Towards Flexible Offloading in Mobile-Cloud Computing

Xin Zhang    Xujie Si    Ravi Mangal    Hadi Esmaeilzadeh    Mayur Naik
Georgia Tech

Abstract
Mobile-cloud computing seeks to boost mobile devices by offloading compute-intensive tasks in mobile applications to more powerful machines. Existing mobile-cloud systems use a restricted strategy of computation and communication, which limits the scope of offloaded tasks and the applications that can utilize offloading. We explore the opportunities and challenges of relaxing this strategy along two axes: allowing communication while performing offloaded computation, and preserving remote state across offloaded computations. We propose four strategies across these two axes and develop a unified framework to find an offloading solution for a given program trace under each strategy. We do an empirical study on traces of 14 Android applications in various network settings. Our study yields insights into the limits and benefits of each strategy, and shows that allowing communication while performing offloaded computation gives significant performance and energy gains (2.7× average speedup and 4.4× average energy reduction).

1. Introduction
As the computing landscape changes to provide a more personalized and targeted experience to users, the demand for compute-intensive tasks on mobile devices such as tablets, smartphones, and smart watches increases. However, this demand coincides with diminishing returns from transistor scaling [10] that may curtail mobile devices from providing new capabilities. Battery and power constraints further limit the performance of mobile devices. Alternative approaches to mobile computing are therefore needed to improve performance without significant costs in energy.

Mobile-cloud computing is one such emerging approach in which the mobile device offloads portions of its computation over the network to a more powerful machine, speeding up execution and even potentially conserving battery energy [12, 15]. This paradigm is applicable to a spectrum of computing and networking contexts, with the remote system ranging from a personal laptop in close proximity of the mobile device, to a local-area server, to a remote commercial cloud. A central challenge in mobile-cloud computing lies in seeking general strategies that exploit the ever-increasing connectivity of devices to enable higher performance and richer applications on mobile devices.

The predominant strategy in existing mobile-cloud computing systems [7, 8, 14, 19] prevents the remote system from communicating with the mobile device until the offloaded task is completed. Moreover, the remote system is not expected to preserve the state of the offloaded task. While this strategy simplifies the overall system, however, it limits the scope of offloaded tasks and the applications that can utilize offloading.

In this paper, we explore the opportunities and challenges of relaxing the existing restricted strategy of computation and communication in mobile-cloud computing. We define, evaluate, and compare three additional strategies that expect the remote system to preserve state regarding the offloaded task and/or communicate with the mobile device while performing the offloaded computation. We categorize these four strategies across the two axes of remote state and communication pattern, as presented in Figure 1: (1) transient unidirectional, (2) persistent unidirectional, (3) transient bidirectional, and (4) persistent bidirectional. In our taxonomy, the mobile device can offload different portions of a program’s execution to the remote system. Therefore, preserving program state on the remote system reduces the amount of data transfer between the two ends. Allowing the remote system to communicate with the mobile device during computation enables enlarging the scope of offloadable computation and consequently avoiding the cost of fragmented offloading.

Besides defining the four strategies in a unified framework, we also empirically evaluate their limits and potentials in different networking environments. We use 14 Android applications of which seven are selected from the Google Play market to evaluate these strategies. We generate fine-grained traces of their execution and develop an optimization framework to systematically find an offloading solution for a given trace under each strategy. We formulate this problem as finding a min-cut of a graph, taking into account the constraints of the strategy and the computation and communication costs both in energy and delay. Our formulation is solvable in polynomial time in trace size, making it feasible to use in tools for partitioning mobile applications, and also enabling us to study more realistic workloads than...
past approaches that are based on integer linear programming (ILP) [7, 8, 24, 34], which takes worst-case exponential time. Our empirical evaluation yields several insights that can pave the way for practical mobile-cloud computing:

1. Formulating offloading as a min-cut problem generates the optimal offloading in under five seconds on our largest trace (432 million Java bytecode instructions) compared to over eight hours for existing ILP algorithms on the smallest trace (30 million Java bytecode instructions).

2. Our persistent bidirectional strategy yields significantly higher benefits (on average 42% more speedup and 91% more energy savings than the existing transient unidirectional strategy), delivering 2.7× speedup and 4.4× energy savings. These gains are achieved without changes to current hardware and networks.

3. Moreover, our sensitivity studies show that, as the communication latency reduces, e.g., due to advances in network technologies, our strategy leverages the benefits more effectively than the existing strategy.

