**Panappticon：基于事件跟踪来测量移动应用和平台的性能**

**摘要**

提高和优化手机用户接收性能需要理解设备、系统和应用真实工作负载时的行为。但是，测量这样的性能是非常有挑战的，因为在现代应用程序中使用多线程异步编程范式，并且用来响应用户输入事件的硬件和软件具有多个层次。我们使用一个针对于安卓的轻量级、全系统、细粒度的事件跟踪系统Panappticon自动识别用户事务的关键执行路径。Panappticon检测应用、系统和内核软件层并且能够识别源于应用设计缺陷、动力不足的硬件和明显无关的应用程序之间的有害交互导致的性能问题。我们完成一个有14位用户历时一个月的由Panappticon监控的安卓智能手机系统的研究，这项研究揭示了很多系统设计者、应用开发者和设备制造者感兴趣的提高性能的特殊问题和领域。

**介绍**

大多数应用程序是交互的。典型地，用户触发一系列操作中的一个输入最终形成用户可见的输出，通常情况下是显示的更新。用户体验依赖于接收响应能力，所以控制这种事务的延迟对于应用程序设计者、操作系统设计者和硬件平台设计者是非常重要的。

我们定义用户在设备上操作而开始（例如，一次屏幕的点击或者按键的点击）、显示更新而结束的一系列操作作为用户接收事务。直观地，事务捕捉用户指示设备去做一些事情和期待的结果被显示出来的间隔。尽管这种交互延迟对于用户体验很重要，几乎没有存在的开发工具支持来定义或者决定慢事务的原因，导致开发者一直在黑暗中。这主要是由于交互型应用的异步、多线程特点。为了保存用户接口响应，应用程序必须在后台线程中做时间长的或者可能阻塞的操作，使得单个事务执行流的轨迹变得复杂。由于同一个设备上运行的其他无关应用程序或者系统进程可能影响感知性能，使得分析变得更加复杂。并且，分析一个应用程序的内部行为来完整地表征事务延迟和原因是不够的。AppInsight监控应用程序边界来研究用户事务。它识别一些性能问题，但是不能解释由进程间直接交互导致地差地性能。我们在Section6部分展示一个由我们的工具识别这个问题的例子。所有的影响彼此的进程和应用程序应该被考虑在内。

本项研究中，我们描述了Panappticon，一个识别真实世界用户的用户接收事务并且可以帮助诊断差的用户接收性能的原因的系统。我们用一个由14各用户历时一个月的研究来举例证明它。Panappticon对以下三类人有用。

应用开发人员可以使用Panappticon来识别无效应用程序代码并且根据它来进行优化。比如，在我们的研究中，我们发现一个受欢迎的安卓应用程序Reddit News由于它的主线程和由主线程触发的非关键系统活动间的CPU争夺情况导致有很多慢事务。非关键工作延迟执行直到用户事务完成之后将会提高用户接收性能。

操作系统开发者能够使用Panappticon来优化系统策略。比如，在我们的研究中我们发现，尽管为交互式工作负载设计，Android中包含的默认DVFS（动态电压和频率缩放）调控器几乎翻倍持续时间大约超过80毫秒的交易的延迟。需要一个改进的调控器并且我们获得这个内容的相关建议。

硬件设计者和制造者能够使用Panappticon来更好地理解未来设备结构决定地影响。我们观察到典型的非游戏应用是非并行的，并且不享受多内核处理器的好处，这就是说设计者不应该忽略单核性能。

Panappticon通过异步调用和跨进程通信边界来跟踪多线程执行流从而建立用户输入和显示器显示的因果关系。这**通过监测事件处理程序、异步调用接口和包括用户层和内核层的跨进程通信策略以记录可以重建执行流程的事件。系统更是记录资源使用信息，包括上下文切换和网络接口和存储设备阻塞来帮助识别慢事务的根源。**

我们在十个开源应用程序上验证了Panappticon，手动确认检测到的用户事务和延迟是正确的。Panappticon平均产生6.1％性能负载，对电池寿命的影响并不明显。

