Accelerating Mobile Applications through Flip-Flop Replication

Mark S. Gordon David Ke Hong
Peter M. Chen Jason Flinn Scott Mahlke Zhuoqing Morley Mao
EECS Department, University of Michigan
Ann Arbor, MI, USA
{msgsss,kehong,pmchen,jflinn,mahlke,zmao}@umich.edu

ABSTRACT

Mobile devices have less computational power and poorer Internet connections than other computers. Computation offload, in which some portions of an application are migrated to a server, has been proposed as one way to remedy this deficiency. Yet, partition-based offload is challenging because it requires applications to accurately predict whether mobile or remote computation will be faster, and it requires that the computation be large enough to overcome the cost of shipping state to and from the server. Further, offload does not currently benefit network-intensive applications.

In this paper, we introduce Tango, a new method for using a remote server to accelerate mobile applications. Tango *replicates* the application and executes it on both the client and the server. Since either the client or the server execution may be faster during different phases of the application, Tango allows either replica to lead the execution. Tango attempts to reduces user-perceived application latency by predicting which replica will be faster and allowing it to lead execution and display output, leveraging the better network and computation resources of the server when the application can benefit from it. It uses techniques inspired by deterministic replay to keep the two replicas in sync, and it uses *flip-flop replication* to allow leadership to float between replicas. Tango currently works for several unmodified Android applications. In our results, two computation-heavy applications obtain up to 2–3x speedup, and five network applications obtain from 0 to 2.6x speedup.

1. INTRODUCTION

Mobile devices have less computational power and poorer Internet connections than desktop and server computers. Size, power, and wireless networking constraints make it likely that this relationship will continue for the foreseeable future.

One way to increase the apparent computational power of a mobile device is to partition computation and *offload* some to a remote server [1, 6, 7, 13, 16]. However, offloading is not a panacea. Finding the optimal partition can be difficult, and the right partitioning strategy may change with environmental conditions such as network latency. Using a suboptimal partitioning strategy may increase network traffic and lower performance, and switching between local and offloaded computation may incur significant overhead due to the need to transfer state (e.g., all inputs and outputs of the offloaded computation) between computers.

One way to mitigate poor wireless network quality (e.g., high latency) is to reduce the number of network round-trips required to complete a user action. However, multiple network round-trips usually arise from data dependencies between consecutive requests to remote services. Costly reengineering of the application and/or service may be necessary to either eliminate such dependencies or hoist them into the cloud service.

The fundamental problem is that the right place to run a program depends on many factors. For example, code that interacts frequently with the user will often run best on the mobile device, while code that performs heavy computation or communicates frequently with other computers will often run best on a cloud server. Dynamically varying resource constraints, such as network quality and pre-existing load on the remote server, play a large role in determining where best to run code; for example, if the network is congested, it may be better to run more computation on a slower mobile device to avoid waiting for data to arrive at a remote server.

In this paper, we propose an alternative approach for improving user-perceived performance on mobile devices. Our approach is based on *replication* rather than partitioning. Instead of predicting which parts of the code should run on the mobile device and which parts should run in the cloud, our system, called Tango, runs the code on both devices and attempts to achieve the user-perceived performance of the faster replica.

Tango uses five main techniques to achieve this goal. First, Tango allows *either* replica to send output to the user. This is the fundamental property that reduces user-perceived latency: to the user, a Tango application will ideally appear to be running the faster replica—whichever replica that happens to be at the moment. For example, output from the remote server can be displayed on the mobile screen, even before the mobile computation reaches the point in its execution where the output is produced.

Second, Tango uses *deterministic replay* to ensure the replicas perform the same computation and (importantly) produce the same output [3]. This guarantee is what lets Tango safely use the output of the first replica: the trailing replica will *always* produce the same result, so it does not matter which replica's output is externalized. Deterministic replay typically designates one replica as the *leader*. The leader logs all non-deterministic events (e.g., network input, user input, and thread scheduling). The other replica is the *follower*—it supplies non-deterministic results from the log rather than re-executing such operations. Since the vast majority of operations performed by an application are deterministic, logging

MobiSys'15, May 18–22, 2015, Florence, Italy.
Copyright © 2015 ACM 978-1-4503-3494-5/15/05 ...$15.00
http://dx.doi.org/10.1145/2742647.2742649.

and replaying a (relatively) small amount of information ensures replica equivalency.

Third, in contrast to traditional deterministic replay, Tango shares the role of logging non-deterministic inputs from external sources between computers. Tango also shares the role of externalizing outputs from the replicated computation. For example, in the domain of mobile computing, some input sources are inherently tied to the mobile device (e.g., user input), while others are best tied to a remote server (e.g., network input, since the remote server usually has a better connection). Tango splits roles according to the type of I/O. The mobile device logs user input and sensor data, and it externalizes screen output. The remote server logs network input and externalizes network output.

Fourth, Tango replicates some I/O sources to reduce the amount of non-determinism that needs to be logged and to improve performance. Tango replicates data storage functionality such as the file system and databases. It also replicates substantial portions of the user interface stack.

Fifth, Tango allows either replica to log *internal* sources of non-determinism, which include thread scheduling, time queries, and asynchronous event scheduling. Unfortunately, prior uses of deterministic replay forced one computer (the follower) to always lag behind the other (the leader) to log such events. This a-priori designation limits performance in instances where the leader replica runs slower than the follower. Thus, Tango allows the role of leader to float between the two replicas: a technique we call *flip-flop replication*. When Tango predicts that the follower would be the faster replica, it flips the leader and follower roles. The role of leader may change many times during the execution of an application, e.g., due to alternation between periods of user interaction and periods of computation and/or network communication.

The Tango approach has many benefits. First, Tango achieves interactive performance that is very close to a system that always chooses the fastest location for execution. It does so without needing to predict resource availability, profile applications, or predict user behavior. Second, Tango supports low-overhead control transfer. In contrast to offloading, it does not need to ship all data used in offloaded computations because such data is automatically produced by the remote replica. This is particularly important because the amount of state reachable by offloaded computation can often be quite large. Third, Tango can provide fault tolerance by running multiple replicas. Importantly, by persisting its deterministic log within the cloud, Tango allows remote components to safely communicate over the network, so it hides the latency of multiple round trips over high-latency mobile networks. Finally, Tango can achieve these properties without modifying applications or profiling their executions. In practice, however, our prototype does not fully support the complexity of the Android platform, as discussed in Section 5, and this limits the set of applications that can be run with Tango at present. In addition, some applications may not be able to benefit from Tango due to more fundamental design limitations discussed in Section 8.

We show results from seven applications running Tango. Two computation-heavy applications obtain up to 2–3x speedup, and five network applications obtain from 0 to 2.6x speedup.

2. ARCHITECTURAL OVERVIEW

Tango targets interactive applications that run in the Dalvik VM on the Android platform (this includes the vast majority of Android applications). Tango harnesses a trusted remote server to accelerate an application running on a mobile Android device. The server could be managed, for example, by a trusted cloud service provider and be located in a data center.

Tango splits an application into a replicated portion and a non-replicated portion. The entirety of the replicated portion runs on *both* the mobile computer and remote server. In contrast, different components of the non-replicated portion run on *either* the mobile computer or the remote server. The high-level view of Tango's architecture is depicted in Figures 1 and 2. Each of four high level components can be thought of us as existing in their own address space; communicating over unidirectional channels shown by the arrows. In each diagram the left two components exist on the server while the right two exist on the client. The two diagrams demonstrate how communication patterns change when the leader is switched. Differences are highlighted as dotted lines.

2.1 Replicated Portion

The bulk of the replicated portion consists of the application code running in the Dalvik VM. The entire Dalvik execution of the application within the VM is replicated; Tango deterministic co-execution guarantees that both replicas execute the exact same Dalvik instructions on the same data. Since these instructions are deterministic, they produce the same result.

Our choice of enforcing determinism at the level of the Dalvik VM was driven by the desire to be processor-agnostic. The diversity in mobile phone and server hardware means that different platforms run a diverse set of processors with different ISAs (e.g., an x86 server and an ARM phone). It would be exceedingly difficult to guarantee determinism for binaries across ISAs (e.g., through deterministic binary-to-binary translation). We could emulate a different architecture on the server [21] but only at a significant performance cost. However, by implementing determinism in a VM such as Dalvik, we can simply guarantee determinism at the level of the Dalvik virtual ISA; this is sufficient for deterministic execution as long as the VM on each platform correctly implements the language ISA specification.