2. Towards Flexible Offloading

Consider a program that reads an image from a mobile device’s camera, performs a heavy computation on that image, and outputs its result to the device’s screen. Additionally, if the computation is long-running, the program displays a progress indicator to keep the user engaged. Figure 2 captures the essence of such a program. It consists of a main loop that reads an input into variable a1, calls a function foo, and outputs variable a2. Function foo consumes the input and produces the output, using some intermediate state (a3 and a4) during a heavy computation. Additionally, foo calls either bar or taz, producing output a4 in the former case. An execution of this program on a mobile device is shown in Figure 3(a). It starts at the entry of main at t0. The first instance of foo executes from t1 to t2, and the second instance starts at t3. The first instance of foo is ideal to offload to a remote machine; by virtue of calling taz instead of bar, it represents a batch task that is both compute-intensive and does not perform I/O via sensors on the device. The resulting mobile-cloud execution is shown in Figure 3(b). Notice that data a1 must be sent over the network from the device to the remote system, and data a2, a3 must be sent back. This illustrates a fundamental tradeoff in mobile-cloud computing: the speedup of the offloaded task must offset the latency of the data transfer. This tradeoff depends on factors such as relative processor speeds and the network latency.

The mobile-cloud execution in Figure 3(b) is feasible even under a restrictive strategy that we call the transient unidirectional strategy. In this strategy, the remote system does not communi-cate with the device until the offloaded task is completed, which prevents offloading instances of foo that call bar instead of taz. Secondly, the state of the offloaded task is not preserved by the remote system, which requires re-sending data a3 in subsequent instances of foo that are offloaded to the remote system. These restrictions result in lost opportunities to offload computation.

We introduce the persistent bidirectional strategy that generalizes the transient unidirectional strategy by lifting its restrictions. It has two key features that we illustrate using the execution in Figure 3(c). First, it allows the remote system to communicate with the device while executing an offloaded task. This feature allows the first instance of foo executing on the remote system to call bar on the device, sending data a4 that bar needs to output on the device. When bar finishes, control returns to the remote system, which resumes executing the long-running task foo. Secondly, this strategy lets the remote system preserve state across offloaded tasks. This feature allows the second instance of foo running on the remote system to reuse data a3 that was written by the first instance of foo, which also executed on the remote system.

The two features of the persistent bidirectional strategy are orthogonal and complementary in avoiding fragmentation of offloaded computation. For instance, as evident from Figure 3(c), bidirectional communication enables continuity within each offloaded instance of foo, whereas persistent state enables continuity across offloaded instances of foo. This motivates two new strategies where these features are independently enabled: the transient bidirectional and persistent unidirectional strategies.

In summary, we seek to make mobile-cloud computing more widely applicable, by generalizing the current strategy of computation and communication along two dimensions. In doing so, we arrive at three new strategies, of which the persistent bidirectional strategy is the best in both theory and practice.

3. Algorithm for Optimal Trace Offloading

We seek to empirically evaluate the limits and benefits of the different offloading strategies. To this end, we develop an algorithm that, given a detailed trace of a mobile application, efficiently computes an optimal offloading solution for the trace, i.e., a solution that respects the constraints of the strategy and minimizes the
in which offloading is performed. Specified in Table 1, it includes the computation cost $R(\bar{c})$ of executing instructions $\bar{c}$ on the device, the communication cost $T(\bar{a})$ of transferring data $\bar{a}$ between the device and the remote system, and the computation cost reduction factor $S$ of the remote system over the device. For reference in later parts of this section, Table 1 also includes parameters extracted from the execution tree.

### 3.2 Min-Cut Optimization Framework

Our optimization framework takes an execution tree and analytical model as inputs, and outputs an optimal valid offloading under a given strategy. It encodes all valid offloadings and their costs as a weighted graph. The encoding differs for our four strategies as they differ in what constitutes a valid offloading. By posing the Min-Cut problem on the graph to an off-the-shelf solver, we obtain a valid offloading solution $f$ with minimal cost under the strategy, where $f(p) \in \{M,C\}$ maps each procedure call $p$ to run on either the mobile device $M$ or the remote system $C$.

We first introduce weighted graphs and Min-Cut:

#### Definition 3.1 A weighted graph is a directed graph $G = (V, E, v_s, v_t, \gamma)$ where $V$ and $E$ are the vertices and edges, $v_s, v_t \in V$ and $v_s, v_t \in V$ are special vertices called source and sink, and $\gamma \in E \times R^+$ maps each edge to a positive weight.

