出的算法可能会由于截断效应的存在而变得不正确。比如考虑下面两段记录：

第一段记录：

时刻 0，进入函数 A

时刻 1，退出函数 A

时刻 2，退出函数 B

….

第二段记录：

时刻 0，进入函数 A
时刻 1，进入函数 B
时刻 2，退出函数 B
** end **

在第一段记录中，当函数 B 退出的时候，会出现栈空的情况。为什么会出现这种情况呢？这是因为由于截断效应的出现，进入函数 B 这一时间早于开始记录的时间。如果出现这种情况，会导致内存错误，程序崩溃。

在第二段记录中，当记录全部解析完毕后，栈里面仍然留有元素。这是因为，函数 A 退出的时间还要晚于记录结束的时间。如果出现这种情况，则函数 A 的运行记录就不会被输出出来，结果不完整。

针对第一种截断效应的算法修改

大体思路：在得到函数退出的事件的时候，如果栈为空，那么要进行特殊的处理。直接修改算法：

3.2. 如果为退出函数事件：
   a. 如果当前栈非空：
      i. 设栈顶为 top，top 的 exclusive time 应该加上当前时间减去最后一次操作这个元素的时间。同时，设最后一次操作这个元素的时间为现在时间，然后调用 add_to_output 函数把这条记录先存到输出缓存里面。
      ii. 如果当前栈元素个数大于 1，令 below_top 为栈顶下面的元素。这个元素的 exclusive time 要加上栈顶元素的入栈时间-该元素的最后访问时间。最后，令该元素的最后访问时间为当前时间。
      iii. 栈顶弹出。
   b. 如果当前栈空：
      i. 那么将所有在 output 中的函数都加上自己的函数名作为前缀。然后将自己添加到 output。
      ii. 将 blank 置为 0.

针对第二种截断效应的算法修改

在算法最后添加第四步，处理栈里面的剩余元素:

4. 从栈顶开始遍历栈里面的所有剩余元素

4.1. 该函数的 exclusive time 加上现在时间-最后一次该记录的处理时间，该函数的最后处理时间设为当前时间。然后调用 add_to_output 函数添加到 output 里面。

4.2. 如果当前栈内元素多于一个，令 below_top 为栈顶下面的元素。这个元素的 exclusive time 要加上栈顶元素的入栈时间-该元素的最后访问时间。最后，令该元素的最后访问时间为当前时间。

4.3. 弹出这一条记录。

5.5.4 add_to_output 函数实现

这个函数中，首先先将栈里面的函数名依次拼接成 A;B;C 的格式，然后查询在 output 中是否已经有相应记录。如果没有，则创建记录。之后再对应记录上面加上这次计算出的 exclusive time 的值。

5.5.5 最终的算法流程

1. 读入 trace 文件的 Key 部分：这部分包含 Version 段、Threads 段、Methods 段，顺序读入即可。每当读入一个 Method 的时候，记录下函数名和 Method ID 的对应关系。当读到”*end”的时候，表明 Key 部分结束。

2. 读入 trace 文件的 data 部分：data 部分由”SLOW”标记开始，紧跟着是一个 offset。通过 offset 和文件总长度可以计算出记录的数量。之后顺序读入每一条记录，按照上面说的格式来分割记录，暂时全部存在一个列表里面。

3. 利用栈来模拟函数调用过程，并且计算出每个函数执行的 exclusive time。首先将栈初始化为空。然后遍历每一条记录。

3.1. 如果为进入函数事件：

a. 如果当前栈空，那么说明刚刚是空闲的时间，因此要将这一部分时间记录到 blank 里面。所以让 blank 加上 time － last_time。

b. 将这个函数入栈，记录下这个函数的 Method ID 和入栈的时间。初始的 exclusive time 设置为 0，然后将最后一次操作这个元素的时间设置为

当前时间。

3.2. 如果为退出函数事件：

a. 如果当前栈非空：

i. 设栈顶为 top，top 的 exclusive time 应该加上当前时间减去最后一次操作这个元素的时间。同时，设最后一次操作这个元素的时间为现在时间，然后调用 add_to_output 函数把这条记录先存到输出缓存里面。

ii. 如果当前栈元素个数大于 1，令 below_top 为栈顶下面的元素。这个元素的 exclusive time 要加上栈顶元素的入栈时间-该元素的最后访问时间。最后，令该元素的最后访问时间为当前时间。

iii. 栈顶弹出。

b. 如果当前栈空：

i. 那么将所有在 output 中的函数都加上自己的函数名作为前缀。然后将自己添加到 output。

ii. 将 blank 置为 0.