A second advantage of replicating at the level of the Dalvik VM is flexibility: a user can choose to run some applications with Tango and some without Tango. Since each application has its own instance of the Dalvik VM, we can replicate only the VMs for the chosen applications. The required isolation of application data in Android also enables this design choice.

In addition to the Dalvik VM, Tango also replicates many of the native methods that are invoked by the Dalvik VM. First, many native methods in standard libraries are deterministic and have no external side effects. We have identified and flagged many such methods, such as Java's math, compression, and locale implementations. Each Tango replica simply executes such flagged methods as though it were part of the Dalvik VM itself. Second, Dalvik applications have frequent interactions with the UI stack; these interactions are low-level and much more frequent than displaying user output or receiving user input. Tango therefore replicates the UI stack on the server; it simply omits any externalization of data to an I/O device. Third, storage (file system and database) interactions are very deterministic, so Tango replicates the storage subsystem.

We note, however, that our original decision assumed that applications are mostly composed of managed code, and native code execution would be a rare exception. As we discuss in Section 5, this assumption appears to be growing less true over time. Therefore, future systems may wish to revisit this decision and consider alternative replication strategies.

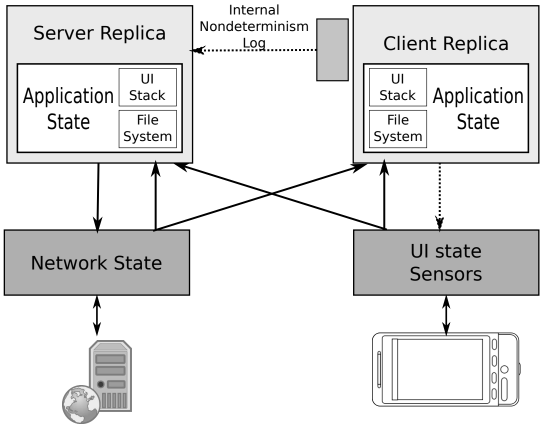

Figure 1: Tango's architecture: client is the leader. (Note that Client Replica externalizes the UI.)

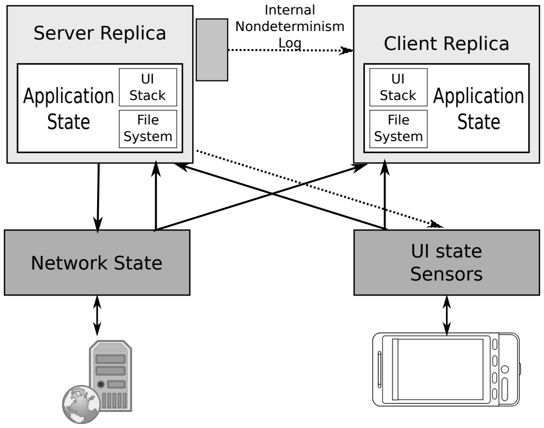

Figure 2: Tango's architecture: server is the leader. (Note that Server Replica externalizes the UI.)

2.2 Non-Replicated Portion

Some components of the system are nondeterministic (e.g., those that receive input from the user). These components cannot be executed independently on both replicas, lest the results differ and the replicas' executions diverge. Instead, Tango executes these methods only once, on behalf of the leading replica that reaches the native method first. Tango logs the result (return code and any modified Dalvik state). When the following replica arrives at the native method, Tango does not execute the method—instead, it replicates the effects on the Dalvik state and returns the same result. Figures 1 and 2 show the separation between these non replicated components (the darker boxes) and the application replicas, depending on which endpoint is leading.

Many of these non-replicated methods perform I/O. Such methods are sources of *external nondeterminism*. Data received from the user, the network, inter-process communication (IPC), sensors, and other sources must be logged and replayed so that each replica receives the same inputs. We observe that many of these data sources have natural affinity with one particular replica, and we pin each to its natural replica. For example, user input, IPC, and sensor data must come from the mobile client. In contrast, the cloud server almost always has a better network connection (e.g., lower latency and higher bandwidth) than the mobile device, so network communication should be done on the server. This is why network state is kept on the server while UI state and sensors are tied to the client in Figures 1 and 2.

Tango partitions execution so that these pinned methods run in a separate thread of control, called the *native thread*. Because they are deterministic, both replicas will execute the same methods with the same data in the same order. To call such a method, each replica sends a message to the *native thread*; for the replica on the local computer, this is a memory operation, but for the other replica, this is a network message. The native thread executes the method and sends the results to *both* replicas, as shown in Figures 1 and 2. The trailing replica may not have reached this point in its execution, so it logs the result until it is needed. Note that the follower need not send a request to the native thread.

There exists a final class of nondeterminism that is not inherently tied to any computer, which we call *internal nondeterminism*. This includes thread scheduling decisions, the timing of the delivery of asynchronous events, time queries, and entropy queries (e.g. reading `/dev/random`). Because Dalvik thread scheduling decisions are frequent, Tango implements a deterministic thread scheduler so that each replica independently makes the same scheduling choices—this avoids the need to communicate such choices between replicas. For asynchronous events and time queries, a single replica, called the leader, is responsible for executing such operations and sending the result to the other replica, called the follower. For example, in Figure 1, the client is the leader and thus handles the internal non-determinism. The follower generates the same asynchronous events at the same point in its execution and supplies the same time values to ensure replica equivalence. Because asynchronous events can be scheduled at any point in execution, the follower is prevented from surpassing the leader.

Tango implements a mechanism we call *flip-flop replication* to allow leadership to switch between replicas. Using a set of simple heuristics, this mechanism attempts to have the replica that can execute faster be the leader at any moment. Since different computers have different strengths, it is common for leadership to switch back and forth between computers; for instance, the mobile phone typically leads during periods of user interaction, and the server typically leads during compute-intensive periods or periods with network communication. The differences in Figures 1 and 2 illustrate that when the server is the leader, it can externalize the UI state on behalf of the client. In contrast, when the client is the leader, only the client will generate UI output.

3. EXAMPLE SCENARIO

In this section, we illustrate how Tango works by describing how it would accelerate an example application.

The scenario begins with the user interacting intensively with the application user interface. Events such as button and screen presses are broadcast to both replicas, but the client replica receives them first because communication with the server replica is subject to a network round trip. Thus, the client replica is the leader and decides when these events are scheduled into the application execution. The application may execute many synchronous native methods that query UI state; the leading client replica receives a quick response because of its co-location with the UI.

In a standard thin client solution, the server replica would lag far behind during this stage since it would initiate many synchronous UI requests and each would be subject to a full round-trip delay over a high-latency mobile network. In Tango, however, the UI events, the asynchronous scheduling decisions, and the result of synchronous native methods are all *pushed* to the server before the

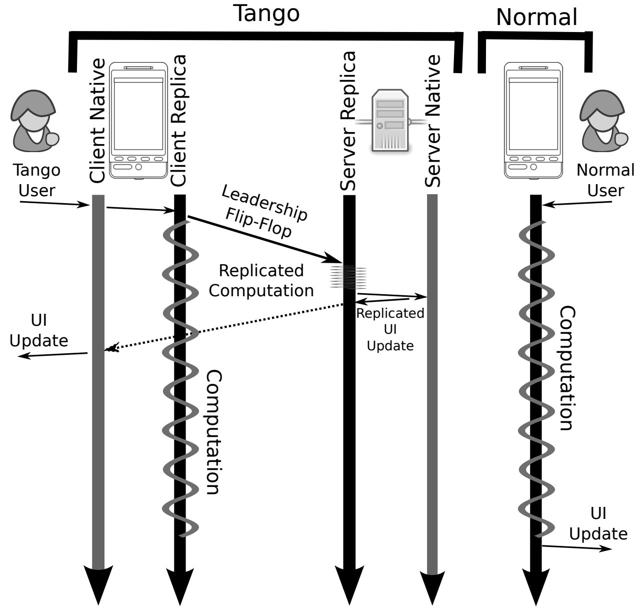

Figure 3: Compute-intensive phase: comparing Tango's speedup from faster server execution with normal execution.

server asks for the results. Thus, all the information the server will need for its execution is sent in pipeline fashion, *before the server even asks for the data*, and the server replica lags only a one-way network delay behind the client replica at the end of this phase.

The application next enters a compute-intensive phase of execution triggered by the user's input, as illustrated in Figure 3. Since the server has a faster processor than the client, the server replica quickly catches up and Tango switches leadership to the server. Note that because the server replica has followed the same execution as the client, it has the same state as the client replica and no state needs to be sent during the leader switch.

During the compute-intensive phase of execution, the server replica calls UI methods that send output directly to the client native thread. The client displays this output even though the local replica on the mobile computer is lagging behind the server because it is still working through the computation that was just externalized by the client native thread. This allows the user to experience the faster responsiveness of the server replica during this phase.

