Panappticon: Event-based Tracing to Measure Mobile Application and Platform Performance

Lide Zhang* David R. Bild† Robert P. Dick† Z. Morley Mao† Peter Dinda⋆

* Facebook Inc.
Menlo Park, CA, USA
lidezhang@fb.com

† EECS Department,
University of Michigan
Ann Arbor, MI, USA
{drbild,zmao,dickrp}@umich.edu

⋆ EECS Department,
Northwestern University
Evanston, IL, USA
pdinda@northwestern.edu

ABSTRACT

Improving and optimizing user-perceived smartphone performance requires understanding device, system, and application behavior for real-world workloads. However, measuring such performance is challenging due to the multi-threaded, asynchronous programming paradigms used in modern applications and the multiple layers of hardware and software used to respond to user input events. We address this challenge with Panappticon, a lightweight, system-wide, fine-grained event tracing system for Android that automatically identifies critical execution paths in user transactions. Panappticon monitors the application, system, and kernel software layers and can identify performance problems stemming from application design flaws, underpowered hardware, and harmful interactions between apparently unrelated applications. We carried out a 14-user, one-month study of an Android smartphone system instrumented with Panappticon, which revealed a number of specific problems and areas for improvement that may be of interest to system designers, application developers, and device manufactures.

1. INTRODUCTION

Most mobile applications are interactive. Typically, an input from the user triggers a series of operations culminating in user-visible output, often an update to the display. User experience depends on perceived responsiveness [1], so controlling the latencies of such transactions is important for designers of applications, operating systems, and hardware platforms.

We define a *user-perceived transaction* as a series of operations started by the user's manipulation of the device (e.g., a screen touch or key press) and ended by a display update. Intuitively, the transaction captures the interval between the user instructing the device to do something and the expected result being displayed. Despite the importance of such transaction latencies to user experience, there is little existing development tool support to identify or determine the causes of slow transactions, leaving developers in the dark. This is mainly due to the asynchronous, multi-threaded nature of interactive applications. To keep the user interface (UI) responsive, applications must do lengthy or potentially blocking operations on background threads, complicating tracking of the execution flow of a single transaction. Analysis is further complicated by other unrelated applications or system processes running on the same device that may influence perceived performance. Therefore, analyzing the internal behavior of an application is not sufficient to fully characterize transaction latencies and causes. AppInsight [2] instruments application binaries to study user transactions. It identifies some performance problems but cannot explain poor performance resulting from indirect interaction among processes. We show an example of such a problem identified by our work in Section 6. All processes and applications that may influence each other should be considered.

In this work, we describe Panappticon[1], a system that identifies the user-perceived transactions of real-world users and can help diagnose the reasons for poor user-perceived performance. We illustrate its use with a 14-user, one-month study. Panappticon is useful to three types of people.

- Application developers can use Panappticon to identify inefficient application code and optimize accordingly. For example, in our study we found that Reddit News, a popular Android application, has many slow transactions due to CPU contention between its main thread and the non-critical system activities it triggers. Delaying the non-critical work until after the user transaction completes would improve user-perceived performance.
- Operating system designers can use Panappticon to optimize system policies. For example, in our study we found that the default DVFS (Dynamic Voltage and Frequency Scaling) governor included with Android nearly doubles the latency for transactions over 80 ms in duration, despite being designed for interactive workloads. An improved governor is needed and we have some suggestions on that topic.
- Hardware designers and manufacturers can use Panappticon to better understand the implications of architectural decisions for future devices, e.g., the relative speeds of different hardware components. We observe that typical non-gaming applications are not parallelized and do not benefit from multi-core processors, suggesting that designers should not neglect single-core performance.

Panappticon establishes causal relationships between user inputs and display updates by tracking execution flow among threads, through asynchronous calls, and across interprocess communication boundaries. This is achieved by instrumenting event handlers, asynchronous call interfaces, and the interprocess communication mechanisms in both user space and kernel space to log events from which the execution flow can be reconstructed. The system further logs resource usage information, including context switches and blocking on network interfaces and storage devices, to help identify the root causes of slow transactions.

* This work was performed while Lide Zhang was with the EECS Department, University of Michigan, Ann Arbor. It was supported in part by the National Science Foundation under award CNS-1059372.

CODES+ISSS'13, September 29–October 4, 2013, Montreal, Canada.

[1] A Panopticon is a type of building that allows a watchman to unobtrusively observe all occupants.
http://en.wikipedia.org/wiki/Panopticon

978-1-4799-1417-3/13/$31.00 ©2013 IEEE

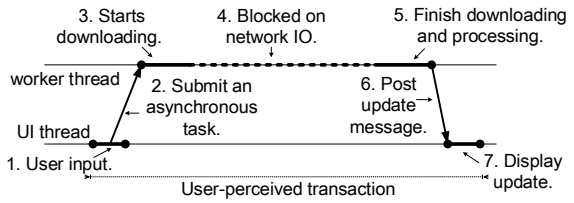


Figure 1: Illustrative example of a user-perceived transaction: the horizontal axis represents time and the vertical axis distinguishes threads.

We validated Panappticon on ten open-source applications, manually confirming that the detected user transactions and latencies were correct. Panappticon incurs an average 6.1% performance overhead and has unnoticeable impact on battery life.

This work makes three major contributions.

- We provide an unobtrusive methodology and design for extracting user-perceived transaction latencies based on causal relationships among the operations triggered by an input.
- We describe Panappticon, an open-source[2] system that applies the above methodology to automatically characterize user-perceived transactions and provide detailed resource usage information, allowing for root-cause diagnosis of the causes of slow transactions.
- We present results of a real-world user study of 14 Android users running Panappticon on their phones for one month and provide three case studies showing how Panappticon can benefit application developers, system developers, and smartphone manufactures. The data traces are publicly available[2].

2. GOALS AND DESIGN CHALLENGES

In this section, we give the formal definition of our goal: making characterization of user-perceived transactions easy. We also give an example to illustrate the use of Panappticon and explain the challenges we faced during its design.

2.1 Design goal and example

We define a *user-perceived transaction* to be a series of operations started by a user's input to a computer system, e.g., a screen touch or button press, and ending with a display update. Generalizing the concept and implementation to other outputs, such as the audio device, would not pose major challenges. The latency between the UI input and the update captures the latency perceived by users. That is, any operation not included in the user-perceived transaction does not influence the perceived latency and therefore does not directly impact user experience. Some UI inputs can trigger multiple display updates; we define the last display update as the end of the transaction.