这项工作作出三大贡献。

•我们提供了一种不引人注目的方法和设计，用于根据输入触发的操作之间的因果关系提取用户感知的事务延迟。

•我们描述了Panappticon，一个开源的系统，它采用上述方法来自动表征用户感知的事务，并提供详细的资源使用信息，从而可以根本原因诊断慢事务的原因。

•我们展示了在手机上运行Panappticon的14个Android用户一个月的真实用户研究结果，并提供了三个案例研究，显示了Panappticon如何有益于应用开发人员，系统开发人员和智能手机制造商。 数据跟踪是公开的（http://ziyang.eecs.umich.edu/projects/panappticon）。

3, start, 
S. Finish 
and 
Figure I: Illustrative example of a user-Ikrcened transaction: the 
horizontal axis represents time and the vetltcal axis distinguishes 

图1描绘了用户感知的事务。 想象一下下载和显示名人报价的简单应用程序。 如图1所示，通过按钮按钮启动事务。 在由此输入触发的UI事件处理程序中，主UI线程将异步任务提交给工作线程以下载报价。 在执行异步任务期间，工作线程可能会在等待来自网络的数据包时阻塞。 下载完成后，工作线程将一条消息发回UI线程以显示报价。 在此示例中，用户输入和显示更新之间的操作形成一个用户感知的事务。

Panappticon旨在识别这种用户感知的事务，并确定每个事务的性能瓶颈。 为了在上述示例中实现此目标，系统必须记录用户输入，并且基于异步调用在UI事件处理和工作线程执行之间建立因果关系。 然后，系统必须记录网络块，最后通过跟踪发布的消息将工作线程执行链接到UI更新。

为了优化用户感知的事务延迟，我们需要确定每个事务的关键路径，并了解主要组件。 关键路径是瓶颈执行路径，其长度捕获用户输入和显示更新之间的感知等待时间。 关键路径上的节点导致延迟; 增加执行时间将增加输入和更新之间的延迟。 在上面的示例中，通过工作线程的输入和更新节点之间的路径表示关键路径，网络导致大部分延迟。

**2.2设计挑战**

我们在设计Panappticon时面临以下主要挑战。

**•用户事务可以涉及跨线程和进程边界的执行。** 例如，在前面的例子中，主线程提交后台线程进行异步处理的工作，这是一个常见的Android编程模式，如第4节所述。更具挑战性的是，一些应用程序包含两个或多个独立的进程，一个线程运行UI， 其他线程与所有积极参与用户感知交易进行异步工作。 因此，我们必须跟踪跨线程和进程边界的异步（和同步）调用。

**•显示更新并不总是由最近的用户输入事件引起。** 在上述示例中，可能通过输入事件和显示更新之间的系统状态的改变（例如，网络连接的改变）来触发其他显示更新。 在更复杂的应用程序中，第二个用户事务可能在第一个完成之前启动。 这两种情况都会导致显示更新由于其他用户输入事件与引起的该显示的用户输入事件分离。 这样可以防止用户将其显示更新与其最近的输入事件分组。 Panappticon必须明确地跟踪操作之间的因果关系。

**•有必要了解底层的硬件状态。** 我们的目标之一是帮助系统和应用开发人员确定长时间交易的原因。 可能的原因包括处理器的争用，网络或磁盘IO上的长时间阻塞或系统策略（如DVFS）的问题。 例如，在我们的运行示例中，如果网络阻塞时间减少，用户感知的等待时间也将减少。 为了实现这一目标，我们必须有效地记录细粒度的资源使用信息，例如IO阻塞时间和上下文切换。