After receiving the output, the user spends some time thinking about his/her response to this output. While the user is thinking, the client replica catches up to the current state of the server. When the user interacts with the UI, the client again becomes the leader. In the event that the client does not have time to catch up before the next user interaction, the server will continue to operate as leader until the client does catch up. During this time Tango will be operating like a thin client, with all user interactions being routed through the server.

The next phase of the application involves frequent network communication with an external computer, as shown in Figure 4. The network operations are best done on the server, since it has a lower latency connection to the network. The server therefore assumes leadership.

The network communication involves several rounds of synchronous message exchanges with the external computer. Each round involves sending a message to the external computer, waiting for its response, then computing the next step in the message exchange (e.g., an SSL handshake followed by several database queries with data dependencies). In an unmodified client, this phase is slow since it involves many round trips over the high-latency wireless network. However, the server replica interacts with the external computer much faster because it runs in a data center and has an excellent Internet connection. In a mirror image of the first phase, network input, asynchronous scheduling decisions from the leading server replica, and responses to network methods are all sent to the client in pipeline fashion. Even with a slower processor, the client remains only a one-way network delay behind the server. Although the client takes longer to perform the small amount of computation in this phase, it catches up during network communication. The server waits for the remote computer to respond to each message, while the slightly-lagging client replica already has the response in its log. Thus, either the server or the client replica can quickly display output at the end of this phase.

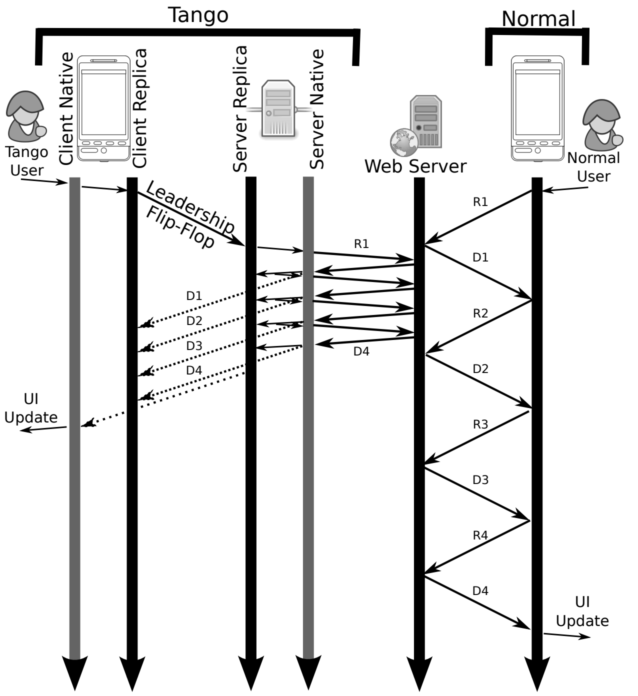

Figure 4: Network-intensive phase: comparing Tango's speedup from faster network response times from the server replica with a normal interaction with a Web server.

This multi-phase execution may repeat indefinitely. For example, the application may next interact heavily with the user, and so on. In each phase, Tango allows external components (i.e., the user and other computers) to experience the faster responsiveness of the replica that executes most quickly during that phase, while the slower replica can catch up to the faster replica during idle periods (e.g., when the application waits for the user and other computers). These two phases, compute-intensive and network-intensive, can be combined to get further benefits from Tango.

This scenario helps identify the specific instances in which we expect Tango to show the most benefit. First, to benefit from use of a remote server, the application should include at least one computation-heavy phase or network phase with multiple communication round-trips. Second, for Tango to outperform a thin-client solution, the application should have a phase that interacts with the user or other I/O on the client. Finally, if the application has compute-heavy phases, there should also be phases with user think time or I/O during which the client computation can catch up with the server. We believe that many mobile applications share these exact characteristics, in other words they combine user interaction with computation and/or network communication.

4. IMPLEMENTATION

To build our prototype, we modified the Dalvik Virtual Machine used in the Android operating system. Our modifications are built on top of version 10.1 of the CyanogenMod open source project [8] (based on Android's Jellybean branch).

Conceptually, an application running within the Dalvik VM can be thought of as consisting of two components: the Java state managed by the Dalvik VM and the native state managed by native code external to the Dalvik VM. The Dalvik VM's behavior is mostly deterministic, while interactions with many sources of nondeterminism such as user input, the network, the file system, and device sensors go through the native code. Therefore, it is most natural to deterministically replicate the Dalvik code, but not (by default) the native code.

4.1 Thread Scheduling

In order to make the behavior of the Dalvik VM fully deterministic, each replica must make the same thread scheduling decisions. Potentially, Tango could require the leader to record all thread scheduling decisions and send them to the follower, which would then replay those decisions. However, that strategy would add considerable network traffic to communicate those decisions. Instead, Tango uses a *deterministic* thread scheduling algorithm so that each replica independently makes the same decisions.

Tango allows a single *managed* thread to execute at a time. A thread is managed any time it is directly interacting with Dalvik VM state. As will be discussed in the next section, a thread interacting with native state can be *unmanaged* and can execute concurrently with a managed thread or other unmanaged threads.

Tango performs deterministic round-robin scheduling among the managed threads. The pre-existing Dalvik VM garbage collection mechanism intercepts all backwards jumps, method invocations, and exceptions raised—an executing Java thread must occasionally execute some of these *GC points* in order to continue to run. If a thread executes a pre-defined number of GC points without blocking, the Tango deterministic scheduler preempts it and runs the next runnable managed thread in the round-robin queue.

The Tango scheduler also provides deterministic implementations of synchronization primitives such as locks and condition variables. For instance, a thread requesting a lock simply checks if the lock is already owned. If so, it adds itself to a wait list for the lock and yields execution to the next runnable managed thread. Otherwise, it acquires the lock and continues executing.

4.2 Native Methods

A Java thread invokes native methods (which are usually written in C/C++) by using the Java Native Interface (JNI). Native methods do not interact directly with Dalvik VM state; instead, they use the JNI to do things like create objects, invoke Java methods, and access/modify object data. Tango records which calls are made by native code and what parameters are provided, so that it can faithfully replicate interactions between the Dalvik VM and the native code on both replicas. Since Tango records and deterministically reproduces *all* interactions between native methods and Dalvik state, it is able to run a native thread concurrently with another managed thread without leading to replica divergence (The reason it cannot run two managed threads concurrently is that it does not observe interactions via shared memory).

Thus, when a Java thread invokes a native method, Tango's default behavior is to move it from managed to unmanaged execution. This process requires no logging; the thread simply allows the next managed thread to execute and continues executing itself (although it may no longer access Dalvik VM state).

When the native method finishes, the unmanaged thread that invoked it must transition back to managed mode. At this point, the thread is inserted back on the round-robin scheduling queue of the Tango scheduler. However, to prevent replica divergence, both replicas must insert the thread at the exact same point in the execution of the managed threads. Essentially, the two replicas must agree on when an *asynchronous event*, in this case the completion of the native method, happens in relation to the execution stream of the Dalvik VM.

Tango uses the number of context switches as a convenient counter that denotes particular points in the application's execution. Asynchronous events are allowed to occur only at context switches. The leader has sole authority over determining when an asynchronous event happens. It logs the counter value and sends it to the follower. The follower schedules the identical event at the same point in its execution stream, ensuring replica determinism. Note that this implies that the follower should never execute ahead of the leader; otherwise, it may learn too late that it should have inserted an asynchronous event at a context switch that it has already executed.

In summary, when a native method finishes, the leader transitions the invoking thread from unmanaged to managed mode at the next context switch among managed threads. The follower will execute the same transition at the same point. Tango uses this technique for all asynchronous events. For instance, when a Java thread performs a timed wait on a condition variable, the replicas must agree on when the thread wakes up. Tango casts the wakeup as an asynchronous event to ensure determinism.

Finally, there can be long periods of execution where the leader does not need to record any asynchronous scheduling decisions. To allow the follower to make progress, the leader occasionally sends a negative acknowledgment confirming that no asynchronous events were inserted up to the current point in its execution. This allows the follower to continue execution up to that point.

4.3 Native Method Invocation

We next describe the invocation of a generic native method; the following section describes specific optimizations that Tango employs for certain classes of native methods.

A generic native method is not called directly by the unmanaged thread; instead the native method is run on the client in another thread context, called the *native thread*. Thus, if the client is the leader, its unmanaged thread performs an IPC to invoke the method. If the server is the leader, its unmanaged thread performs a remote procedure call.