#### Definition 3.2 Given graph $G = (V, E, v_s, v_t, \gamma)$, an s-t cut $(V_s, V_t)$ is a partition of vertices $V$ such that $v_s \in V_s$ and $v_t \in V_t$. The Min-Cut problem is to find a s-t cut of minimum capacity, $\text{Cap}(V_s, V_t) = \sum_{(u,v) \in (V_s \times V_t) \cap E} \gamma(u,v)$.

Figures 6 and 7 show the graph encodings (on the right) for example execution trees (on the left) under the transient unidirectional and persistent bidirectional strategies, respectively. In the graphs, source vertex $C$ represents the remote system while sink vertex $M$ represents the mobile device. Each procedure call is added as a vertex. Our key insight is to generate the graph such that:

1. The minimum cut maps to an optimal offloading solution such that i) the capacity of the cut is equivalent to the solution’s total cost, ii) procedure calls in the $M$ partition ($V_I$) are mapped to run on the mobile device, and iii) procedure calls in the $C$ partition ($V_s$) are mapped to run on the remote system.
2. Any invalid offloading solution maps to a cut with infinite capacity. The mobile-only execution maps to a cut with finite capacity, which guarantees that the Min-Cut solution can never map to a cut with infinite capacity, thus ensuring that our encoding never produces an invalid offloading solution.

For brevity, we omit the formal construction and correctness proofs, and informally describe how the graph is built by encoding three aspects: the computation cost, the offloading direction constraint, and the communication cost.
Enclosing computation cost. For each procedure call $p$, we add an edge $(C, p)$ with its mobile computation cost $R(inst(p))$ as the weight. Intuitively, if the Min-Cut solution puts this vertex in the $M$ partition, $R(inst(p))$ will be charged in the capacity for running $p$ on the mobile device. Similarly, we add an edge $(p, M)$ with the remote computation cost $R(inst(p))/S$ of $p$ as the weight. If $p$ has I/O instructions as children, we assign a weight of infinity to edge $(p, M)$ to denote that it must execute locally.

In Figure 6, edge $(C, main)$ has weight $R(e1.e4)$, denoting that the cost of running main on the device is that of executing $e1$ and $e4$. But edge $(main, M)$ has infinite weight as main must run on the device since it contains I/O instructions.

Encoding offloading direction constraint. The key difference between the unidirectional and bidirectional strategies is that the bidirectional strategy allows the remote system to communicate with the device when executing an offloading task, whereas the unidirectional strategy does not. To encode this constraint of the unidirectional strategy, for each procedure call $p$, we add an edge with infinite weight from it to each of its callees.

In Figure 6, the weight of edge $(main, foo)$ is infinity, which prevents any Min-Cut solution that runs main on the remote system and foo on the mobile device.

Encoding communication cost. We discuss encoding this cost for the transient and persistent strategies separately.

Under the transient strategy, offloaded tasks do not share program state with each other. As a result, if a procedure call $p$ is chosen to be offloaded, the data that must be transferred is the set of addresses accessed by both $p$ (or its descendants) and other procedure calls. To encode this constraint, we add edges $(p, caller(p))$ and $(caller(p), p)$ with weight $T(data(p))$. The weight of edge $(p, caller(p))$ represents the communication cost if $caller(p)$ runs on the mobile device and $p$ runs on the remote system.

Likewise, the weight of edge $(caller(p), p)$ represents the communication cost if the roles of $caller(p)$ and $p$ are switched.

In Figure 6, the weight of edge $(foo, main)$ is $T(a.b)$, denoting the cost to transfer $a$ and $b$ if foo runs on the remote system and main runs on the device. The weight of edge $(main, foo)$ is infinite due to the constraint of the unidirectional strategy.

Under the persistent strategy, offloaded tasks can share state. To minimize communication cost, an address is transferred only if any of its readers is on the different end than its writer. To capture this constraint, for each address $a$, we create vertices $a_M$ and $a_C$ to represent the copies of $a$ on the mobile device and remote system respectively. For the writer of $a$, we add edges $(a_C, writer(a))$ and $(writer(a), a_M)$ with weight $T(a)$. For each reader of $a$, we add edges $(r, a_C)$ and $(a_M, r)$ with infinite weight where $r \in readers(a)$.

In Figure 7, address $a$ is written by foo and read by main and bar. Vertices $a_M$, $a_C$, and corresponding edges are created following the above process. There are two cases depending on whether foo runs on the remote system or the device. When foo runs on the remote system, we need to charge the cost of transferring a from the remote system to the device if main or bar runs on the device. When main or bar runs on the device, the Min-Cut algorithm will put $a_M$ into the $M$ partition to avoid infinite capacity incurred by $(a_M, main)$ or $(a_M, bar)$. As a result, $T(a)$ will be charged in the capacity if foo runs on the remote system. The case when foo runs on the device is similar where the goal is to charge the cost of transferring a from the device to the remote system if main or bar runs on the remote system.