4. 从栈顶开始遍历栈里面的所有剩余元素

4.1. 该函数的 exclusive time 加上现在时间-最后一次该记录的处理时间，该函数的最后处理时间设为当前时间。然后调用 add_to_output 函数添加到 output 里面。

4.2. 如果当前栈内元素多于一个，令 below_top 为栈顶下面的元素。这个元素的 exclusive time 要加上栈顶元素的入栈时间-该元素的最后访问时间。最后，令该元素的最后访问时间为当前时间。4.3. 弹出这一条记录。

5. 将 blank 记录添加到 output。

6. 输出 output 中的结果。

5.6 测试结果

5.6.1 测试对象和测试内容

对于使用火焰图的动态分析方法，我们想其进行两个测试。

第一个测试仍然使用和静态分析一样的测试样本，Omnidroid。针对上面在静

态分析中找出的部分性能缺陷，观察使用静态分析能否找出类似的性能缺陷。

第二个测试的目的是，对于表现结果而言，火焰图能否更直观的表现出性能缺陷的位置。这一个测试选用一个开源的游戏辅助软件，PadSolver，因为这个软件里面明显存在卡顿现象。我们将用不同的表现形式观察火线图的展示效果。

5.6.2 第一个测试

在真机上运行起 Onmidroid 之后，操作手机进入到添加过滤器的界面。然后在 DDMS 界面 点击 Start Method Profiling，在手机上选择添加过滤器，在 DDMS 上点击 Stop Mrthod Profiling。会得到下面的火焰图：