**•移动设备资源有限。** 系统必须具有低性能和能源开销，以免影响用户体验。 有限的智能手机CPU吞吐量，内存和能量容量可防止记录非必要数据，并将用户事务分析归功于服务器。

**3.方法概述**

我们开发并实施了基于事件的跟踪基础架构，可以有效地捕获（1）操作之间的关系，以识别线程或进程之间的用户事务;（2）线程的哪些资源（例如CPU，网络，磁盘等） 在每一时刻使用或阻止，以显示性能瓶颈。 本节介绍我们的方法，如图2所示。

Input 
Thread 
Service 
Worker Thread 
up date 
Input 
Thread 
-RPC Bl„k 
.-4 
Submit 
RPC eek 
Begin 
Finish 
Fork 
(a) Execution traces divided into "atomic" intervals. The dotted ar- 
rows show the dependencies between intervals. The grey intervals 
are unrelated to this user transaction and the dashed portions indi• 
eate the thread is idle. 
Worker Th read 
(b) The events logged by Panappticon in order to reconstruct the re- 
lationship graph. Submit events mark the submission of tasks to the 
UI. AsycTask. and Service threads. begin and finish events demarcate 
the rocessin ofeach task. and a fork event marks the creation of the 
Thread 
Service 
Worker Thread 
oo 
O 
O 
O 
worter thre aä. 
UI Thread 
Asyn k 
Service 
Worker T h read 
O 
O 
O 
o 
O 
O 
O 
O 
(e) Relationship graph showing execution dependencies for this 
transaction. Each node represents one execution interval. with an in• 
edge indicatin the interval was triggered by the preceding 
node. Given this g . the transaction latency is easily computed 
from the critical path. 
(d) From the events logged above. the full relationship graph can 
be reconstructed. The begin and finish events demarcate the execu• 
tion boundary for each node and the submit and fork events indicate 
which node triggered an interval. 
Figure 2: Example execution sequence illustrating our methodology for uswperceived transaction extraction. Figures 2(a) and 2(c) illustrate 
that the sequence can be viewed as a directed acyclic graph of dependent execution intervals. Figures 2(b) and 2(d) show how the graph 
can be reconstructed from a log of simple events. In this transaction. the user input enqueues an AsyncTask that. after communicating with 
a background ser,'ice via RPC. updates the display. It also forks a background thread to read from disk and then update the display. The 
transaction ends after the second display update. 

**（a）执行痕迹分为“原子”间隔。** 虚线箭头显示间隔之间的依赖关系。 灰色间隔与该用户事务无关，虚线部分表示线程空闲。

**（b）Panappticon记录的事件，以重建关系图。** 提交事件标记将任务提交到UI，AsycTask和Service线程，开始和结束事件划分每个任务的处理，fork事件标记工作线程的创建。

**（c）关系图显示此事务的执行依赖关系。** 每个节点表示一个执行间隔，其中一个进入边缘指示间隔由前一个节点触发。 给定该图表，可以从关键路径轻松计算事务延迟。

**（d）从上面记录的事件可以重建完整的关系图。** 开始和结束事件划分每个节点的执行边界，并且提交和fork事件指示哪个节点触发间隔。

**3.1方法概述**

我们在线程和进程之间跟踪用户事务的技术基于识别和链接线程执行的“原子”间隔。 这样的间隔表示连续发生的工作，例如工作线程从任务队列处理一个任务。 处理相同事务的间隔因果依赖 - 一个间隔触发下一个的执行。 图2（a）示出了划分为间隔的示例执行跟踪，箭头指示因果关系。 图2（c）显示了我们希望获得的关系图。

提取关系图需要（1）将每个执行跟踪分成“原子”间隔，（2）识别间隔之间的因果关系，（3）识别用户接受事务的初始（例如用户输入）和终端（例如，显示更新）间隔。 我们记录每个任务足够的事件，如图2（b）所示。 对于前两个任务，我们监测Android平台和内核以记录流行的编程范例的事件。 为了解决第三个任务，我们监测平台记录用户输入和显示更新，标记当前时间间隔。 这些允许图形构建，如图2（d）所示。

为了确定延迟的原因，我们记录指示资源使用的事件，例如CPU的上下文切换和磁盘和网络访问的阻塞时间。 第4节描述所有捕获的事件。

**3.2架构概述**

ace 
Application 
Device side 
side 
Datvik VM 
user 
Application 
E vent 
us. r logger 
Kernel logger 
Figure 3: System architecture overview. 