Putting it all together. Following the above steps, we obtain the weighted graphs shown in Figure 6 and Figure 7. The graph in Figure 6 has two cuts of finite capacities:

$$\text{Cap}(\{C\}, \{M, main, foo\}) = R(e1.e4) + R(e2.e3)$$

$$\text{Cap}(\{C, foo\}, \{M, main\}) = R(e1.e4) + R(e2.e3)/S + T(a.b)$$

They represent two offloadings that are valid under the transient unidirectional strategy with the capacities as the total costs. When $R(e2.e3)/S + T(a.b) < R(e2.e3)$, the Min-Cut algorithm se-

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(e_1, ..., e_n)$</td>
<td>Computation cost of executing sequence of instructions $e_1, ..., e_n$ on the mobile device.</td>
<td></td>
</tr>
<tr>
<td>$T(a_1, ..., a_n)$</td>
<td>Communication cost of transferring set of data $a_1, ..., a_n$ between the mobile device and remote system.</td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td>Computation cost reduction factor of the remote system over the mobile device.</td>
<td></td>
</tr>
</tbody>
</table>

**Table 1: Model parameters and parts on which they depend: trace $\Sigma$, mobile device $\square$, remote system $\bigcirc$, or network $\circ$.**
lects the latter cut as the solution, which offloads foo. Otherwise, it selects the former cut, which does not offload anything.

For the graph in Figure 7, we show two possible cuts, labeled X and Y. Cut X represents the offloading in which foo and bar are offloaded while main runs locally. The capacity of the cut represents the total cost of the offloading:

$$\text{Cap}(X) = T(a) + R(e1)/S + R(e2) + R(e3)/S$$

Cut Y represents the offloading in which only foo runs remotely, and the correspondent capacity is

$$\text{Cap}(Y) = T(a) + R(e1)/S + R(e2) + R(e3)$$

Note that although there are multiple reads to address a, the communication cost is charged only once.

To obtain the Min-Cut solution, our framework uses the pseudoflow algorithm [16], which solves the problem in time $O(|V||E|\log|V|)$. The complexity can be further reduced to $O(|V||E|)$ using recent, more sophisticated algorithms [25].

### 3.3 Execution Tree With Offloading

Our final output is the input execution tree augmented with offloading. We define the augmented execution tree by extending the syntax of procedures in Figure 5 as follows:

$$p ::= \ldots | \text{suspend}(A_1) p \text{resume}(A_2) \ [\text{where} A_1, A_2 \subseteq A]$$

That is, the augmented tree has two additional kinds of instructions: suspend($A_1$), which transfers control and sets of addresses $A_1$ from one end to the other as in a remote procedure call (RPC); and resume($A_2$), which returns control and set of addresses $A_2$ as in a return from an RPC.

To generate an augmented execution tree from the Min-Cut solution $f$, for any $p$ such that $f(p) \neq f(\text{caller}(p))$, we insert a suspend($A_1$) before $p$ and a resume($A_2$) after $p$, where $A_1$ and $A_2$ are generated as follows. In the transient strategy, to minimize the communication cost, we construct $A_1$ as all addresses in data($p$) that are read by $p$ (or its descendants), and $A_2$ as all addresses in data($p$) that are written by $p$ (or its descendants). In the persistent strategy, a read of address $a$ is valid if $a$ is written on the same end or is transferred from the other end in a prior suspend()resume(). To satisfy this constraint and minimize the communication cost, for any address that is accessed by both ends, we include it in the $A_1$/$A_2$ of the suspend()resume() immediately after it is written.

### 4. Experimental Methodology

We evaluate the four strategies by applying our algorithm in Section 3 to 14 Android applications. This section describes the methodology used in our evaluation, especially how we generate the input to the algorithm, which consists of a detailed execution trace and an analytical model of the computation and communication costs. To collect traces of Android applications, we built a trace generation framework by instrumenting the Java virtual machine on Android. To model the computation and communication costs in terms of performance and energy, we built a trace-based simulator. Trace-based simulation is a common practice in evaluating complex systems as ours to study first-order trends. To estimate both these costs in a realistic setting, we incorporate the results from measurements on a Samsung Galaxy S2 smartphone. In particular, for performance, we incorporate the runtime measurements by invoking a Linux API with nanosecond resolution; for energy, we measure the power consumption by connecting the smartphone to an external power meter [1] with a resolution of 0.001 mW. By feeding a trace collected using the above framework and the results of the measurements to the simulator, we evaluate the performance and energy gains of a given offloading strategy.