Figure 1 depicts a user-perceived transaction. Imagine a simple application that downloads and displays celebrity quotes. As shown in Figure 1, the transaction is started by a button press. In the UI event handler triggered by this input, the main UI thread submits an asynchronous task to a worker thread to download the quote. During execution of the asynchronous task, the worker thread may block while waiting for packets from the network. After the download finishes, the worker thread posts a message back to the UI thread to display the quote. In this example, the operations between the user input and display update form one user-perceived transaction.

Panappticon seeks to identify such user-perceived transactions and determines the performance bottlenecks for each. To achieve this goal in the preceding example, the system must record the user input and, based on the asynchronous call, establish a causal relationship between the UI event handling and the worker thread execution. Then, the system must record the network block and, finally, link the worker thread execution to the UI update by tracking the posted message.

To optimize user-perceived transaction latency, we need to identify the *critical path* for each transaction and understand the dominant components. The critical path is the bottleneck execution path whose length captures the perceived latency between user input and display update. The nodes on the critical path are responsible for delay; increasing their execution times would increase the latency between input and update. In the example above, the path between the input and update nodes through the worker thread represents the critical path, with the network responsible for most of the delay.

2.2 Design challenges

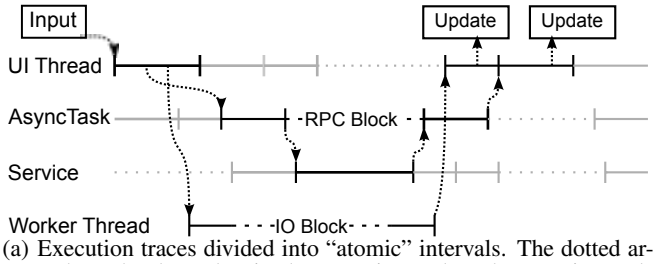
We faced the following major challenges in the design of Panappticon.

- *A user transaction can involve execution across threads and process boundaries.* For example, in the preceding example, the main thread submits work for asynchronous processing by a background thread, a common Android programming pattern as explained in Section 4. Even more challenging, some applications contain two or more independent processes, one running the UI and the others doing work asynchronously with all actively involved in the user-perceived transaction. Therefore, we must track asynchronous (and synchronous) calls across threads and process boundaries.
- *Display updates are not always caused by the most recent user input event.* In the preceding example, it is possible that other display updates are triggered by a change in system state (e.g., a change in network connectivity) between the input event and display update. In a more complicated application, a second user transaction might start before the first finishes. Both scenarios result in display updates being separated from the user input events that caused them by other user input events. This prevents one from grouping display updates with their most recent input events. Panappticon must explicitly track causal relationships among operations.
- *It is necessary to know the underlying hardware state.* One of our goals is to help system and application developers determine the reasons for lengthy transactions. Possible causes include contention for the processor, long blocking times on network or disk IO, or problems with system policies such as DVFS. For instance, in our running example, if the network blocking time is reduced, the user-perceived latency would also be reduced. To achieve this goal, we must efficiently log fine-grained resource usage information, such as IO blocking times and context switches.
- *Mobile devices are resource constrained.* The system must have low performance and energy overheads so as to not influence user experience. Limited smartphone CPU throughput, memory, and energy capacities prevent logging of non-essential data and relegate user transaction analysis to the server.

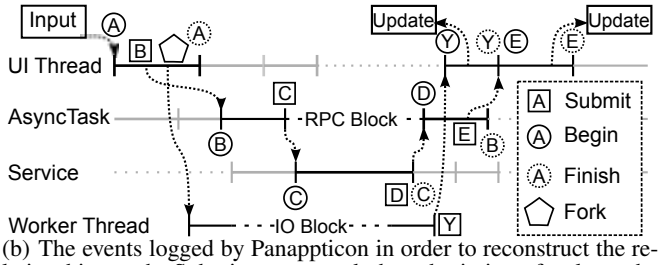
3. APPROACH OVERVIEW

We developed and implemented an event-based tracing infrastructure that can efficiently capture (1) the relationships between operations to identify user transactions across threads or processes and (2) which resource (e.g., CPU, network, disk, etc.) a thread is

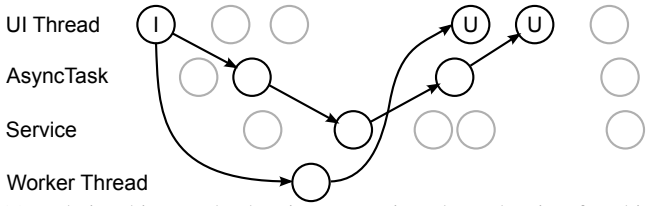
[2]http://ziyang.eecs.umich.edu/projects/panappticon/



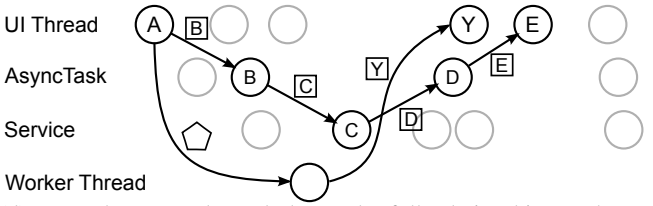
(a) Execution traces divided into "atomic" intervals. The dotted arrows show the dependencies between intervals. The grey intervals are unrelated to this user transaction and the dashed portions indicate the thread is idle.



(b) The events logged by Panappticon in order to reconstruct the relationship graph. Submit events mark the submission of tasks to the UI, AsycTask, and Service threads, begin and finish events demarcate the processing of each task, and a fork event marks the creation of the worker thread.



(c) Relationship graph showing execution dependencies for this transaction. Each node represents one execution interval, with an incoming edge indicating the interval was triggered by the preceding node. Given this graph, 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 execution 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 user-perceived 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 service 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.

using or blocking on at each instant to reveal performance bottlenecks. This section describes our methodology, shown in Figure 2.

3.1 Methodology overview

Our technique for tracking user transactions across threads and processes is based on identifying and linking "atomic" intervals of thread execution. Such intervals represent work that happens contiguously, e.g., a worker thread processing one task from a task queue. Intervals processing the same transaction are casually dependent—one interval triggers the execution of the next. Figure 2(a) shows an example execution trace divided into intervals, with arrows indicating the causal relationships. Figure 2(c) shows the relationship graph we wish to obtain.

Extracting the relationship graph requires (1) separating each execution trace into "atomic" intervals, (2) identifying the causal relationships between intervals, and (3) identifying the initial (e.g., user input) and terminal (e.g., display update) intervals of user-perceivable transactions. We log events sufficient for each task, as illustrated in Figure 2(b). For the first two tasks, we instrument the Android platform and kernel to record events for popular programming paradigms. For the third, we instrument the platform to record user input and display updates, tagging the current interval. These allow graph construction, as shown in Figure 2(d).