Panappticon包含图3所示的五个主要组件：用户空间记录器，内核空间记录器，事件收集器，服务器端收集器和用户事务分析器。

用户空间记录器和内核空间记录器：用户空间和内核空间记录器记录上一节中提到的事件。 具体来说，输入事件，显示更新事件以及指示因果关系的大多数事件被捕获在用户空间中，其中使用高级编程范例使事件推理更容易。 内核记录器捕获资源利用率事件和一些指示跨进程边界的因果关系的事件，例如分叉和IPC事务。 第4节列举了这两个记录器捕获的所有事件。 为了最小化这些记录器的性能影响，缓冲区事件记录在内存中，当缓冲区已满时将它们批量发送到收集器。

事件收集器和服务器收集器：事件收集器负责将记录器中的跟踪传输到服务器端收集器进行处理。 为了最小化性能和能源开销，并避免丢失数据，只有在WiFi可用时，才会批量上传日志。 失败的传输被缓冲到SD卡并稍后重试。

用户事务分析器：用户事务处理分析器使用3.1节描述的方法提取关系图。 从这些图中，它提取用户事务及其相应的用户感知延迟和资源使用。 第4节描述了图提取过程。

**4.监测详情**

在Panappticon的设计可以确定之前，有必要回答以下问题。 （1）需要捕获哪些数据？ 它们应该足以以低开销来识别每个用户事务及其资源使用。 （2）应如何使用这些数据构建关系图以将用户输入与显示更新相关联？ （3）每笔交易如何进行资源核算？ 本节回答这些问题。

Panappticon基于Android 4.1.2。 下面的一些实现细节是这个Android版本的具体内容。

**4.1背景：Android**

Android是由Google为智能手机和平板电脑等移动设备开发的基于Linux的操作系统。本小节总结了Android属性，为理解Panappticon提供了背景资料。

专门的UI处理：Android上的应用程序是以UI为中心的。所有UI相关事件，包括提供用户交互和更新显示，都在一个专用线程上处理，这也是每个应用程序的主线程。为了维护UI的高响应度，开发人员应该避免冗长的操作，或阻止UI线程[3]。因此，他们应该创建单独的工作线程。该属性激励我们跟踪跨线程的异步调用。

Looper线程：由Android系统管理的大多数Java线程，包括用于UI事件的每个应用程序的主线程，都使用消息队列模型。消息被放置在队列中，并且线程无限期循环，处理来自队列头部的消息。在Android中，这样的线程称为Looper线程，并共享消息队列和looper功能的常见实现。共享代码便于记录消息提交和执行事件。

**4.2我们要捕捉什么信息？**

Panappticon收集的信息量与其开销之间存在折衷。例如，系统框架中应用程序在交易过程中可能会调用的每个方法的完整跟踪将使我们能够充分了解系统状态。但是，这会使应用程序的执行速度以数量级减慢。为此，我们记录识别用户感知交易及其性能瓶颈所需的最少信息量。该信息可以放在以下类别中：（1）用户交互事件，包括屏幕触摸和按键，（2）线程内和跨线程的异步调用和回调之间的因果关系，（3）线程和进程之间的进程间通信， 4）各种线程同步机制和线程之间因同步引起的因果关系，（5）每个线程的资源计费，如网络和磁盘IO上的上下文切换和阻塞;（6）线程之间的其他因果关系，例如fork线程， （7）显示更新，（8）有助于跟踪前景应用程序和应用程序名称的附加信息。每个事件的记录包含以下字段：Timestamp，Event\_type，TID，Data，（CPU\_core）。 CPU\_core指定与事件关联的核心，仅适用于内核事件。表1总结了Panappticon日志的事件。我们现在解释每种类型的事件。