#### 4.1 Benchmarks

The 14 Android applications are shown in Table 2. Seven of them are selected from the Google Play market, three of which are used in prior work [14]. The rest of the apps in [14] have hardware-specific dependencies or are no longer available. We added four more real-world Android apps: the ametreo navigation app, the BootCV video processing app, the ImageJ image processing app, and the ZXing barcode scanning app. We also ported seven desktop applications to Android, representing future apps that could be enabled on mobile devices by offloading. These include five applications from the DaCapo [6] suite. The rest of the DaCapo applications depend on JDK features that are not available on Android. We also ported the Sat4j SAT solver and the Rhino Javascript engine. We did not reject benchmarks because of shortcomings in performance or energy benefits from any of the four strategies. These benchmarks represent a diverse domain of applications including navigation, video processing, image processing, games, simulation, program optimization, document processing, web browsing, and constraint solvers.

#### 4.2 Execution Trace Generation

We generate detailed execution traces for each application at the Java bytecode level. These traces are parsed into execution trees (Section 3.1). We discuss how we generate such traces below.

**Trace Format.** Our trace contains the following information: (1) number of executed instructions per instruction type, (2) method entry and exit events, (3) Java heap reads and writes with their addresses and sizes, and (4) markers that identify native methods. We use the number of instructions to estimate the computation time and energy, the method call information to identify offloading boundaries, and the reads and writes to estimate the communication time and energy.

**Trace generation.** To collect traces, we instrument the Dalvik VM of Android 4.3 (JellyBean). We also instrument exception handling and thread IDs to correctly capture method entry and exit events. We verify offline that these events match perfectly in all collected traces. Naively instrumenting Android applications can change their observable behavior by violating timing assumptions in the Android framework. We implement several optimizations to reduce instrumentation overhead and ensure that it does not alter the observable behavior of our applications.

**Handling native code.** Many Android applications use native code to improve performance and execute system-specific operations. We instrument the Java Native Interface (JNI) bridge of
Dalvik VM to capture data flow and control flow between managed Java code and native code, and mark the native methods in the execution traces. We conservatively mark native methods performing system calls unoffloadable as system calls often access physical components in the mobile device. We describe how we identify these native methods below.

**Identifying pinned code.** To conservatively mark pinned (un-offloadable) native methods, we identify the set of offloadable native methods. We use common offloadable native methods identified by COMET [14] as our initial set. We manually inspect source code of native methods not in this set and add them if we can conclude they do not invoke any system call. We conservatively mark native methods without source code as pinned.

**Trace collection platform.** We collect traces of all 14 applications by running the instrumented Android-x86 image inside QEMU with KVM support. QEMU runs on an Opteron 6220 processor (16 cores) with 128GB memory. The host operating system is Debian Linux version 6 with 2.6.32 kernel.

### 4.3 Modeling Computation and Communication Costs

To enable our optimization framework to compare the performance and energy gains of the four strategies, we build a trace-based simulator with the traces generated above as inputs. This simulator estimates computation and communication costs both in terms of performance and energy. The simulator uses models to estimate the following components: (1) Computation cost of executing different sequences of instructions on the device; (2) Communication cost of transferring different sizes of data between the device and remote system; and (3) Computation cost reduction factor of the remote system over the device. We next discuss the performance and energy models separately.

#### 4.3.1 Performance Model

To estimate the above three components with performance as the cost metric, our trace-based simulator needs models for application runtime, communication latency, and cloud speedup. We model cloud speedup based on prior work [10]. We build the other two models using real measurements.

**Measurement setup.** To build the models for application runtime and communication latency, we collect data on a Samsung Galaxy S2 phone. For application runtime, we measure the running time of every application in our benchmark suite; for communication latency, we measure the time to exchange different sizes of data between the phone and a server. All measurements are done using the `clock_gettime` function in Linux which provides nanosecond resolution in Linux 2.6.12 and higher.