To determine the causes of latency, we record events indicating resource use, e.g., context switches for CPU and blocking times for disk and network access. Section 4 describes all captured events.

3.2 Architecture overview

Panappticon contains the five major components shown in Figure 3: a userspace logger, a kernelspace logger, an event collector, a server-side collector, and a user transaction analyzer.

Userspace logger and kernelspace logger: The userspace and

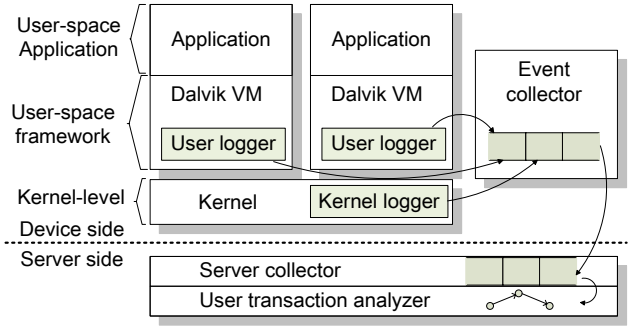


Figure 3: System architecture overview.

kernelspace loggers record the events mentioned in the preceding section. Specifically, input events, display update events, and most events indicating causal relationships are captured in userspace, where the use of high-level programming paradigms make event-based inference easier. The kernel logger captures resource utilization events and some events indicating causal relationships across process boundaries, e.g., forking and IPC transactions. Section 4 enumerates all the events captured by these two loggers. To minimize the performance impacts of these loggers, both buffer event records in memory, sending them to the collector in batch when the buffer is full.

Event collector and server collector: The event collector is responsible transmitting traces from the loggers to the server-side collector for processing. To minimize performance and energy overheads and avoid losing data, the logs are uploaded in batch only when WiFi is available. Failed transmissions are buffered to the SDcard and retried later.

User transaction analyzer: The user transaction analyzer ex-

tracts the relationship graphs using the method described in Section 3.1. From these graphs, it extracts user transactions and their corresponding user-perceived latencies and resource usages. Section 4 describes the graph extraction process.

4. INSTRUMENTATION DETAILS

Before Panappticon's design could be determined, it was necessary to answer the following questions. (1) What data need to be captured? They should be sufficient to identify each user transaction and its resource use with low overhead. (2) How should these data be used to construct the relationship graph to link user input with display update? (3) How should resource accounting be done for each transaction? This section answers these questions.

Panappticon is based on Android 4.1.2. Some of the implementation details below are specific to this Android version.

4.1 Background: Android

Android is a Linux-based operating system developed by Google for mobile devices such as smartphones and tablets. This subsection summarizes Android properties to provide background useful for understanding Panappticon.

Dedicated UI handling: Applications on Android are UI-centric. All UI related events, including serving user interactions and updating the display, are handled on one dedicated thread, which is also the main thread of each application. To maintain a responsive UI, developers should avoid lengthy operations, or blocking of the UI thread [3]. Instead, they should create separate worker threads. This property motivates us to track asynchronous calls across threads.

Looper thread: Most Java threads managed by the Android system, including the main thread of each application used for UI events, use a message queue model. Messages are placed in a queue and the thread loops indefinitely, processing messages from the head of the queue. In Android, such threads are called Looper threads and share a common implementation of the message queue and looper functionality. The shared code facilitates easy logging of message submission and execution events.

4.2 What information do we capture?

There is a trade off between the amount of information Panappticon gathers and its overhead. For example, a complete trace of every method in the system framework that applications could potentially call during a transaction would allow us to have full knowledge of the system status. However, this would slow down the application by orders of magnitude. To this end, we record the minimum amount of information that is necessary to identify user-perceived transactions and their performance bottlenecks. This information can be placed in the following categories: (1) user interaction events including screen touch and key press, (2) causality between asynchronous calls and callbacks within and across threads, (3) inter-process communication between threads and processes, (4) various thread synchronization mechanisms and the causality between threads due to synchronization, (5) resource accounting for each thread such as context switching and blocking on network and disk IO, (6) other causality relationships between threads, e.g., forking a thread, (7) display update, and (8) additional information that helps to track foreground applications and application names. Records for each event contain the following fields: `Timestamp, Event_type, TID, Data, (CPU_core)`. `CPU_core` specifies the core associated with an event and is only available for kernel events. Table 1 summarizes the events Panappticon logs. We now explain each type of event.

User input: We record input events due to screen or button touches and those from the software keyboard. In Android, the first type of input is dispatched directly to the foreground application through the onInput() callback method in the View class. Unlike the first type of input, software keyboard input is dispatched to the foreground applications through another systemUI application, which translates screen touch events into keyboard inputs.

Asynchronous calls and callbacks: As we mentioned in Section 4.1, the UI-centric nature of Android applications requires developers to use asynchronous worker threads for lengthy operations. To achieve this, there are two common programming models in Android: (1) start a worker thread and post a message to the UI thread to update the display after the work is done or (2) submit a task to the pool of thread executors.

To handle the first case, we instrument the MessageQueue class in the Android framework library to capture the causal relationships between message enqueue and dequeue events. Each message is associated with a unique ID. For the second case, we instrument the ThreadPoolExecutor class in the generic Java library to record the causal relationship between task submission and consumption. These events are matched using task ID.

Inter-process communication: As mentioned above, Android uses a kernel-level inter-process communication implementation called Binder for RPC. For each process, Binder manages a pool of threads to execute incoming RPC requests. For each call or Binder *transaction*, we log the full RPC call by tracking events across process boundaries.

Synchronization mechanisms: Contention for virtual resources (worker threads, shared data segments, etc.) can result in slow transactions. Access to such resources is usually mediated by synchronization primitives, so we log contested accesses to the following in-kernel primitives: waitqueues, semaphores, mutexes, and futexes. Specifically, we log when a thread blocks waiting for access, when it resumes from that block, and, after releasing a primitive, which waiting threads are awakened. We do not log lock, unlock, or spinlock events due to the volume of accesses—contested accesses are much rarer.

Resource accounting: To help determine the bottlenecks for each transaction, we log access to the three main time-shared resources used by Android applications: processor, network, and disk. For processors, we log each context switch, including the incoming thread ID, the outgoing thread ID, and the new state of the old thread (still runnable, interruptible sleep, etc.). For the disk and network, we log when a thread blocks on a read request and then when it resumes.

Display update: Android provides developers two main paths to update the display: the View class in the framework library and OpenGL to render display directly. Panappticon currently considers only the first path because OpenGL is mainly used for graphics rendering in gaming applications. The concept of user transaction in such gaming application is different from our definition in interactive applications due to the use of animation, as discussed in Section 5.