[PC calls 
mutcx, 
semaphore. 
an d waitqueue 
Display 
in formation 
NPU 
UI_KEY 
DEQUEQUE_MSG 
INDER_P DU E_ NEWAY 
BINDER_REPI.Y 
BINDER_CONSUME 
CONT EXT SWITCH 
'ORK 
UI_INVALIDATE 
UI_UPDATE 
THREAD_NAME 
ENTER/EXIT_FOREGROUND 
Table 1: 
User 
List of Events Captured 
message u 
krk_id, notify_pid 
old_pid , new_pid 
t_pid. 
of 
with q. q —_id 
ed thread 
10 w 
to the b I mks 
Remote thread value hack 10 caller 
A thread 
waiting to 
Resume from block waiting access to 
Notify wailing thread 
Context switch from old_pid to ncw_pid 
on mang or 
on t o 
child 
Invalidates the viea, display update 
the v i cm 
Thread l_pid 
Current PID being app 

**用户输入：**我们记录由于屏幕或按钮触摸的输入事件以及来自软件键盘的输入事件。 在Android中，第一种类型的输入通过View类中的onInput（）回调方法直接发送到前台应用程序。 与第一种类型的输入不同，软件键盘输入通过另一个systemUI应用程序调度到前台应用程序，该应用程序将屏幕触摸事件转换为键盘输入。

**异步调用和回调：**正如我们在第4.1节中提到的，Android应用程序以UI为中心的特性要求开发人员使用异步工作线程进行冗长的操作。 为了实现这一点，Android中有两种常见的编程模型**：（**1）启动一个工作线程，并在UI线程上发布一条消息，以便在完成工作后更新显示，或（2）将任务提交到线程执行程序池。

为了处理第一种情况，我们在Android框架库中调用MessageQueue类，以捕获消息入队和出队事件之间的因果关系。 每个消息都与一个唯一的ID相关联。 对于第二种情况，我们在通用Java库中调用ThreadPoolExecutor类来记录任务提交和消费之间的因果关系。 使用任务ID匹配这些事件。

**进程间通信：**如上所述，Android使用称为Binder for RPC的内核级进程间通信实现。 对于每个进程，**Binder管理一个线程池以执行传入的RPC请求。 对于每个呼叫或Binder事务，我们通过跟踪跨进程边界的事件来记录完整的RPC调用。**

**同步机制：**虚拟资源（工作线程，共享数据段等）的争用可能导致缓慢的事务。访问此类资源通常由同步原语进行调停，因此我们**记录对以下核心内容的有争议的访问：waitqueues，semaphores，mutexes和futexes。**具体来说，当一个线程阻塞等待访问，当它从该块恢复时，以及在释放一个原语之后，哪个等待的线程被唤醒，我们记录。我们不会记录锁定，解锁或spinlock事件由于访问量 - 有争议的访问是非常罕见的。

**资源统计：**为了帮助确定每个事务的瓶颈，我们记录访问Android应用程序使用的三种主要的时间共享资源：**处理器，网络和磁盘。对于处理器，我们记录每个上下文切换，包括传入线程ID，传出线程ID和旧线程的新状态（仍然可运行，可中断睡眠等）。对于磁盘和网络，我们记录当线程阻塞读取请求时，然后恢复。**

**显示更新：**Android为开发人员提供了两个主要路径来更新显示：框架库中的**View类**和OpenGL直接呈现显示。 Panappticon目前仅考虑第一条路径，因为OpenGL主要用于游戏应用程序中的图形渲染。 这种游戏应用中用户交易的概念与我们在交互式应用中的定义不同，因为使用动画，如第5节所述。

**附加信息：**除了捕获事件之间的因果关系所需的所有信息之外，我们还收集了更多信息，以帮助我们更好地了解交易的上下文。 例如，我们记录了进入和退出前台的应用程序，以区分用户与系统应用程序进行交互的应用程序。 这是通过对Activity框架类的更改完成的。 同样，内核记录每个线程的名称。

**4.3我们如何构建关系图？**

ord«ing 
relationship 
EMI 
Figure 4: Illustrative example Of relationship graph. 