**Application runtime.** We use a linear model to estimate the application runtime of a given sequence of instructions. We divide the 246 types of Dalvik bytecode instructions into 13 categories and use the number of instructions in each category as the features in our model. We collect the data needed to build the model in two runs of each application with the same input. In the first run, we measure the runtime of the application for the given input. In the second run, we use lightweight instrumentation to collect the number of executed instructions of each type. This process is repeated hundred times on each application using various inputs. Figure 8 shows the measured runtime (x-axis) and estimated runtime (y-axis) of four applications. Every point in each chart represents a different run. We observe a tight correlation between the estimated and the measured runtime. Also, note that our model is conservative as it underestimates the runtime for most runs. This in turn makes the trace-based simulator underestimate the performance gain of a given strategy.
Communication latency. We estimate communication latency using a model based on measured link latency (link latency) and bandwidth (bandwidth). As per this model, the time to transfer $x$ bytes of data can be calculated as $\text{link latency} + \frac{x}{\text{bandwidth}}$. To construct this model, we measure the time to exchange different sizes of data between a Samsung Galaxy S2 phone and a server. The size of the transferred data ranges from 1 byte to 100KB. For each particular size, we average the transfer time of 40 measurements. Given these measurements, we conservatively obtain the link latency and bandwidth for the model. In our study, we use three network connectivity settings and construct a separate model for each of them. These three settings correspond to a Samsung Galaxy S2 phone connecting to (1) a local server on the university campus through Wi-Fi (campus server Wi-Fi); (2) an Amazon EC2 server through Wi-Fi (remote cloud Wi-Fi); and (3) an Amazon EC2 server through AT&T LTE (remote cloud LTE). Figure 9 shows the average transfer time for different sizes of data under the campus server Wi-Fi setting. We obtain (1.5ms, 5MB/s) as (link latency, bandwidth) for the corresponding model. The solid line in Figure 9 shows the transfer time estimated by the model. Note that our model is conservative as it overestimates the communication latency for most sizes of data. Similarly, we obtain (8ms, 3MB/s) and (30ms, 3MB/s) as (link latency, bandwidth) for the remote cloud Wi-Fi and the remote cloud LTE network settings respectively.

Cloud speedup factor. We conservatively use the speedup factor of $10\times$ for the remote system from prior work that measures the performance of high-performance and low-power processors [10]. Their measurements show that the Nehalem-based Intel Core i7-965 (130 W chip power budget) is around $12\times$ faster than the Intel Atom ZS20 (2.2 W chip power budget). They do not consider lower power architectures such as ARM which are common in mobile devices, making our estimate conservative.

Application speedup. To estimate the application speedup due to an offloading strategy, the trace generated for an application is fed to our optimization framework along with the trace-based simulator, which models the computation and communication cost. The framework produces an offloading solution considering the computation and communication tradeoffs as described in Section 3. We estimate the speedup as follows.

$$\text{Speedup} = \frac{\text{Execution Time on Mobile w/o Offloading}}{\text{Execution Time on Mobile w Offloading}}$$

In this formulation, for any given method invocation $m \in M$ in a trace, our simulator estimates $T_{\text{Local}}(m)$, which is the local execution time of the method instance. The simulator also estimates $T_{\text{Remote}}(m)$ and $T_{\text{Comm}}(m)$. These values represent the time to execute the method instance remotely and the time to communicate the necessary data for the method instance to run on the remote system. In the formulation, $M_{\text{Local}}$ represents the subset of method instances that run locally and the $M_{\text{Remote}}$ represents the subset of method instances that run on the remote system. The union of $M_{\text{Local}}$ and $M_{\text{Remote}}$ is $M$ and these subsets are the output of the optimization framework discussed in Section 3.

4.3.2 Energy Model

To estimate the computation and communication cost with energy as the cost metric, our trace-based simulator needs the related energy models. As energy is $\text{Power} \times \text{Time}$, it suffices to build the corresponding power models and use them in conjunction with the performance models from Section 4.3.1. Our power model estimates the power consumption of the mobile device when it is in the following states:

- $P_{\text{Local}}$: the device is executing the application.
- $P_{\text{Comm}}$: the device is exchanging data with the remote system.
- $P_{\text{Wait}}$: the device is waiting for the offloaded execution to return from the remote system.

Note that our study only concerns energy reduction on the device, and so our model only estimates the device power.

Measurement setup. To collect data for building the power model, we measure the power consumption of a Samsung Galaxy S2 phone using a Monsoon Power Meter [1]. Following its manual [2], we connect the meter to the phone such that it bypasses the battery, supplies a stable voltage of 3.70 V, and samples the power of the phone with a resolution of 0.001 mW every 0.2 ms. The meter is also connected to a Windows laptop to read the sampled data.

Computation and communication power. To estimate the power consumption of the mobile device in the three states described above, we run test apps that exercise the device in the corresponding states. We collect the power measurements by running each test app for one minute. For all measurements, we keep the screen on as mobile apps are typically interactive. We also kill all background user apps before running the test apps to avoid them from influencing the measured power consumption.