The use of native threads allows the progress of the native code to be decoupled from the progress of the replica. For instance, if the server is the leader, it can update the screen via a native method running on the client, even though the corresponding thread on the client replica may still be working through the proceeding computation.

In our implementation, there are actually many native threads. Each native thread corresponds to a Java thread within the Dalvik VM. A native thread operates by running an event loop waiting for invocations from the corresponding Java thread on the leader replica. After finishing, it sends its response, containing the return values and/or modified Dalvik state, to both replicas. Tango synchronizes the native threads so that they do not perform concurrent conflicting operations.

The following replica may actually receive the response before it reaches the point in its execution where it invokes the native method. In this case, it logs the response and simply uses the logged values rather than invoking the method again. If the response has not yet arrived by the time it reaches the method invocation, the unmanaged thread on the follower blocks until it receives the response.

4.3.1 Native Method Annotations

Handling every native method in a totally generic way is impractical given the overhead of the generic approach and the large number of native methods called during an application's execution. Fortunately, there are several optimizations we can make for many native methods that have known behavior. We have annotated the most commonly used methods in the Android runtime to identify which optimizations can be used.

The possible annotations are as follows:

- *Deterministic:* the native method always produces the same results given the same inputs and has no external side effects. Such a method may be natively deterministic, or we may have added sufficient logging and replay of logged values to make it deterministic. Each replica executes a deterministic method locally; there is no divergence because both executions produce the same result.
- *Sync:* a method with the deterministic annotation that is also guaranteed never to block. For such methods, the calling Dalvik thread remains in managed mode rather than switching to unmanaged mode. This avoids the overhead of scheduling an asynchronous event to transition back to managed mode.
- *Inlineable:* a method with the sync annotation that is also guaranteed to be of short duration and which can run in any thread context. Calls to this method may be performed directly on the invoking thread within the replica as a performance optimization. Section 4.3.3 has additional details about these methods.
- *Ideal:* an inlineable method that also relies on no state from other native methods and does not externalize output. For Tango's purposes, these methods are considered part of the Dalvik VM itself. For example, java.lang.Math.log() is ideal.

4.3.2 Pointer Derandomization

Many methods that we would like to annotate as deterministic are not actually deterministic from Tango's perspective because they write a pointer to some native resource (as an int or long) into the Dalvik VM's state. If both replicas run such a method locally, then they would write different values to the VM state, leading to divergence. To address this issue, Tango implements pointer aliases. Pointer aliases map the native pointers to a deterministic index in a table. The alias can then be stored in the Dalvik VM without leading to state divergence. When the resource is accessed, it must be dealiased by the native code.

To implement this correctly, any thread that attempts to create a pointer alias should operate independently and draw from a different set of aliases. Otherwise it would be necessary to communicate the order in which the aliases were created. Tango accomplishes this by using the high bits of the alias to represent the thread context that created it and the low bits to represent the index into a pointer alias table for that context.

4.3.3 Inlining

The primary overhead when invoking native methods with Tango is that all data must be communicated between the Java thread and the corresponding native thread. Moving this data and switching thread contexts is substantially more expensive than simply invoking the native method directly on the Java thread. The *Inlineable* annotation provides a hint that a method can be run in the context of the calling Java thread to avoid this overhead.

However inlining works against one of the goals of Tango; namely, that the native state should be able to progress beyond the replica state. When a method is inlined, it gets its inputs from the calling replica, which prevents the desired decoupling. Therefore, a method is only inlined by the leading replica.

Additionally, the decision to inline a method cannot easily be undone. If a method was inlined and then did not return quickly, it could keep the replica and native state coupled for an extended period of time. It is for this reason that we use the inlineable annotation only for short-lived methods.

Finally, allowing a method to be inlined means that it may be executed in different thread contexts (the Java thread or the corresponding native thread). Some native methods make use of thread-local storage or are otherwise sensitive to the thread context, and they will not function correctly when inlined. Therefore, a native method must be agnostic to its executing thread in order to be marked inlineable.

4.4 External Nondeterminism

External nondeterminism refers to nondeterminism that has its source outside of the Dalvik VM and can be generated by only one device. Examples include user input, interprocess communication, network communication, and querying device sensors. Most of these sources of nondeterminism must be provided by the client device. However, some sources, like network communication, are better provided by the replay server.

4.4.1 External Inputs

The client and server run proxy threads that receive input from external sources such as the user interface and network. When such input is received, the proxy thread pushes the data to both the client and server replicas. This avoids the need for the leader to ask for input data when the data is not received by the computer that is currently leading (for example, network data when the client is leading).

Although both replicas receive the same data, the leader decides *when* the data arrives within the replicated execution stream. The arrival of external data is an asynchronous event that is scheduled using the methods described in Section 4.2.

Next, we describe how Tango handles each source of external nondeterminism.

4.4.2 Network Communication

One of the main ways that Tango improves user-perceived performance is by moving network communication to the server. Data received from the network is pushed to both replicas as soon as it arrives. The leader decides when the data arrives in the execution stream. This means that it decides the *number* of bytes returned by a `read` socket call (though the content of those bytes is pushed to both replicas). If the following replica has not yet received the data, it blocks until it arrives and delivers the specified amount of bytes. The leader also decides the timing and results of `poll` and `select`.

4.4.3 User Interface

User input is handled similarly to network input, except that it is the client that receives and broadcasts the input to both replicas.

User output is more complicated because Java threads invoke many native methods to perform even the simplest UI output operations. We originally found these interactions to be a significant source of performance overhead. However, we observed that almost the entirety of the Android UI stack is in fact deterministic, or can be made to behave deterministically with minor modifications including the use of pointer aliases described in Section 4.3.2. Therefore, Tango replicates the UI stack and runs separate copies on the client and server.

When the server is the leader, it broadcasts invocations of native methods in the UI stack to both of the replicated stacks. The server stack does not update the screen, but it provides immediate responses to the server replica to each of the numerous method invocations without intervening round-trip network delays. This lets the server execute quickly and continue to generate UI updates. *Critically, the server knows what information the client user interface stack will need because it has its own stack requesting the same data*. This allows all of the needed data to be sent to the client in pipelined fashion.

4.4.4 File System

Tango replicates portions of the client file system on the server. A significant percentage of the file system on Android phones is normally mounted as read only; this portion is simply replicated on the server.

Additionally, the Android file system implements strong partitioning separating different applications. Tango replicates the application-specific portion of the file system by having each replica update its local copy. Since each replica deterministically performs the same operations and the file system is itself deterministic, this ensures that the state of the file system remains consistent.

The remaining portion of the file system (e.g., the SD card) can be updated by many applications, some of which may run under Tango and some of which may not. Therefore, reads and writes to these files are not deterministic and instead must be performed by a native thread running only on the client.

This strategy makes all file system operations deterministic. Most operations require no extra communication, since they are performed deterministically on both the client and server. For instance, a newly installed application will start with the initial application-specific file system portion on both the client and server. Since the same file system operations are performed in both locations, the application-specific portion remains in sync throughout multiple executions of the application. Our implementation also provides deterministic access to Dalvik's SQLite interface, which is the standard database implementation on Android.

4.4.5 Time

Android applications frequently query the time. In Tango, the leader produces the timestamp and the follower uses the same value in response to the time query. During a leader switch, the previous leader sends the new leader its current timer values. The new leader adjusts time values that it subsequently returns to the application by an appropriate delta. To reduce the amount of data sent from the leader to the follower, we could potentially adopt a semi-deterministic clock, such as the one used in Arnold [9].

4.5 Leader Switching

Switching leadership is implemented by having the current leader send a message to the follower requesting a switch. This message includes the point where the current leader is in the execution of the program, and the current leader pauses its execution at this point until it receives a reply. If the follower accepts leadership, it will act as leader once it reaches the designated point in the program's execution. If the follower rejects leadership, it sends a rejection message to the current leader; that replica resumes execution and acts as leader. Thus, for any given point in the execution of the program, there is exactly one leader. A leadership switch incurs at least a one-way message delay in program execution (and possibly more if the follower has not fully caught up when it receives the message). A rejected leadership switch incurs a RTT delay.

The last challenge of Tango is then to determine when to switch leaders, using the above mechanism. Intuitively, the server should become the leader when there is a significant amount of computation to perform, or when there is network communication. Similarly, we want the client to lead when there is user interaction or other activities that are pinned to the client.