Listing I Example trace of correlation graph 
(1) 
(EMI) 
id:l 
(DNII) 
id:l 
id:l 
(EM2) 
(lht2) 
(INV) 
USER_INPUT 
FAQUEUE_MSG 
DEQUEUE_MSG 
SUBMIT ASYNCTXSK 
coNSLIME_ASYNcrASK 
SOCK_RESUME 
ENQUEUE_MSG 
DEQUEUE_MSG 
ULINVALIDATE 
UI_UPDATE 
pid:l 
pid:l 
pid:l 
pid: 
pid: 
pid. 
pid:O 
message 
message 
task 
_id:l 
task 
message_ld •2 
message _ •d 

为了基于事件流识别用户事务，我们基于事件之间的关系构建有向非循环图。 跟踪中的每个事件条目都标识为图形中的一个节点。 边缘表示事件对之间的关系。

图4说明了基于清单1中的示例跟踪构建的关系图。节点名称和事件之间的映射也显示在跟踪中。 跟踪的逻辑在第2节中描述。

我们确定两种类型的关系：因果关系和时间排序。

**因果关系：**较早间隔触发后者的两个执行间隔之间的关系，如第3.1节所述。 它由图中的实线表示。 例如，消息入队事件触发消息出队动作。 换句话说，没有消息入队操作，消息出队操作将不存在。 我们确定了与因果关系相关的以下节点对。

•具有相同message\_id的ENQUEUE\_MSG和DEQUEUE\_MSG。

•具有相同task\_id的SUBMIT\_ASYNCTASK和CONSUME\_ASYNCTASK。

•BINDER\_PRODUCE或BINDER\_REPLY事件和具有相同transaction\_id的BINDER\_CONSUME。

•由FORK事件创建的两个节点：节点在父线程上表示FORK事件，而节点表示子线程上的初始执行。

•UI\_INVALIDATE及其最接近的UI\_UPDATE在同一个线程中。

**时间排序：**执行间隔内的事件之间的关系。 它由虚线表示。 例如，在前面的示例中，消息1被排队，而由输入触发的回调方法被执行。 类似地，在执行出队消息1期间提交异步任务。

时间排序的一个主要挑战是确定何时结束执行间隔。 如果未正确确定，同一线程上的所有事件将被虚假连接，导致用户事务被错误地分组。 我们将线程放在Android应用程序中分为两类，我们对每种类型使用不同的方法。

•基于任务的线程是Android中最常见的背景线程模式。 当队列为空时，这些线程从队列中消耗任务并阻塞。 每个任务的结束指示执行间隔的终止。 例如，每个应用程序中的主UI线程是一个等待传入消息并处理它们的Looper线程。 处理一个消息时发生的所有事件都属于一个执行间隔。 同样的方法适用于Binder线程和异步任务线程，这些线程等待新的事务和新任务。 我们没有明确地显示其他后台线程。 相反，来自锁定原语的事件指示任务队列的生产者/消费者，允许推断执行间隔。 第4.5节介绍了使用WebKit应用这种方法的应用。

•工作线程也可能被分叉一次执行。 在这种情况下，我们的方法会自动推断执行间隔。

我们假设在从队列处理特定任务时不执行不相关的工作。 A（可能是错误的）程序员可能具有特定消息的处理程序来检查不相关的条件，并使消息处理它，导致错误依赖。 在实践中，这应该是罕见的，因为Android API鼓励良好做法来维护响应式的用户界面线程。 手动检查以验证Panappticon的应用程序都没有显示这样的错误依赖。 此外，导致显示更新的唯一（罕见）错误依赖将影响关键路径分析。

使用上述方法，我们可以在重叠时推断出单独的用户事务。 通过在每个事务中进一步从UI\_INPUT和UI\_UPDATE中提取关键路径，我们可以导出每个用户感知事务的延迟。