Additional information: In addition to all the information necessary to capture causality between events, we also collected additional information to help us better understand the context of the transactions. For example, we recorded the application that enters and exits the foreground to distinguish applications users interact with from system applications. This is done via changes to the Activity framework class. Similarly, the kernel records the name of each thread.

4.3 How do we construct relationship graph?

To identify user transactions based on event streams, we con-

Table 1: List of Events Captured

Category	Event type	Space	Data field	Description
User inputs	UI_INPUT	User	null	User's input on the touch screen or hardware button
	UI_KEY	User		User manipulation of the software keyboard
Async callbacks	ENQUEUE_MSG	User	message_id, queue_id time_to_dequeue	Enqueues and dequeues a message with message_id on queue with queue_id
	DEQUEQUE_MSG			
	SUBMIT_ASYNC_TASK	User	task_id	Submits/consumes a task to one of the pooled thread executors
	CONSUME_ASYNC_TASK			
IPC calls	BINDER_PRODUCE_ONEWAY	Kernel	transaction_id	Caller sends arguments to the remote w/o blocking
	BINDER_PRODUCE_TWOWAY			Caller sends arguments to the remote and blocks
	BINDER_REPLY			Remote thread sends return value back to the caller
	BINDER_CONSUME			A remote thread begins execution
Locks: mutex, semaphore, futex, and waitqueue	*type*_WAIT	Kernel	lock_id	Block waiting for access to lock
	*type*_WAKE		lock_id	Resume from block waiting for access to lock
	*type*_NOTIFY		lock_id, notify_pid	Notify waiting thread to wakeup
Resource accounting	CONTEXT_SWITCH	Kernel	old_pid, new_pid	Context switch from process old_pid to new_pid
	SOCK_BLOCK/RESUME	Kernel	null	Blocks and resumes on socket waiting for connection
	DATAGRAM_BLOCK/RESUME	Kernel	null	Blocks and resumes on socket waiting for UDP data
	STREAM_BLOCK/RESUME	Kernel	null	Blocks and resumes on socket waiting for TCP data
	IO_BLOCK/RESUME	Kernel	null	Blocks and resumes on disk IO
Other dependency	FORK	Kernel	parent_pid, child_pid	Parent process forks child process
Display update	UI_INVALIDATE	User	null	Invalidates the view, schedules display update
	UI_UPDATE	User	null	Redraw the view
Additional information	THREAD_NAME	Kernel	t_pid, t_name	Thread t_pid has name t_name
	ENTER/EXIT_FOREGROUND	User	null	The current PID enters/exits being foreground app

struct directed acyclic graphs based on the relationships between events. Each event entry in the trace is identified as a node in the graph. The edges represent the relationships between pairs of events.

Figure 4 illustrates the relationship graph constructed based on the example trace in Listing 1. The map between node name and event is also shown in the trace. The logic of the trace is described in Section 2.

We identify two types of relationships: causal relationship and temporal ordering.

Listing 1: Example trace of correlation graph

```
(I)    USER_INPUT              pid:0
(EM1)  ENQUEUE_MSG             pid:0  message_id:1
(DM1)  DEQUEUE_MSG             pid:0  message_id:1
(ST1)  SUBMIT_ASYNCTASK        pid:0      task_id:1
(CT1)  CONSUME_ASYNCTASK       pid:1      task_id:1
(B)    SOCK_BLOCK              pid:1
(R)    SOCK_RESUME             pid:1
(EM2)  ENQUEUE_MSG             pid:1  message_id:2
(DM2)  DEQUEUE_MSG             pid:0  message_id:2
(INV)  UI_INVALIDATE           pid:0
(UP)   UI_UPDATE               pid:0
```

Causal relationship: A relationship between two execution intervals where the earlier interval triggers the latter one, as explained in Section 3.1. It is represented by solid edges in the graph. For example, the message enqueue event triggers the message dequeue action. In other words, without the message enqueue operation, the message dequeue operation would not exist. We identified the following node pairs correlated with causality.

- ENQUEUE_MSG and DEQUEUE_MSG with the same message_id.
- SUBMIT_ASYNCTASK and CONSUME_ASYNCTASK with the same task_id.
- BINDER_PRODUCE or BINDER_REPLY events and BINDER_CONSUME with the same transaction_id.
- Two nodes created by the FORK event: the node representing FORK event on the parent thread and the node representing the initial execution on the child thread.
- UI_INVALIDATE and its closest UI_UPDATE within the same thread.

Temporal ordering: A relationship between events within an execution interval. It is represented by a dotted edge. For instance, in the previous example, message 1 is enqueued while the callback method triggered by the input is executed. Similarly, an asynchronous task is submitted during the execution of the dequeued message 1.

One major challenge of temporal ordering is to determine when to end an execution interval. If this is not correctly determined, all events on the same thread will be spuriously connected, causing user transactions to be mistakenly grouped. We place the threads in Android applications in two categories and we use different approaches for each type.

- Task-based threads are the most common background thread pattern in Android. These threads consume tasks from a queue and block when the queue is empty. The end of each task indicates the termination of an execution interval. For example, the main UI thread in each application is a Looper thread that waits for incoming messages and processes them. All the events happening when processing one message belong to one execution interval. The same approach applies to Binder threads and asynchronous task threads, which wait for new transactions and new tasks. We do not explicitly instrument other background threads. Instead, events from the locking primitives indicate the producer/consumer of a task queue, allowing inference of execution intervals. The application of this approach to applications using WebKit is explained in Section 4.5.
- Worker threads may also be forked for one-time execution. Our approach automatically infers the execution interval in this case.

We assume that no unrelated work is performed while processing a particular task from a queue. A (probably misguided) programmer might have the handler for a particular message check an unrelated condition and enqueue a message to handle it, leading to a false dependency. In practice, this should be rare because the Android APIs encourage good practices to maintain a responsive

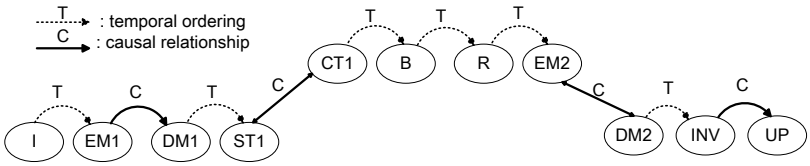


Figure 4: Illustrative example of relationship graph.

UI thread. None of the applications manually inspected to validate Panappticon exhibited such false dependencies. Further, only (rare) false dependencies leading to display updates would impact critical path analysis.

Using the methodology above, we can infer separate user transactions when they overlap. By further extracting the critical paths from UI_INPUT and UI_UPDATE in each transaction, we can derive the latency of each user-perceived transaction.