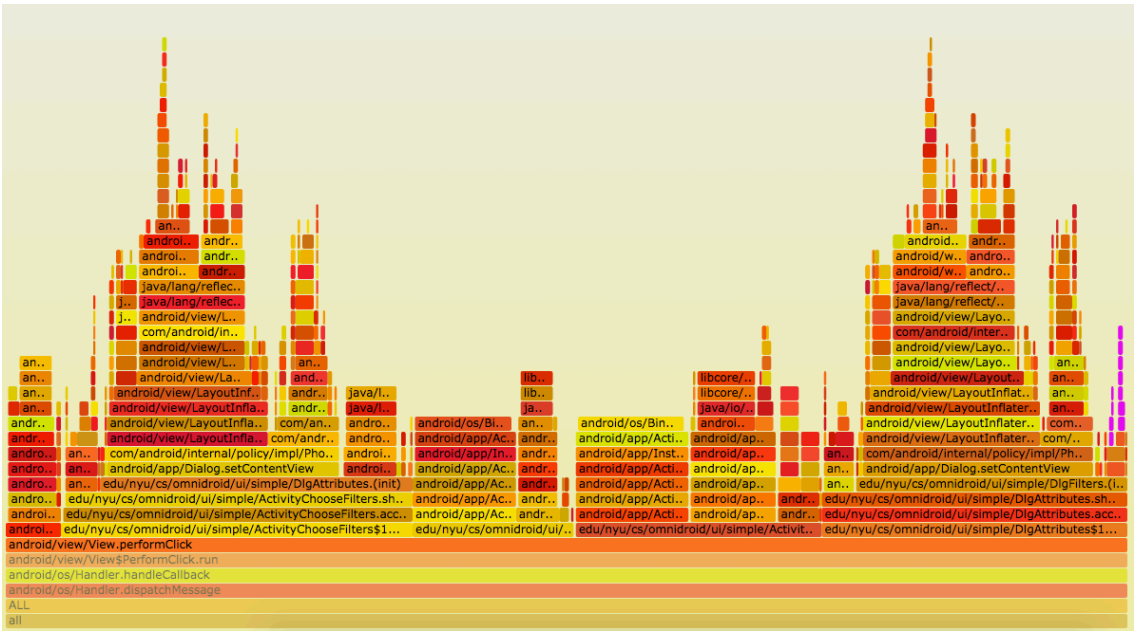


Figure 12 - Dynamic3

其中操作 sql 数据库相关的部分在这个图的右边用粉色高亮标记出来了。可以看出，数据库操作的确占用了一定的时间，可能造成图形用户界面响应速度上的损失。

5.6.3 第二个测试

使用火焰图作为表示方法

PadSolver 是一个开源的游戏辅助软件，具体功能就不做过多讲解了。当用户点击按钮开始载入图片的时候，会发生明显的卡顿。因此，为了寻找发生卡顿现象的根本原因，我们先采用本章节描述的基于火焰图的动态分析方法。

1. 将这个软件的源码从 github 上面下载下来，然后在 eclipse 里面编译成功，连接真机调试。

2. 进入 DDMS 界面，点击 Start Method Profiling，手机上点击载入图片，完成后再 DDMS 界面中点击 Stop Mrthod Profiling。

3. 保存 trace 文件，使用 parser.py 进行格式转换，然后使用 FlameGraph 工具导出 svg 格式的火焰图。

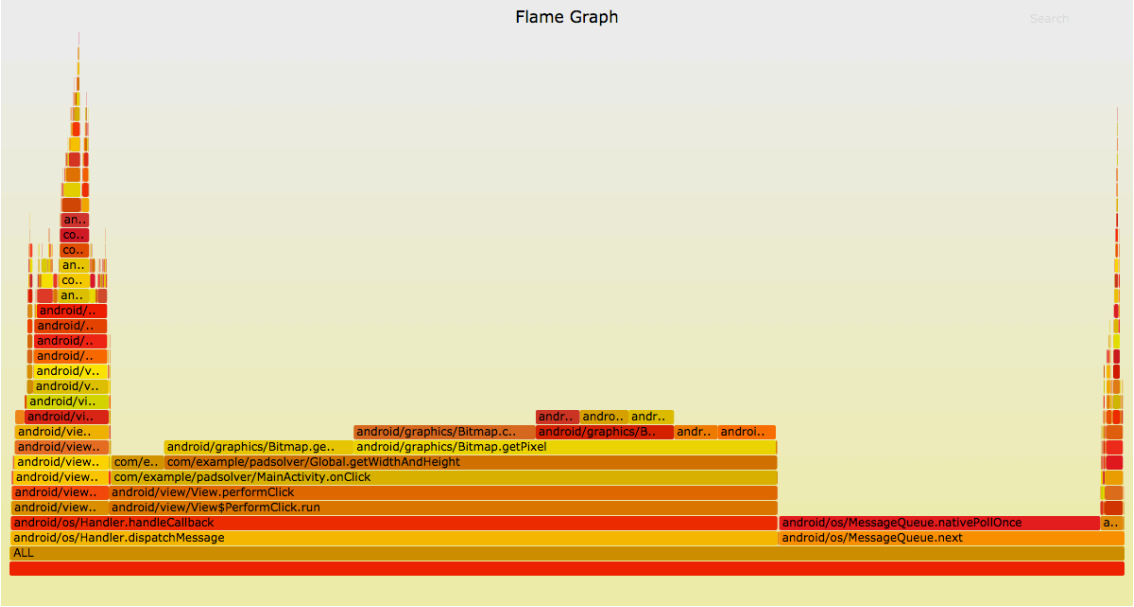


Figure 13 - Dynamic4

可以直观看出，函数 com/example/padsolver/Global.getWidthAndHeight 占用了非常长的时间。所以这时候，开发者可以准确定位到这个函数，观察里面的调用关系和具体实现并且进行修改。