Because switching leadership is cheap, it always makes sense to switch when there is a dependency that will require an RTT to satisfy by the current leader. Therefore, if the leader needs data from the follower and the follower replica is not too far behind, the leader switches leadership to the follower. This heuristic is how switches due to UI input, network communication, and pinned native methods are informed. In the case of network communication, switches are triggered during a call to connect or when data is sent. This could degrade performance only in the case that no data is received in response to a request or when another dependency forces leadership back to the client before a response is received. Otherwise, this heuristic does not degrade performance compared to normal application execution.

Deciding to switch leadership due to computation is significantly more difficult. Tango must predict that there will be enough computation before the next dependency pinned to the client to at least make up for the RTT penalty incurred by switching leaders back and forth. Tango uses a technique similar to COMET [16] in which it tracks how much computation has been done since the last invocation of a native method that was pinned on the client. Currently, we switch after 200 ms of computation. Empirically, this value seems to set a good balance between not triggering superfluous leader switches and not forcing the client to lead for large amounts of computation. A more sophisticated approach would also factor in RTT and relative computation performance.

However, in all cases, a switch does not occur until Tango believes that the execution of the follower has "caught up" to the execution of the leader. This property is somewhat tricky to determine in a distributed system where there can be substantial latency between the two replicas.

To address this problem, we use a simple heuristic based on the following replica's *lag*. When the follower receives an asynchronous event or negative acknowledgment, it tags the event or negative acknowledgment with a timestamp indicating when the event was received. When the follower reaches the corresponding point in its execution, it compares the current time with the tagged timestamp to determine the lag for that event. If the lag exceeds a pre-defined threshold (500 ms), the follower transitions into a *lagging* state, pushing a notification of this change to the leader. While the follower is in this state, it rejects all leadership switches, and (upon receipt of the notification) the leader will no longer offer to switch, even if one of the triggering events above occurs.

To switch out of the lagging state the follower must observe an event lag of under 100 ms. This transition causes a notification to be pushed to the leader, and the leader may again offer to switch leadership if a triggering event occurs.

4.6 Server faults

Tango is designed to tolerate server faults. Consider the following potential problem. If the remote server is the leader, it may fail after sending several messages to remote computers. The content of these messages may depend on non-deterministic operations that the server has previously executed (e.g., random number generation, thread scheduling, or data received from a remote computer in a previous decision). The following replica on the client does not know these non-deterministic values, so it cannot produce the same application state that led to the messages already seen by external computers. Thus, there is no way to continue application execution. This scenario is why current offloading solutions such as Maui [7] and CloneCloud [6] do not allow methods that involve external communication to execute on the server.

In Tango, we solve this problem by sending the log of non-determinism to a backup server located near the remote replica, a standard log-based, rollback-recovery technique [11]. The non-deterministic events are sent to the backup server asynchronously as they are recorded. Prior to externalizing any output (e.g., network messages and UI updates), the remote server waits for the backup server to acknowledge the receipt of all non-deterministic events that precede the output.

When the client detects that the server has failed, it can fetch the logged non-deterministic events from the backup server. The client replica simply replays these events in the same manner as if they had come directly from the remote server. Once the client replica is caught up to the last operation performed by the remote server, it continues alone as a non-replicated execution. In principal, it could also start a new remote replica and resume Tango replication.

Tango currently tolerates a single server stop fault, but it could tolerate up to n simultaneous stop faults by persisting the log to n backup servers. Our measurements show that the overhead of persisting the log is negligible for $n = 2$. Note that it would be much harder to provide this fault tolerance in a standard offload system; one would either need to save an entire checkpoint of application state on the backup server prior to externalizing any output, or one would have to reproduce much of the Tango functionality to add log-based rollback-recovery to an existing offload system.

5. LESSONS LEARNED

We learned two major lessons in the course of this project; one was painful and the other was a pleasant surprise. We discuss both lessons in this section.

5.1 Engineering complexities

Retrofitting an established platform like Android that was not created with Tango's goals in mind proved to be considerably more challenging than we originally expected. Our expectation, outlined in our original position paper [14], was that using the Dalvik VM as the unit of application replication would lead to a clean partitioning, with the vast majority of the application executing within the replicated portion and there being only a few narrow interfaces between the replicated VM and the non-replicated system infrastructure. Unfortunately, as evidenced by the long time we have worked on this idea since publishing the original position paper, that expectation proved to be overly optimistic.

The complexity of the Android platform had two effects. First, it greatly increased the work required to support the unmodified applications reported in this paper. Second, even after significant implementation effort, many applications are still unable to reap the benefits of Tango that they probably should. Broadly speaking, there are three common reasons that applications that could benefit from Tango do not: implementation errors in our code, interprocess communication, and use of extensive native libraries such as WebKit.

Tango operates on top of the large Android code base, significant portions of which we have retrofitted to behave deterministically (or to which have added sufficient logging to compensate for nondeterminism). Ensuring correctness with a small developer team has been very challenging. Many applications that should benefit from Tango crash due to the resulting instability.

The interface between Dalvik applications and the rest of the Android system has proven to be broad rather than narrow. For instance, interaction with the UI stack was so frequent that we were forced to replicate the UI stack itself. A current source of pain for us is that Android IPC calls are common—all applications must communicate with the system server (itself a Dalvik VM instance). For many of these IPC calls, nothing is returned so Tango can faithfully replay them without logging. However, many other calls that are inherently deterministic may return values that are dependent on state within the system server. A potential solution to this problem is to replay the system server alongside the application. However, this is a substantial engineering challenge because the system server makes use of many native methods not available within normal applications.

Most Android applications execute a large number of native methods, and the percentage of such methods appears to be growing over time. This is challenging for Tango because we must identify, categorize, and possibly make deterministic each such method. Additionally, some applications make heavy use of native libraries, notably the WebKit browser engine. While it should be technically possible to make most elements of WebKit and similar libraries behave deterministically, the engineering effort is again substantial. So, Tango currently does not work effectively with applications that use these libraries. In particular this means that Tango does not work with Web browsers that use WebKit.

Tango is designed to work with any conformant implementation of the Dalvik Virtual Machine. Therefore Tango should work just as well in the presence of a JIT or within Android's new Ahead-of-Time compilation system, ART. However, the current prototype does not support the JIT and was built on the traditional Dalvik virtual machine. Supporting the JIT would require more engineering effort. Switching over to ART would require a significant porting effort.

Given these limitations, it is clear that Tango should be regarded as a research prototype for demonstrating the potential of replication. We believe it has been successful in this regard. Were we to start afresh, we might consider other methods of replicating execution: for example, requiring that the ISA be the same between client and server and enforcing determinism at the process level inside the OS.

5.2 Benefit to network applications

Originally, we envisioned Tango solely as a way to accelerate applications with computation-heavy phases. The pleasant surprise during the course of this project was discovering that Tango could also benefit applications with network-intensive phases. The key insight is that the log of non-determinism simultaneously provides a method to keep the client and server executions in sync *and* a log for rollback-recovery that allows the server to safely externalize network output after committing the log to another server.

Indeed, we expect that the number of network-intensive applications that could benefit from Tango currently far exceeds the number of compute-intensive applications that could benefit. Mobile application developers currently go to considerable effort

Benchmark	Package Name	Description	Interaction	Network RTTs
Sudoku	de.georgwiese.sudokusolver	Game aid, solves Sudoku puzzles given known values	Finish solving a Sudoku grid given a single cell.	N/A
Poker	com.leslie.cjpokeroddscalculator	Game aid, uses Monte Carlo simulation to find odds of winning a poker game	Compute the probability of winning from initial poker state.	N/A
Hoot	com.hootsuite.droid.full	Manages posts to/from multiple social networks including Twitter, Facebook, LinkedIn, and Foursquare.	Search Twitter given a keyword (queries api.twitter.com)	5
TapTu	com.taptu.streams	A social news reader that supports Facebook, Twitter, and LinkedIn	Update Facebook feed (queries api.facebook.com)	4
Email	com.android.email	A built-in email application for Android platforms	Update Email's inbox	4
Instagram	com.instagram.android	A social network application for sharing photos and videos	Update Instagram posts	3
Pinterest	com.pinterest	A social network application for creating and sharing visual bookmarks	Update Pinterest boards	2-8

Table 1: Description of applications used in macrobenchmarks.

to reduce the number of round-trips between the client and the cloud, adding considerable development cost and complexity. With Tango's form of replication, this application complexity is not necessary; the cost of multiple round-trips are automatically hidden by execution on the server replica.

6. EVALUATION

To evaluate Tango, we ran it both against small microbenchmarks and whole applications from Google Play. The macrobenchmarks in Section 6.2 show how Tango performs on unmodified, real-world applications, demonstrating that performance gains can be obtained for both compute-intensive and network-intensive applications. The microbenchmarks in Section 6.3 quantify the overhead of preemption and the overhead added to the Java Native Interface.