4.4 Resource accounting

The major research questions we use resource accounting to answer are, "what are the causes of delay for transactions with noticeable delay?" and "what can be done to speed them up?" To this end, we analyze the resource use on the critical path for each transaction in two steps.

First, we add the resource accounting-related kernel events into the correlation graph to indicate the use of physical resources. In particular, context switch events allow per-thread accounting of CPU use. Similarly, network and disk IO access events allow per-thread accounting of network and disk use. In additional to physical resources, we also add synchronization events into the graph to represent the use of virtual resources. For example, a thread can block on a mutex and wait for other threads to release it. Note that these events are added to the graph based on temporal correlation.

Second, we analyzed the edges on the critical path to understand the reason for its latency. All edges fall in the following categories.

- Edges that indicate the corresponding thread is running and occupying the processor. For example, edges between any events that occur between two context switches. Latency due to this type of edge depends on processor speed.

- Edges that suggest the corresponding thread is blocked, waiting for some resource. We further place different resources in the following categories. (1) Physical resources such as network, IO, and CPU. Waiting for CPU means the thread gets context switched out when it is still eligible to run. (2) Virtual resources such as locks or thread execution. For example, the latency between submission and consumption of an asynchronous task can be due to the worker threads being occupied and therefore blocking the consumption of the new task. Approaches that shorten these edges vary based on specific resource.

Thus, we can determine the time spent on each edge, allowing the causes of long transactions to be determined and helping to identify solutions.

4.5 Example graph of common programming models

We now present an example graph generated from common programming patterns in the Android framework. Both this pattern and that in Figure 4 are commonly found in the traces gathered in real-world use. Note that our approach is not limited to these programming patterns.

WebView applications: The WebView class is a view that renders web pages using WebKit. It is extensively used by application developers. Figure 5 shows an example of a user transaction that loads a webpage the first time after an application launches. Note

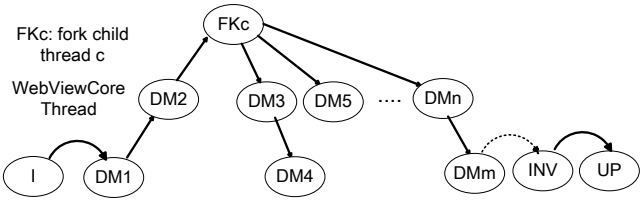


Figure 5: Example trace of application using WebView.

that for demonstration purpose, all the message enqueue nodes within one execution interval are merged into the prior node. For example, the dequeue of message 1 enqueues message 2 and hence has an outgoing edge to the dequeue of message 2. After the WebViewCore thread dequeues message 2, it (1) forks the rendering thread, which prepares different objects on the webpage for rendering. After preparation, (2) it enqueues a message to the WebViewCore thread, which eventually triggers the display to update.

One major challenge we have for WebView-related applications is that the rendering thread is a native task-based thread that we do not instrument explicitly. Therefore, to infer the termination of each execution interval, we leverage the kernel waitqueue locking primitives. We observed that when the rendering thread does not have a task to work on, it is blocked on a waitqueue until some other thread puts a task in the queue and notifies it again. Accordingly, we use the kernel event indicating blocking on the queue to infer the end of execution interval and use the notify event to infer the causal relationship between producer and consumer threads.

5. VALIDATION

This section describes our efforts to determine whether Panappticon correctly identifies user-perceived transactions and reports its performance and energy overheads. All experiments are done on Galaxy Nexus phones running Android 4.1.2.

5.1 Accuracy analysis

We examine the accuracy of Panappticon by evaluating ten applications, five synthetic benchmarks, and five open-source applications (see Table 2). These applications cover all the common programming patterns we have identified, e.g., using AsyncTask or worker threads. In these tests, we wish to (1) verify that Panappticon correctly links each UI input to the resulting display updates and (2) confirm that the extracted relationship graphs are correct.

For the first test, we manually instrumented the applications to measure the latencies between user input and the resulting display update. Given the source code, we identify the methods that receive, process, and update the display in response to user input. By comparing the timestamps recorded in these methods and measured by Panappticon, we concluded that Panappticon correctly identified and linked the inputs and display updates for all ten applications, reporting the correct transaction latencies.

For the second test, we compared visualizations of the generated transaction relationship graphs with our understanding of the source code. For example, we knew from studying the source if an application used an asynchronous task or RPC call to perform part of the transaction. The generated graphs were consistent with

Table 2: List of Applications

Type	Name	Description
Synthetic	Async Task	Starts an AsyncTask after button press and updates the display
	Worker Thread	Forks a worker thread after button press and updates the display
	Service	Starts a remote service, make an IPC call, and updates the display
	WebView	Loads a webpage using standard Android API
	Animation	Starts an AsyncTask while displays the loading animation, terminates the animation after the task is done
Open source	CrossWords	Loads a cross word game, displays solution based on user inputs
	ReadForSpeed	Downloads text and displays it based on timer
	Android browser	The default browser on Android
	K9 mail	Mail client
	NPR news	News reading application

our expectations, indicating that Panappticon extracted the correct graphs.

Although Panappticon performs well on our intended workloads, it cannot handle all applications and behaviors. We describe some of its limitations here.

- **Approach limitation:** Panappticon does not track data or control dependencies directly, but relies on instrumented system or platform libraries that provide support for high-level programming paradigms like task-queues and semaphores. Applications that implement their own coordination primitives or use lockless synchronization cannot be tracked by Panappticon. Tracking data dependencies requires techniques like taint tracking [4] that incur overheads too high for online use.

Our definition of user-perceived transaction is not appropriate for animated applications like games. Our definition tries to capture the time a user spends waiting for an expected result from the input. Animated results manifest over many frames and may be modified by later inputs. As a result, we exclude animated transactions (like most games) from the experiment, as explained in Section 6. Note that according to a recent study, games account for 15% of published applications [5].

- **Instrumentation limitation:** Our specific implementation for Android assumes that background worker threads are task-driven. We modified the common Android-provided task-driven primitives like the Looper, AsyncTask, and Executor classes to record the start and end of each task. While this assumption holds true for most interactive applications, it can miss work done on native, non-Java background threads.

One particular example, the WebKit library used to display web pages, is quite prevalent so we use kernel primitives to infer its task intervals as described in Section 4. WebKit is the only such library we found in our traces, but other less-frequently used libraries might exist. Panappticon would miss them unless the implementation were extended.

5.2 Overhead analysis

We now report the performance and energy consumption overhead of Panappticon.