使用 trace 文件作为表示方法

上文所处理的 trace 文件大小为 852KB，很明显，这是人类无法阅读分析的。

5.6.4 结果分析

根据实验结果，我们认为，基于火焰图的安卓应用卡顿的分析的确效果不错。应用中出现的卡顿时间越长，表现出来的效果就越直观。

然而这种方法也有一定的缺点：

1. 由于这种方法基于动态分析，因此需要将程序运行起来。有时候这可能很不方便，比如运行有些程序所需的代价很大。

2. 当有些卡顿现象出现时间很短的时候，火焰图中对应的部分会不明显，这时候开发者可能不会发现这些问题。然而在上面的静态分析中，仍然会给出对应的提示。

第 6 章　总结与展望

6.1　实验总结

在这篇文章中，我们以安卓平台上面的应用为例，对智能手机应用的性能缺陷，尤其是带有 GUI 应用的卡顿延迟现象进行了研究。我们讨论了两种分析方法，分别基于静态分析和动态分析。

基于静态分析的部分：本文实现了一种基于静态分析的性能缺陷检测方法。

基于动态分析的部分：我们将火焰图引入了安卓应用的性能缺陷分析，并且设计了算法来处理 DDMS 产生的 trace 文件。

经过测试，两种方法都可以有效进行性能缺陷的检测，有时也要根据实际情况，选择合适的检测方法。

6.2　后续工作

接下来，可以考虑将静态分析和动态分析结合起来，进行性能缺陷的分析。比如，先使用静态分析对性能缺陷的产生位置进行大致的定位，然后通过自动的动态运行对定位到的部分进行大量测试和数据收集。最终对这些收集到的信息进行分析，从而更准确地定量地找到性能缺陷的根本原因。

插图索引

参考文献

[1] Graphics Performance Analyzer for Android, https://software.intel.com/en-us/gpa

[2] Performance Analysis of Android Platform, https://jawadmanzoor.files.wordpress.com/2012/01/android-report1.pdf

[3] Finding Performance and Power Issues on Android Systems, http://elinux.org/images/8/83/Finding_performance_and_power_issues_on_android_systems--eric_moore.pdf

[4] Liu Y, Xu C, Cheung S C. Characterizing and detecting performance bugs for smartphone applications[C]//Proceedings of the 36th International Conference on Software Engineering. ACM, 2014: 1013-1024.

[5] Linares-Vásquez M, Vendome C, Luo Q, et al. How developers detect and fix performance bottlenecks in Android apps[C]//Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on. IEEE, 2015: 352-361.

[6] Yoon H J. A study on the performance of Android platform[J]. International Journal on Computer Science and Engineering, 2012, 4(4): 532.

[7] Wang X, Guo Z, Liu X, et al. Hang analysis: Fighting responsiveness bugs[C]//ACM SIGOPS Operating Systems Review. ACM, 2008, 42(4): 177-190.

[8] Jin G, Song L, Shi X, et al. Understanding and detecting real-world performance bugs[C]//ACM SIGPLAN Notices. ACM, 2012, 47(6): 77-88.

[9] Profiling and Debugging Tools for High-performance Android Applications Stephen Jones, Product Line Manager, NVIDIA (sjones@nvidia.com), https://developer.nvidia.com/sites/default/files/akamai/mobile/files/Profiling%20and%20Debugging%20Tools%20for%20Android.pdf

[10] A Survivor's Guide to Java Program Analysis with Soot, http://www.brics.dk/SootGuide/

[11] Omnidroid, https://code.google.com/archive/p/omnidroid/

致　谢

感谢陈渝教授在整个毕业设计的过程中对我的指导和鼓励，这给我带来了很大的动力和帮助。尤其是在我遇到困难的时候，陈渝教授对我细心指导，答疑解惑。因此，我对陈渝教授表示由衷的感谢！

感谢陈康教授对我毕业设计进度一贯的关心与督促！

感谢实验室的师兄师姐们对我的帮助！

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究 工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的 研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出 贡献的其他个人和集体，均已在文中以明确方式标明。

签名：_____日期：_____

附录　外文资料的调研阅读报告（或书面翻译）