For $P_{\text{Local}}$, we measure the power consumption of the mobile device by running the ChessDroid app with the device playing against itself. We use this app as it is computationally intensive and it does not access the network.

For $P_{\text{Comm}}$, we measure the power consumption of the mobile device by running an app which exchanges fixed size (1 KB) TCP packets continuously between the mobile device and the remote system. Our methodology follows the prior work...
on energy modeling of network interface [36]. We measure $P_{Comm}$ under our three network settings separately as power consumption depends on the specific network interface (Wi-Fi or LTE) and the data transfer rate [20, 26, 36].

For $P_{Wait}$, we measure the power consumption of the mobile device with no user application running.

Figure 10 shows the measured values for $P_{Local}$, $P_{Comm}$ under the campus server Wi-Fi setting, and $P_{Wait}$. From these values, we extract $P_{Local} = 2.7W$, $P_{Comm} = 1.1W$, and $P_{Wait} = 0.75W$, which are shown as solid lines in the corresponding charts. These values are conservative as they underestimate the local computation cost, and overestimate the communication cost and remote computation cost.

**Application energy reduction.** The workflow for estimating energy reduction is similar to that for estimating speedup discussed earlier. The application trace is fed to the optimization framework along with the trace-based simulator, which models the computation and communication costs. The framework produces the offloading solution that identifies the offloaded method instances ($M_{Remote}$) and local method instances ($M_{Local}$). Using this information, we estimate the application energy reduction as:

$$\text{Energy Reduction} = \frac{\text{Mobile Energy Dissipation w/o Offloading}}{\text{Mobile Energy Dissipation w Offloading}} = \frac{\sum_{m \in M} E_{Local}(m) + \sum_{m \in M_{Remote}} (E_{Remote}(m) + E_{Comm}(m))}{\sum_{m \in M_{Remote}} E_{Remote}(m) + E_{Comm}(m)}$$

where $E_{Local}(m) = T_{Local}(m) \times P_{Local}$ and $E_{Remote}(m) = T_{Remote}(m) \times P_{Wait}$ and $E_{Comm}(m) = T_{Comm}(m) \times P_{Comm}$. $T_{Local}(m)$, $T_{Remote}(m)$, and $T_{Comm}(m)$ are as defined in the formulation used to estimate speedup, while $P_{Local}$, $P_{Comm}$ and $P_{Wait}$ are defined at the beginning of this section.

5. Experimental Results

Our evaluation consists of three parts. The first part leverages measurements on a specific device and network (Section 5.1). As offloading applies to a variety of computing and networking contexts, we do sensitivity studies that extend our analysis to other application inputs and network settings (Section 5.2). Finally, we study the feasibility of our offloading strategies by conducting a qualitative evaluation and case study (Section 5.3).

We use geometric mean to calculate the average speedup and average energy reduction. Moreover, for a fair comparison of the different strategies, we use the optimal offloading solution generated by our Min-Cut framework for each strategy.

5.1 Measured Results

We evaluate the speedup and energy reduction of the four strategies based on measurements using a Samsung Galaxy S2 phone under the campus server Wi-Fi network setting. We first compare the persistent bidirectional strategy and the transient unidirectional strategy using speedup as the optimization objective. We then compare the results for all four strategies. Lastly, we study the benefits when the objective is to maximize energy reduction.
Speedup and energy reduction. Figure 11(a) shows speedup and Figure 11(b) shows energy reduction for the transient unidirectional and persistent bidirectional strategies. In addition, the last bar represents the idealized benefit when the communication delay and energy is artificially set to zero. The baseline is execution on the mobile device without offloading and the optimization objective for the Min-Cut algorithm is to maximize speedup. The network setting is campus server Wi-Fi with link latency of 1.5 ms and the bandwidth of 5 MB/s. This network provides the fastest connectivity between the mobile device and the remote system. We explore the other slower settings below. As Figure 11(a) shows, the persistent bidirectional strategy enables 2.7× average speedup, while the existing transient unidirectional strategy yields 1.9× average speedup. The speedup is as high as 8.4× in the case of poker for both strategies. As Figure 11(b) illustrates, the persistent bidirectional strategy achieves 4.4× average energy reduction, while the existing transient unidirectional strategy results in 2.3× average energy reduction. The benefit is as high as 22× in the case of poker. The fact that more than 98% of poker is offloadable justifies such large savings (see Figure 11(c)).