Performance: The performance overhead of Panappticon stems mainly from the CPU cycles used to log each event and the memory used to store the logs (reducing memory available for the Linux file cache and Android application cache). The CPU overhead is minimized by eliminating locking in the logging path (e.g., by using per-CPU log buffers in the kernel) and memory overhead is minimized by fixing the buffer size to 30 MB in the kernel and 15 MB in userspace.

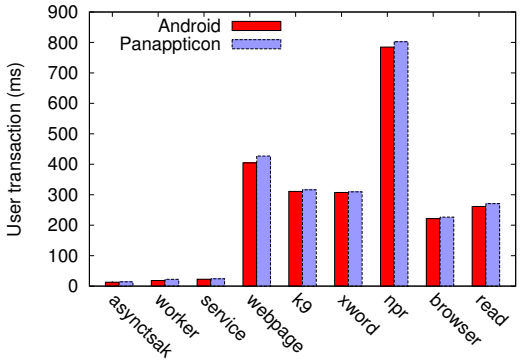


Figure 6: Overhead of Panappticon.

To evaluate the overhead, we compared user transaction latencies on a system with Panappticon to one without. To determine latencies on the system without Panappticon, we manually instrumented several open source applications to directly record transaction lengths. The experiment was conducted on two Galaxy Nexus phones. As shown in Figure 6, the average overhead of Panappticon is 6.1%.

Battery: The energy consumption of Panappticon is mainly due to uploading the logs to our server (300 MB/day for our heaviest user). When WiFi is not available, Panappticon saves data to non-volatile storage and defers the transmission until the phone is charging and WiFi is available. No users reported a noticeable change in battery life.

6. CASE STUDIES FROM REAL-WORLD TRACES

We now present the findings of our user study. We first explain the study design and then proceed through three case studies of real-world traces, showing three specific findings uncovered by Panappticon that may interest application developers, system designers, and smartphone manufacturers.

6.1 Experimental setup

14 students from the University of Michigan volunteered to run Panappticon on their smartphones. We selected only regular, long-time smartphone users in the interest of observing representative smartphone user behavior. Galaxy Nexus phones [6] were used, which have 1.2 GHz dual-core processors.

The study had two major goals. (1) Identify the causes of transactions of noticeable length (i.e., >50–100 ms). This goal can be achieved by analyzing the resource usage for each transaction. (2) Understand the impact of architectural differences on user transactions, e.g., changing the number of CPU cores and the impact of the DVFS (Dynamic Voltage and Frequency Scaling) policy. To achieve this, we periodically disabled one core on the device and changed enabled/disabled DVFS. Specifically, we ran a daemon that changed the number of cores and DVFS status every 10 minutes, randomly switching between the four possible configurations. To avoid degrading user experience during intensive workloads, the configuration was changed only when CPU utilization was below 5%.

Table 3 provides basic information about the deployment. We detected 104,588 transactions in total. Among them, 88,656 trans-

Table 3: Deployment Statistics

Start date	Oct. 31, 2012
Finish date	Nov. 30, 2012
Application count	189
Total transactions	104,588
Without animation	88,656
With animation	15,932

Table 4: Resource Accounting Statistics for Sample Transactions from Reddit News

Total latency(s)	Network block(s)	IO block(s)	Waiting for CPU(s)
3.78	0.98	0	1.39
2.35	0.42	0.02	0.93
1.54	0.23	0	0.89
1.27	0.15	0	0.33

actions do not involve animation. Figure 8 presents the cumulative distribution of latencies for these transactions. Transactions without animation last at most 38.60 seconds with only 2% of transactions lasting longer than 1 second. As explained in Section 5, we focus on non-animation transactions in the next subsection because the lengths of animation transactions are related to user satisfaction in a complex way.

6.2 Case study one: analysis of long transactions for applications

One major goal of Panappticon is to help application developers identify user transactions that may be noticeably and annoyingly long and help expose potential fixes. We now describe how Panappticon helped identify a performance inefficiency in Reddit News, a popular application for browsing the website reddit.com.

Reddit News is a popular closed-source application on the Android Market that has been downloaded millions of times. Panappticon revealed that its produces many transactions longer than one second. Table 4 shows the resource accounting statistics generated from Panappticon for four example transactions. "Network" and "IO" block columns show the time spent blocked waiting for those resources on the critical path. "Waiting for CPU" shows the time spent waiting for the CPU while preempted. The rest of the transaction is spent running on the CPU.

The time spent in preemption is the dominant reason for the high latency and suggests heavy CPU contention during transactions. A deeper look at the critical paths of these transactions reveals two things: (1) the preempting threads that consume the most CPU time during these transactions are the system threads responsible for writing to the emulated SD card and (2) the preemption is triggered by the Reddit News thread after each network block. Longer network blocks trigger longer preemption times. Figure 7 shows one such transaction trace and the CPU usage from the contending system thread. We believe the Reddit News thread is fetching images from the network and caching them to disk immediately after download.

Although saving to the SD card does not block the thread directly, the system thread in charge of SD card writing causes heavy CPU contention with the thread on the critical path. Why does the thread writing to the SD card start working during a user transaction? A deeper investigation into the writing policy reveals that although the write back activity happens asynchronously with the write system call, it is triggered every time the write buffer fills. The images downloaded by Reddit News range from 15KB to 3MB while the default buffer size is 8KB. This means that every time an image is downloaded, the SD card thread immediately begins writing, causing resource contention with the critical path thread. We do not have access to the source code of the SD card driver, but hypothesize that the intensive CPU load results from driver inefficiencies. Others have noted similar effects [7].

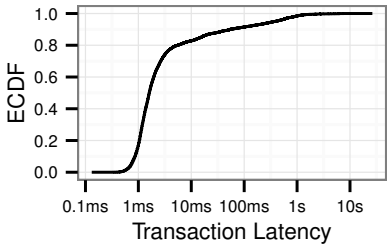


Figure 8: Distribution of non-animation transaction latencies.

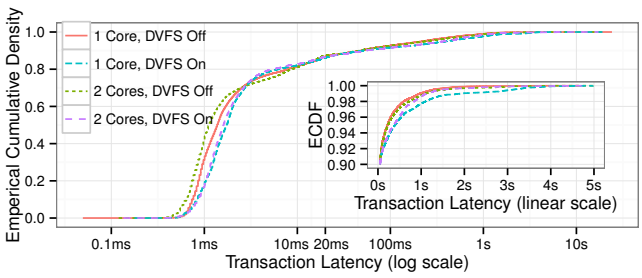


Figure 9: Distributions of user transaction latency for different core count and DVFS configurations. The zoomed-in inset plot highlights the differences in the upper percentiles.

Two potential solutions would reduce contention and thus improve user-perceived latency. The first, making the driver code faster, would address the root cause, but is not an option for application developers and may be intractable even for the driver developers. The second is to defer image caching until after display.