**4.4资源会计**

我们使用资源会计来回答的主要研究问题是“有明显延迟的交易延迟的原因是什么？”和“加快速度可以做些什么？”为此，我们分析了关键的资源使用 每个事务的路径分两步。

首先，将资源会计相关的内核事件添加到相关图中以指示物理资源的使用。 特别地，上下文切换事件允许CPU使用的每个线程计算。 类似地，网络和磁盘IO访问事件允许网络和磁盘使用的线程计费。 除了物理资源之外，我们还在图中添加了同步事件来表示虚拟资源的使用。 例如，线程可以阻塞互斥体并等待其他线程释放它。 请注意，这些事件根据时间相关性被添加到图形中。

其次，我们分析了关键路径上的边缘，以了解其延迟的原因。所有边缘都属于以下类别。

•指示相应线程的边缘正在运行并占用处理器。例如，在两个上下文切换之间发生的任何事件之间的边。由于这种边缘而导致的延迟取决于处理器速度。

•建议对应线程的边缘被阻止，等待一些资源。我们在以下类别中进一步分配不同的资源。 （1）物理资源，如网络，IO和CPU.Waiting for CPU意味着线程在仍然有资格运行的情况下上下文切换。 （2）虚拟资源，如锁或线程执行。例如，异步任务的提交和消耗之间的延迟可能是由于工作线程被占用而导致的，因此阻止了新任务的消耗。缩短这些边缘的方法因具体资源而异。

因此，我们可以确定每个边缘花费的时间，从而确定长期交易的原因，并帮助确定方法。

**4.5常用编程模型的示例图**

我们现在介绍一个从Android框架中的常规编程模式生成的示例图。这种模式和图4中的模式通常在实际使用中收集的踪迹中找到。请注意，我们的方法不限于这些编程模式。

WebView应用程序：WebView类是使用WebKit呈现网页的视图。它被应用程序开发人员广泛使用。图5显示了在应用程序启动后首次打开网页的用户事务的示例。请注意，为了演示目的，将一个执行间隔内的所有消息入队节点合并到先前的节点中。例如，消息1的出列队列消息2，并且因此具有到消息2的出队的出站边缘。在WebViewCore线程出队消息2之后，它（1）分叉渲染线程，该线程准备网页上的不同对象以进行渲染。准备之后，（2）它将消息排入WebViewCore线程，最终触发显示更新。

WebView相关应用程序的一个主要挑战是渲染线程是一个本机的基于任务的线程，我们不明确地显示。因此，为了推断每个执行间隔的终止，我们利用内核等待队列锁定原语。我们观察到，当渲染线程没有任务可以工作时，它在一个waitqueue上被阻塞，直到某个其他线程将一个任务放在队列中并通知它。因此，我们使用指示阻塞在队列上的内核事件来推断执行间隔的结束，并使用通知事件来推断生产者和消费者之间的因果关系。

**5.验证**

本节介绍了我们确定Panappticon正确识别用户感知转换并报告性能和能源开销的优缺点。 所有实验都在运行Android 4.1.2的Galaxy Nexus手机上完成。

5.1精度分析

我们通过评估十个应用程序，五个综合基准和五个开源应用程序来检查Panappticon的准确性（见表2）。 这些应用程序涵盖了我们已经识别的所有常见的编程模式，例如使用AsyncTaskor工作线程。 在这些测试中，我们希望（1）验证Panappticon是否正确地将每个UI输入链接到生成的显示更新，以及（2）确定提取的关系图是正确的。

对于第一个测试，我们手动测试应用程序来测量用户输入和生成的显示更新之间的延迟。 给定源代码，我们确定响应于用户输入接收，处理和更新显示的方法。 通过比较这些方法中记录的时间戳并由Papappticon测量，我们得出结论，Panappticon正确识别和链接所有十个应用程序的输入和显示更新，报告正确的事务延迟。