6.1 Methodology

We tested Tango on a Samsung Galaxy S3 smartphone running the Android 4.2.2 (Jellybean) operating system. To provide support for a variety of mobile platforms we implemented Tango on top of the CyanogenMod [8] 10.1 code base. The replay server used for all experiments has a 3.40 GHz i5 processor (i5-3570) with 4 GB of RAM. For repeatability of results, the mobile client and replay server were connected via USB. Network latency was emulated during experiments by setting netem [19] rules on the client. To further investigate Tango in real world environments, we also evaluated Tango's performance when the mobile client and replay server are connected over a campus WiFi network.

Tango does not currently support the Just-In-Time compiler. All *baseline* builds of the Dalvik VM are therefore built without a JIT for a fair comparison.

6.2 Macrobenchmarks

We used seven applications available on Google Play to evaluate the performance benefits of Tango. Two of these applications, *Sudoku* and *Poker*, are compute-intensive, and the other five, *Hoot*, *TapTu*, *Email*, *Instagram* and *Pinterest* are network-intensive. For each application, we configured Tango to connect its mobile client and replay server over either USB or WiFi and selected a representative latency-sensitive interaction to measure user-perceived latency. Table 1 details the applications and the recorded user interactions. For the emulated configuration, we added from 0 ms to 500 ms of network delay to both the baseline and Tango configurations.

The compute-intensive applications were selected from the applications used in COMET [16]. The network-intensive applications were selected by looking for popular or mash-up applications that contained some repeatable user interaction for testing. From this set, due to the challenges described in Section 5.1, we dropped applications that do not currently work with Tango.

While investigating applications suitable for measurement, we found approximately 10 applications that function properly with Tango but cannot currently achieve performance improvements due to the limitations discussed in Section 5.1. These applications have several client-pinned dependencies due to either IPC or use of WebKit that force leadership back to the client after a compute chunk but before the externalization of output, or, alternatively, before a network transaction completes. In the case of computation-based applications, this results in an RTT performance degradation. In the case of network applications, there is typically no degradation. We did not evaluate one additional application, Gmail, because we saw that it completed its network operation in a single RTT and therefore would not benefit from Tango.

6.2.1 *Compute-intensive Macrobenchmarks*

For the compute-intensive benchmarks, we recorded the time difference between the user input that triggered the interaction and the externalization of the final screen update as their user-perceived latency. We repeated this experiment ten times for each configuration. Figures 5 and 6 show average latencies and 95% confidence intervals for Sudoku and Poker, respectively. For 100 ms round-trip network latency, which is the typical delay for current LTE networks, Tango reduced user-perceived latency for *Sudoku* by 0.6 seconds (50%). For *Poker* at 100 ms network latency, Tango reduced the user-perceived latency by 1.9 seconds (68%). Over WiFi, Tango decreased user-perceived latency for *Sudoku* and *Poker* by 0.6 seconds (53%) and 1.9 seconds (68%), respectively.

To better understand where the performance benefits are coming from and how much further Tango could be improved, we took

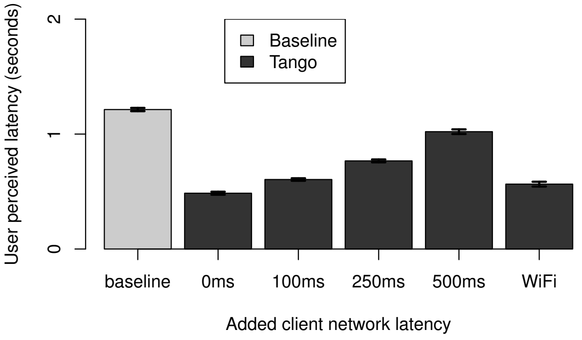

Figure 5: **Sudoku performance**

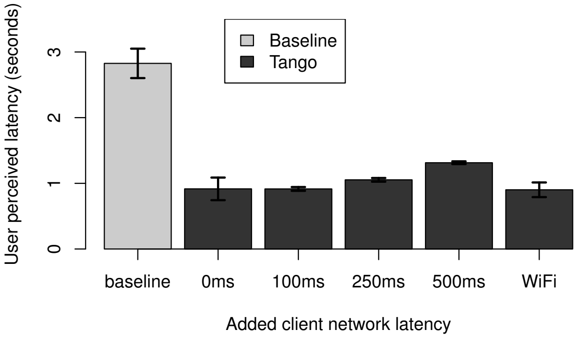

Figure 6: **Poker performance**

a closer look at the Sudoku application. Tango took on average 485 ms when performing the Sudoku interaction with no latency. The first 268 ms after user interaction were spent with the client leading execution. This corresponds to approximately 68 ms of UI interaction, followed by 200 ms of computation before the leadership switch is triggered. After the leadership switch, the server finishes the computation and executes the screen updates on its local UI stack in another 100 ms. The remaining 117 ms are spent by the client to display the screen output created by the server. This overhead is exacerbated by CPU contention with the client replica still performing the replicated computation. By momentarily stopping the replica execution on the client this overhead can be brought down to 45 ms. This indicates that more sophisticated scheduling could yield further performance benefits.

6.2.2 *Network-intensive Macrobenchmarks*

We evaluated *Hoot*, *TapTu*, *Email*, *Instagram* and *Pinterest* to investigate Tango's performance gain in network-intensive applications. *Hoot* and *TapTu* use social network APIs to allow users to interact with multiple sites such as Facebook and Twitter. *Email* is the built-in Email application in Android systems that retrieves email update from remote servers. *Instagram* is a social network application for users to share images and videos. *Pinterest* allows users to bookmark image based posts on their pin boards.

In these network applications, we focus on the user-perceived latency for a network update by refreshing the screen, a common

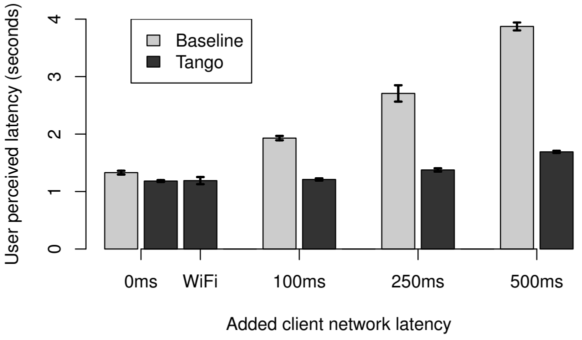

Figure 7: **Hoot performance**

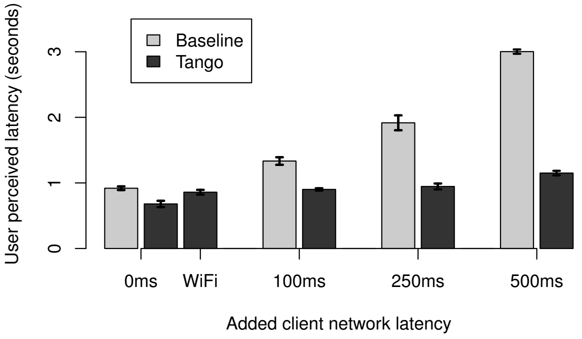

Figure 8: **TapTu performance**

action in many network-based mobile applications. Similar to the computation benchmarks, we recorded the time difference between the first user input and the final display output as the user-perceived latency. We repeated these experiments 14 times for each configuration (we ran more trials because of the added variance inherent in accessing external sites). The average latencies and 95% confidence intervals are presented in Figures 7, 8, 9, 10 and 11. The results show that Tango achieves speedup of 1.6–2.3x, 1.5–2.6x, 1.5–2.2x and 1.3–2.2x under different network latencies for *Hoot*, *TapTu*, *Email* and *Instagram*, respectively. The user-perceived delay for *Pinterest* was similar in all scenarios except that there was a 17% improvement in user-perceived latency at 500 ms network latency.