6.3 Case study two: impact of DVFS policy on user transaction latency

The second case study illustrates how Panappticon can help system designers by determining the impact on user transaction length of a specific system setting, activation or deactivation of dynamic voltage and frequency scaling (DVFS). DVFS attempts to reduce energy consumption by reducing processor voltage and frequency to the lowest point still providing the needed throughput. This strategy often hurts low-throughput, latency-sensitive tasks that do not trigger the faster DFVS states. The Android DVFS policy, *interactive*, tries to address this issue, but as we show, only partially succeeds.

Figure 9 shows the empirical distributions of user transaction latencies for four configurations: DVFS on (the *interactive governor*) and off (the highest frequency, 1.2 GHz) with one and two cores available. In both cases, transaction latencies are higher with DVFS enabled. The difference is negligible for short transactions ($<$ 20 ms), but significant for longer transactions. In dual-core mode, the difference is 170 ms at the 96th-percentile and for single-core, 517 ms at the 98th. DVFS negatively and noticeably impacts user transaction latency.

A close look at the *interactive* governor policy reveals the cause. The policy matches the CPU frequency to the utilization over the prior 20 ms, but includes three optimizations for latency-sensitive tasks. First, the frequency is boosted to 700 MHz on each user input, e.g., touchscreen press. Second, it is similarly boosted if the utilization is over 85% in the first 20 ms after leaving the CPU idle state. Third, a frequency is held at an increased level for at least

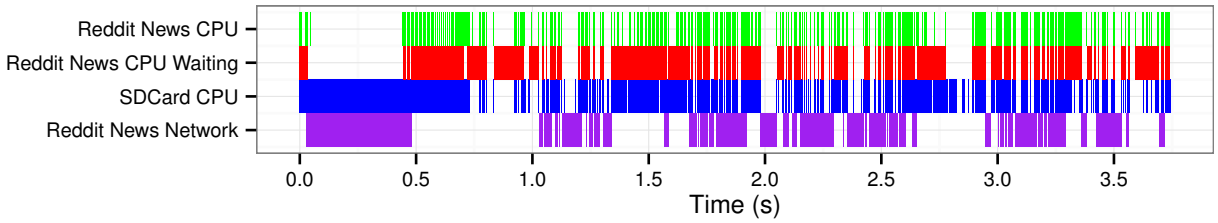


Figure 7: Transaction trace for Reddit News showing the main thread contending for the CPU with system-owned threads used to control the emulated SD card. For the Reddit News thread, time spent using the CPU is shown in the first row and time waiting for the CPU in the second row. Waiting until after the display is updated to cache the downloaded data to the SD card would reduce this contention and shorten the user-perceived transaction latency.

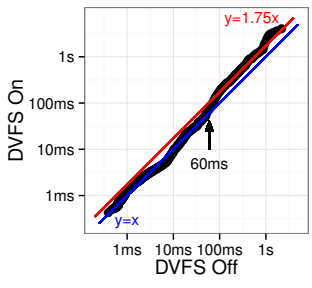


Figure 10: Q–Q plot [8] comparing latency distributions with DVFS on and off for a single core. DVFS has little impact on short transactions (< 60 ms), but hurts longer transactions by up to 1.75×. This is because the *interactive* governor allows the frequency to drop 60 ms after the user input event.

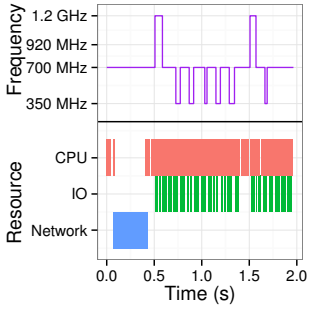


Figure 11: Transaction illustrating the reason for the poor behavior of the DVFS policy. The CPU use is interleaved with disk blocks and thus although the transaction includes significant CPU time, the utilization is low and the DVFS policy keeps the CPU frequency well below maximum.

60 ms before being lowered. We thus hypothesize that short transactions are slower because the initial boost is to 700 MHz, not the full 1.2 GHz. Longer transactions are slower because the frequency often drops after 60 ms.

The Q–Q plot [8] comparing the distributions (Figure 10) supports this hypothesis. Below 60 ms, the distributions are similar (DVFS off is slightly faster below 10 ms), but above, DVFS is much slower, with latencies averaging 1.75× higher.

As shown in Figure 11, lengthy user transactions include periods of blocking on network and disk interleaved with CPU use. Thus, despite significant CPU use, the utilization is low and after 60 ms, the *interactive* governor drops the frequency. The transaction finishes at the lower speed, increasing latency. For workloads with substantial time spent blocking on network and disk, CPU utilization is a poor metric for frequency control.

Our goal is demonstrating the value of Panappticon, not fixing the *interactive* DVFS policy. However, we offer two possible solutions for future investigation. First, increase the default 60 ms timeout for dropping the frequency. This is simple (that parameter is already configurable), but would hurt energy consumption. Second, include the time blocked on disk or network when computing CPU utilization for the purpose of DVFS control. This keeps the utilization, and thus frequency, high while tasks are in progress. Of course, the idea is incomplete—false positives due to, for example, intentionally blocked network threads would hurt energy efficiency.

One could argue this behavior is correct: that DVFS policies intentionally trade performance for energy. However, our results show that CPU energy consumption is dominated by other components, so decreasing transaction latencies reduces *overall* energy consumption. Table 5 shows the power consumption for different frequency states, measured on a Galaxy Nexus running a synthetic workload to maintain 100% utilization. CPU energy efficiency increases with decreasing frequency, but overall system energy efficiency decreases. In summary, for current processors DVFS will generally improve energy efficiency for IO- or memory-bound workloads, but hurts for CPU-bound tasks, such as gaming. A DVFS policy taking this into account could improve both energy efficiency and user-perceived latency.

Table 5: Power and Energy Consumption for Different Frequency Levels for Galaxy Nexus

Frequency	CPU Power (mW)	Total Power (mW)	Normalized est energy (X)
350	220	820	2.64
700	610	1260	1.46
920	1000	1650	1.16
1200	1600	2260	1

6.4 Case study three: impact of hardware resource on user transactions

The third and final case study shows how Panappticon can be used to study the impacts of hardware design choices on user transaction latencies. To this end, we consider the following question: how does CPU core count influence the latencies of user transactions?

To answer the question, we compare the distributions of user transaction latencies for single-core and dual-core configurations. To eliminate the effect of DVFS, we consider data gathered with DVFS disabled. As shown in Figure 9, for short transactions (below 2 ms), an additional core reduces transaction time. However, for longer transactions, the additional core does not significantly change transaction time, on average. This suggests that longer transactions are typically not-parallelized, CPU-bound workloads. OpenGL applications such as games were not studied, and might yield different results.