Compared to the compute-intensive applications, the relationship between performance gains and network latency is inverted. The user-perceived latency under different network RTT values follows a relatively simple model of $\alpha_x + \beta_x \times RTT$. For *baseline* and Tango, α_x is approximately the same. However, $\beta_{baseline}$ is proportional to the number of request/response pairs in the interaction while β_{Tango} is fixed at approximately 1, reflecting only two leadership switches. Using a least squares linear regression, we calculate $\beta_{baseline} = 5.05$ for *Hoot*, $\beta_{baseline} = 4.16$ for *TapTu*, $\beta_{baseline} = 3.87$ for *Email* and $\beta_{baseline} = 3.60$ for *Instagram*. These numbers indicate that *Hoot*, *TapTu*, *Email* and *Instagram* do approximately 5, 4, 4 and 3 requests in serial,

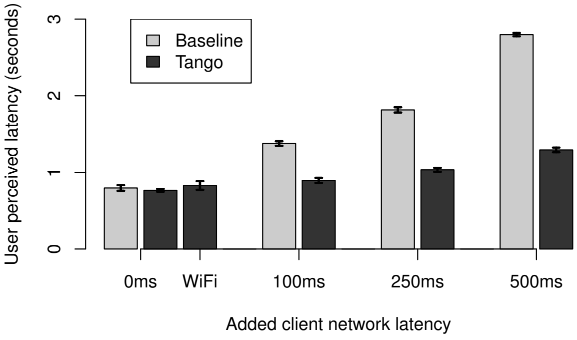

Figure 9: **Email performance**

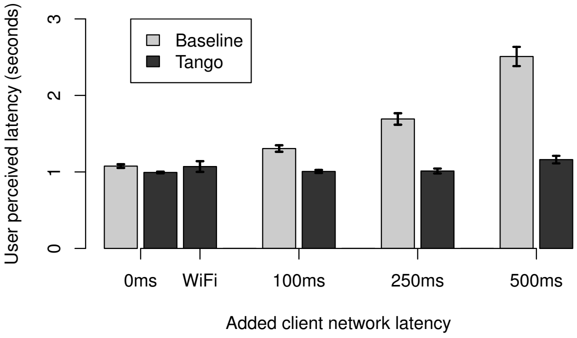

Figure 10: **Instagram performance**

respectively. The same linear regression computes $\beta_{Tango} = 1.05$, $\beta_{Tango} = 0.85$, $\beta_{Tango} = 1.03$ and $\beta_{Tango} = 0.34$ for *Hoot*, *TapTu*, *Email* and *Instagram*, respectively. The coefficient of determination for *Hoot*, *TapTu*, *Email* are over 0.94 in the baseline case and range between 0.7 and 0.93 in the Tango case, indicating that the linear regression model is a good fit for characterizing their latency. *Instagram* and *Pinterest* are exceptions. Applying the same regression on them yields coefficient of determinations of 0.48 and 0.52, respectively. We believe that this mismatch is due to these applications overlapping computation and network interaction in a sophisticated manner.

We verified the regression models using *tcpdump*. *Hoot* updates its Twitter feed by making an HTTPS query to api.facebook.com during the first RTT. An HTTPS request from a new connection takes one RTT to establish a TCP connection, two RTTs to complete an SSL handshake, and a final RTT to request an object over HTTPS and receive a response. In total, 5 RTTs are spent in refreshing the Twitter feed in *Hoot*. Similarly, *Email* makes an HTTPS request from a new connection, taking 4 RTTs. The other applications had similar behavior. *Pinterest*, however, was an outlier in that it appeared to vary the number of serial network requests it made and it also seemed to have enough computation to overlap with network activity so that only one round-trip was on the critical path at low latencies.

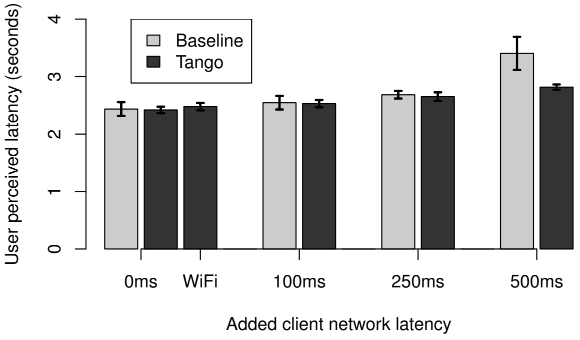

Figure 11: **Pinterest performance**

App.	Baseline	Tango	Percentage
Sudoku	2.47±.18 J	2.43±.17 J	-1.62%
Poker	7.14±.41 J	5.43±.22 J	-23.9%
Hoot	2.77±.15 J	2.46±.21 J	-11.2%
Taptu	2.14±.23 J	1.74±.21 J	-18.7%
Email	1.74±.12 J	1.78±.18 J	+2.30%
Instagram	2.26±.16 J	2.35±.20 J	+3.98%
Pinterest	4.16±.49 J	4.06±.30 J	-2.40%

Table 2: **System energy consumption on the smartphone**

6.2.3 Energy Consumption

We also measured the difference in energy consumption caused by using Tango with each application. To measure energy, we assume that a user will pause for a fixed think-time after seeing each screen update before beginning the next iteration of the benchmark. We used a user think time of 3 seconds for *Poker* and 1 second for all other applications (in the case of *Poker*, this ensures that the client catches up during the designated user think time). The energy measurements are collected using a *Monsoon Power Monitor* with WiFi used for communication and cellular data disabled on the smartphone. Each energy measurement includes the time to finish all computation on both the client and server, as well as the fixed user think-time after displaying the result. Energy measurements on each application benchmark are repeated 10 times. Table 2 shows the average values and 95% confidence intervals for the amount of energy consumed by the smartphone for each application.

Compared to the baseline, Tango uses less energy for most, but not all, benchmarks. It uses 2–4% more energy for *Email* and *Instagram*, but reduces energy usage by up to 19% for the other network-intensive applications. For compute-intensive applications, Tango reduces energy usage by up to 24% (in *Poker*). There are two factors at work. First, Tango introduces some overhead through extra computation and communication, increasing energy usage. However, Tango also reduces the time to display a result, decreasing energy usage because activities take less time.

For the compute-intensive benchmarks, Tango must finish the computation on the client after displaying the result; however, the finish of the computation can be overlapped with user think-time for interactive applications. In the future, we could further reduce energy usage by shipping a checkpoint of application state from the

App.	Received Data	Sent Data
Sudoku	115 KB	25 KB
Poker	49 KB	25 KB
Hoot	195 KB	133 KB
TapTu	74 KB	14 KB
Email	98 KB	49 KB
Instagram	25 KB	45 KB
Pinterest	461 KB	39 KB

Table 3: Size of log being communicated during replay. *Received data* refers to data being received by the client. *Sent data* refers to data being sent by the client.

server to the client to avoid the need to finish the computation on the client.

6.2.4 Log Sizes

Of course, Tango has to communicate log data between the client and server to operate. Table 3 shows the amount of (compressed) data being transmitted over the network between client and server for Tango. The log sizes recorded in the table include application initialization and *three* user interactions. Each run takes approximately 30 seconds, putting the required average client upload bandwidth at 3.7–35 kbps and the average client download speed at 6.7–125 kbps, which is very small compared to typical bandwidth in modern cellular networks.

6.3 Microbenchmarks

There are two primary sources of overhead that Tango introduces into the system; the code to implement preemptions (see Section 4.1) and the handling of native calls made by an application. Therefore, we measure overhead with two different benchmarks; one tests the overhead of pure computation (no native calls) and the other tests the overhead of a tight loop of native calls with varying policies. In these tests, we replaced the replay server with a dummy terminal that records all transmitted data (so leader switching is disabled).

6.3.1 Pure Computation

We tested Tango on two pure computation applications and measured their relative overhead compared to the *baseline* build. The first test, which simply calculates Fibonacci numbers recursively, had an overhead of 8.48%. This test runs entirely in the interpreter which should maximize the cost of preemption. The second test, which allocated objects in a tight loop, had a lower overhead of -2.01%. We expect this test to have a lower overhead than the Fibonacci test because it spends more time doing allocations and garbage collection where no overheads have been added. It is unclear why Tango outperforms the *baseline* build for this test, but the result is reproducible. The most likely cause is our changes to the default Dalvik thread scheduler, which may favor this benchmark.

6.3.2 Native Methods

Next we measured the performance overhead of invoking native methods with different annotations. Table 4 shows the overhead of making native calls with the *Deterministic*, *Sync*, *Inlineable*, and *Ideal* annotations. The definitions of these annotations are given in Section 4.3.1.

Annotation	Baseline	Tango	
	Calls/ms	Calls/ms	Bytes/Call
None	729	9.18	2.87
Deterministic	N/A	9.97	0.565
Sync	N/A	13.1	~0
Inlineable	N/A	196	~0
Ideal	N/A	456	~0

Table 4: The throughput and logging overhead during a tight loop around a native method with various flags.

App.	None	Deter.	Sync	Inlineable	Ideal	Estimated Overhead
Sudoku	20	880	1280	57271	3149	401 ms
Poker	17	935	788	26944	516	254 ms
Hoot	36	11106	3093	46001	9864	1510 ms
TapTu	31	6555	6054	22698	1662	1190 ms
Email	40	1448	3246	22011	655	474 ms
Instagram	2684	1676	851	43666	3721	684 ms
Pinterest	2638	3836	2126	189156	22642	1550 ms

Table 5: Number of different categories of native calls during initialization and three user interactions.