This finding indicates that many applications, and almost all of those we encountered, are not designed for efficient parallel execution, even when the transactions latencies are high enough to justify it. Although the applications use multiple threads, usually only a single thread is active. Determining if the applications are parallelizable is beyond the scope of Panappticon, but we can offer two suggestions.

First, Panappticon can identify applications with slow, CPU-bound, and non-parallelized execution. Developers of these applications should consider parallelizing them. With Panappticon, it is easy to focus on the sections of code that would provide the most benefit if parallelized.

Second, hardware manufacturers should continue to optimize for

single-threaded execution. Many are already doing this, for example, allowing additional cores to be turned off when not in use or providing a single core that operates at higher speed when the others are turned off. Panappticon has shown that these hardware/OS features remain appropriate for many real user transactions, and can be used to quantify their impact.

7. RELATED WORK

This section summarizes other work targeting similar problems: monitoring and debugging of user transitions and the characterization of mobile application performance in real-world use.

Performance monitoring and debugging for user transactions or request: Prior work on performance monitoring and debugging of user transactions falls into the two following categories.

The first category uses developers' knowledge of application semantics to identify transactions [9, 10]. Magpie [9] characterizes the individual requests handled by Windows-based servers. Developer-provided event semantics are used to join the logged events into transactions. In contrast, Panappticon does not require such semantics, reducing developer burden. Further, our focus on mobile platforms, not servers, leads to a different system design.

LagHunter [10] is a debugging tool that identifies perceivable performance bugs. Developers identify *landmark methods*, usually those handling UI inputs, for special instrumentation by which the call stacks are tracked. This approach is unsuitable for multi-threaded, asynchronous systems like Android, because it only allows tracking of synchronized UI event handling.

In contrast, the second category does not require developer input. AppInsight [2] instruments application binaries to identify the critical execution path in user transactions. It uses causality between cross-thread work units to trace execution, as does Panappticon. However, it does not monitor events from the kernel or other processes, and thus cannot reveal inefficiencies due to platform code or poor interactions between applications.

Monitoring mobile applications during typical use: Panappticon is intended to help application and system developers identify bugs and locate inefficiencies in production on real users' phones, not just in the lab. Several tools exist for similar purposes.

The crash reports collected by iOS, Android, and Windows are of limited use, because the triggering conditions, e.g., an unexpected network environment, are not included and slow performance is never reported. Flurry [11] reports more detailed data like application launches and session lengths, but not with sufficient granularity to detect performance issues. Several tools collect traces for different purposes, e.g., detecting inefficient use of energy (e.g., PowerTutor [12], Carat [13], and ADEL [14]) and network bandwidth (e.g, MobiPerf [15]), and are orthogonal to Panappticon.

Characterization of mobile workloads performance: Panappticon should reveal the impact of architectural changes on user-perceived performance, helping guide hardware and operating system designers. Gutierrez et al. [16] developed a smartphone benchmark suite by characterizing representative smartphone applications. They found that smartphone applications have higher instruction cache miss rates than traditional SPEC benchmarks, and suggested increasing the cache size. In contrast, Panappticon is designed to study real workloads in the field, not summarizing benchmarks, and attempts to measure user-perceived performance explicitly.

8. CONCLUSION

This paper describes Panappticon, a system that records application-related events in operating system and framework libraries, correlates related events, and identifies individual user-perceived transactions. Panappticon determines the duration and critical path of each transaction, helping to make the root causes of performance bottlenecks clear.

We have described case studies demonstrating that Panappticon provides information that can be used to identify areas for improvement in smartphone applications, platforms, operating systems, and hardware. First, it can help identify application design inefficiencies, even when the root causes are subtle. Second, it can help system developers understand the impact of policy design decisions on user transactions (e.g., DVFS policy), allowing the designs to be optimized. Third, Panappticon can help smartphone hardware designers to understand the impact of architectural decisions on user-perceived transaction latencies (e.g., the impact of adding a CPU core), helping to guide design decisions.

The Panappticon source code and user study data are available at http://ziyang.eecs.umich.edu/projects/panappticon/.

9. REFERENCES

[1] I. Ceaparu, et al., "Determining causes and severity of end-user frustration," *Int. J. Human-Computer Interaction*, vol. 17, no. 3, pp. 333–356, June 2004.

[2] L. Ravindranath, et al., "AppInsight: mobile app performance monitoring in the wild," in *Proc. Int. Symp. Operating Systems Design and Implementation*, Oct. 2012, pp. 107–120.

[3] "Android SDK reference," http://developer.android.com/reference/packages.html.

[4] W. Enck, et al., "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. Int. Symp. Operating Systems Design and Implementation*, Oct. 2010, pp. 1–15.

[5] J. Smith, "Application breakdown by category," http://www.gottabemobile.com/2011/07/06/ipad-app-store-breakdown-top-apps-categories-chart/.

[6] "Galaxy Nexus," http://www.google.com/nexus/.

[7] H. Kim, et al., "Revisiting storage for smartphones," *ACM Trans. Storage*, vol. 8, no. 4, pp. 1–25, Nov. 2012.

[8] "Q–Q plot," http://en.wikipedia.org/wiki/Q-Q_plot/.

[9] P. Barham, et al., "Using Magpie for request extraction and workload modeling," in *Proc. Int. Symp. Operating Systems Design and Implementation*, June 2004, pp. 259–272.

[10] M. Jovic, et al., "Catch me if you can: performance bug detection in the wild," in *Proc. Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2011, pp. 155–170.

[11] "Flurry," http://www.flurry.com/.

[12] L. Zhang, et al., "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, Oct. 2010, pp. 105–114, http://powertutor.org/.

[13] A. J. Oliner, et al., "Carat: Collaborative energy diagnosis for mobile devices," in *Proc. Int. Conf. Embedded Networked Sensor Systems*, Nov. 2013, pp. 1–6, http://carat.cs.berkeley.edu/.

[14] L. Zhang, et al., "ADEL: An Automatic Detector of Energy Leaks for Smartphone Applications," in *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, Oct. 2012, pp. 363–373.

[15] Q. Xu, et al., "Cellular data network infrastructure characterization and implication on mobile content placement," in *Proc. Int. Conf. on Measurement and Modeling of Computer Systems*, June 2011, pp. 317–328, http://www.mobiperf.com/.

[16] A. Gutierrez, et al., "Full-system analysis and characterization of interactive smartphone applications," in *Proc. Int. Symp. Workload Characterization*, Nov. 2011, pp. 81–90.