Each of these test scenarios calls a native method that performs the same operation; a simple hash computation on a 10 character string passed from Java. It is clear that executing an unannotated native method can be very expensive, with a reduction in throughput of nearly 80X. The *Deterministic*, *Sync*, and *Inlineable* tests each show the reduction in overhead when the output of the native method no longer needs to be logged, when no scheduling decisions need to be logged, and when the native method can be executed directly on the calling Java thread respectively. The most dramatic difference is when the *Inlineable* flag is applied; revealing that the overhead of invoking a native method on a native thread is significant.

The *Ideal* annotation is used when the native method can be passed a completely uninstrumented JNI function table, which allows it to achieve the lowest overhead. The remaining overhead comes from Tango-related code that touches all native functions (i.e., the code that decides how to invoke a native function based on these flags).

While these overheads are fairly high for this microbenchmark, in practice, the number of native functions invoked during an application's execution is much lower. Table 5 shows how many times native functions with different flags are called during a sample run of three apps we used for testing. The overheads were estimated using the data from Table 4. Each of these runs represents about 30 seconds to a minute worth of execution, putting the overhead in the 1–6% range.

7. RELATED WORK

Tango is the first method to deterministically replicate a computation on the mobile client and a remote server to improve performance.

Using cloud resources to improve the performance of mobile applications is a tantalizing goal, so many prior systems have focused on offloading computation from the mobile computer to the server [13]. Prior systems such as Maui [7], CloneCloud [6], Odessa [22], Comet [16], and Spectra [15] all *partition* functionality by executing a given piece of the computation on either a mobile device or a server, but not on both. Partitioning of a mobile application is a specific instance of partitioning distributed applications in general [18]. Partitioning is usually subject to some constraints related to functionality or privacy. For example, some functionality is tied to a specific device (e.g., sensors on the mobile device), and some state may be considered too sensitive to allow it or data derived from it to migrate off the mobile device [4].

Partitioning of an application may take place at units of various granularity, such as function calls or threads. None of these partition-based offloading systems allow code that externalizes network output to be offloaded because of the fault-tolerance issues discussed in Section 4.6; thus, a major benefit of Tango is that it can offload network-intensive phases that prior systems cannot.

Chroma [2] and Slingshot [24] replicate a computation on multiple servers, but they do not enforce determinism at all. Similarly, Higgins et al. [17] replicate a simple speech recognition service on a client and server, but rely on the service to be stateless and deterministic. Unlike Tango, their replicas will diverge and yield different results for more complex services.

Other systems deterministically replicate a mobile application and execute it with additional instrumentation on the server to debug the execution or add security checks [5, 21]. The goal of such systems is not to improve performance; in fact, the server replica always trails the mobile replica and so cannot offer faster response time.

Replication has long been used to provide fault tolerance [11]. Some systems use deterministic record and replay to keep replicas synchronized [3]. However, prior systems only switch the role of leader on failure (because their goal is fault-tolerance, not improving interactive performance). Tango targets a heterogeneous environment in which different replicas have different strengths and weaknesses. Therefore, it is often beneficial to switch the role of leader as the application enters different phases of execution, and Tango is designed to identify such opportunities and switch appropriately.

Tango uses many techniques developed for deterministic record and replay [10, 12, 23, 25] but applies them to replicate computation on multiple computers. It differs from these prior systems in that different types of external nondeterminism are intercepted on different computers for the same logical computation, then broadcast to all computers. Further, some forms of nondeterminism are mitigated by replicating additional software components such as the file system and UI stack on multiple computers.

8. LIMITATIONS

Many of the limitations of Tango from an implementation perspective have been discussed in Section 5.1. The remaining limitations are related to the actual design of Tango.

One of the most significant limitations of Tango is dealing with false dependencies among threads. Suppose that an application has a thread that does network communication and another that is reading sensor information. The former thread would perform better if the server is the leader, while the latter thread would perform better if the client is the leader. With Tango, only one endpoint can lead at any given time.

The other key limitations of Tango primarily affect computation-heavy applications. First, Tango forces serialization of all managed threads within an application. Modern phones usually have more than one CPU core, meaning that applications running within Tango cannot make use of all resources available to them. To remedy this problem, we would need to adopt a deterministic multicore scheduling algorithm such as Kendo [20].

Second, even when Tango correctly predicts a large amount of upcoming computation, it is still possible to see a loss in performance. This is because any time that the server must return leadership to the client (e.g., due to a series of client-pinned native methods) prior to externalizing output, then user-perceived latency does not improve. Effectively, we have simply added an RTT to the user perceived latency. We could potentially overcome this with better prediction, or we could use checkpoint and restore to transfer the server's application state to the client. In the case of checkpoint and restore, the amount of state transferred would have to be small enough to make this practical.

Third, deciding when to switch leaders remains a difficult task. This work provides a simple solution that appears to work well in practice. However, on some applications, an oracle could do significantly better. In the case of especially poor decisions, Tango may even degrade user-perceived latency.

9. CONCLUSION

We have presented a system called Tango, which accelerates mobile applications by replicating their execution on a remote server. By displaying output to the user from the replica that is currently leading the execution, Tango reduces the latency seen by users. To accomplish this speedup, we developed a technique called flip-flop replication, in which the leadership role floats to the replica that is currently faster, with the other replica following the execution through deterministic replay. Tango works for unmodified applications that run on the Dalvik VM on Android devices. We demonstrated speedups of 2–3x for two compute-intensive applications and speedups of up to 2.6x for five network-intensive applications.

Acknowledgments

We thank the reviewers and our shepherd, Lin Zhong, for their thoughtful comments. This material is based upon work supported by the National Science Foundation under grant numbers CNS-0905149, CNS-1059372, CNS-1345226, and CNS-1039657. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the University of Michigan or the National Science Foundation.

10. REFERENCES

[1] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang. The case for cyber foraging. In *the 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[2] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, pages 273–286, San Francisco, CA, May 2003.

[3] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. on Computer Systems*, 14(1):80–107, February 1996.

[4] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning . In *Proceedings of the 2007 Symposium on Operating System Principles (SOSP)*, October 2007.

[5] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 1–14, June 2008.

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic execution between mobile device and cloud. In *Proceedings of the 6th ACM European Conference on Computer Systems*, Salzburg, Austria, April 2011.

[7] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making smartphones last longer with code offload. In *Proceedings of*

the 8th International Conference on Mobile Systems, Applications and Services, pages 49–62, San Francisco, CA, June 2010.

[8] CyanogenMod Community. CyanogenMod – Android community operating system. `http://www.cyanogenmod.com/`.

[9] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, October 2014.

[10] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Boston, MA, December 2002.

[11] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[12] S. I. Feldman and C. B. Brown. Igor: a system for program debugging via reversible execution. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 112–123, 1988.

[13] J. Flinn. *Cyber Foraging: Bridging Mobile and Cloud Computing*. Morgan and Claypool Publishers, September 2012.

[14] J. Flinn and Z. M. Mao. Can deterministic replay be an enabling tool for mobile computing? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile)*, March 2011.

[15] J. Flinn, S. Park, and M. Satyanarayanan. Balancing Performance, Energy, and Quality in Pervasive Computing. In *Proceedings of the 22nd International Conference on. Distributed Computing Systems*, Vienna, Austria, July 2002.

[16] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, 2012.

[17] B. D. Higgins, K. Lee, J. Flinn, T. J. Giuli, B. Noble, and C. Peplin. The future is cloudy: Reflecting prediction error in mobile applications. In *Proceedings of the 6th International Conference on Mobile Computing, Applications, and Services*, Austin, TX, November 2014.

[18] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 187–200, New Orleans, LA, February 1999.

[19] The Linux Foundation, `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`. *netem*.

[20] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–108, March 2009.

[21] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: Versatile protection for smartphones. In *Proceedings of the Annual Computer Security Applications Conference*, December 2010.

[22] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govidan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications and Services*, Washington, DC, June 2011.

[23] S. Srinivasan, C. Andrews, S. Kandula, and Y. Zhou. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Technical Conference*, pages 29–44, Boston, MA, June 2004.

[24] Y.-Y. Su and J. Flinn. Slingshot: Deploying stateful services in wireless hotspots. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications and Services*, pages 79–92, Seattle, WA, June 2005.

[25] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 2007 Workshop on Modeling, Benchmarking and Simulation (MoBS)*, June 2007.