On average, the persistent bidirectional strategy yields 42% more speedup and 91% more energy reduction than the existing transient unidirectional strategy. The extra benefits do not require changes to the existing hardware.

To better understand these gains, Figure 11(c) shows the percentage of execution that is offloadable. Based on Amdahl’s Law, the percentage of offloadable work has a direct correlation with the benefits. As expected, in the ideal case where the communication cost is zero, each strategy offloads the most amount of work, which is on average 89%, which yields ideal average speedup of 5.7× and ideal average energy reduction of 10×. Also, on average, transient unidirectional offloads 45% whereas our proposed persistent bidirectional strategy offloads 75% of the execution; that is only 14% less than the ideal case and 30% more than the existing strategy. Despite being close to the offloadable fraction in the ideal case, the benefits are still far from the ideal case due to communication cost, which we analyze in Section 5.2.

Comparing offloading strategies. Figure 12 shows the average speedup and the average offloadable fraction of computation across all four strategies. The trends are similar for energy reduction. As Figure 12(a) shows, the transient bidirectional strategy gives 26% improvement in speedup over the existing strategy, and the persistent bidirectional strategy further improves it by 16%.

These results suggest that supporting bidirectional communication between the mobile device and the remote system has a higher contribution to the benefits from our proposed approach compared to supporting persistence on the remote system. However, both aspects contribute significantly.

The main source of the benefits is the increased fraction of computation that can be offloaded. Comparing Figure 12(a) and (b) confirms the strong correlation between the benefits and the fraction of computation that is offloadable.

Our proposed bidirectional strategy significantly enlarges the scope of offloading by allowing the offloaded task to communicate back with the mobile device during computation.

In fact, 9 of the 14 benchmarks see significantly larger scope for offloading with the bidirectional strategies compared to the unidirectional cases. These are typically the applications that perform frequent I/O (e.g., accessing flash, GPS, or display) while computing. We inspect ametro more closely in Section 5.3.

Optimizing for energy. Figure 13 shows the energy reduction when the Min-Cut algorithm’s objective is to maximize energy savings. The persistent bidirectional strategy enables 4.7× energy reduction, while the existing transient unidirectional strategy yields 2.4×. The trends are similar for speedup. Comparing the results from Figure 13 and Figure 11(b) shows that the average energy reduction is 7% higher when the objective is energy. Our investigation also shows that the generated offloading is significantly different for several benchmarks. For instance, Figure 11 shows that the persistent bidirectional strategy enables 2.2× energy reduction for BoofCV when the objective is speedup, whereas Figure 13 shows that when the objective is energy, the energy reduction for this application further improves to 3.8× under the same strategy. On the other hand, the speedup drops from 2.2× to 1.6×.

For several applications, when the objective changes from speedup to energy reduction, our optimization framework generates a different scheme that provides significantly different benefits. This result shows the ability of our software-only solution in trading a fraction of the performance gains for energy.

5.2 Sensitivity Studies

We study the sensitivity of our results to different network settings. In particular, we evaluate the performance and energy gain under the three representative network scenarios identified in Section 4.3.1. We further study the effects of each component of network connectivity separately, including the communication power, link latency, and bandwidth. Also, we study the sensitivity of our results to the inputs used to generate the traces.
**Benefits with different network settings.** We examine the benefits of all four strategies with different network settings, including campus server Wi-Fi, remote cloud Wi-Fi, and remote cloud LTE. The fastest setting is the campus server Wi-Fi and the slowest one is remote cloud LTE. Figure 14 shows the average speedup and energy reduction across all strategies across these network settings. The optimization objective is speedup. As expected, the gains drop as the connection speed drops. Moreover, as Figure 14 shows, under the fastest network (campus server Wi-Fi), our approach (persistent bidirectional) provides significantly higher benefits than the existing strategy (transient unidirectional). However, the benefits are negligible (less than 8%) with the slowest network setting (remote cloud LTE).

These results confirm that our approach performs as well as the existing strategy with slow connectivity while it provides higher benefits as the connectivity improves.

**Sensitivity to the power consumption of communication.** Figure 15 shows the average speedup and average energy reduction when the communication power of the mobile device is set to 0.55 W, 1.1 W and 2.2 W. In these experiments, the computation power and waiting power of the device are uniformly set to 2.7 W and 0.75 W respectively, while the link latency and the bandwidth of the network are set to 1.5 ms and 5 MB/s. As expected, the increase in the communication power consumption results in lower benefits, but these benefits are still more than 2.5× in both speedup and energy reduction. The bidirectional approaches consistently deliver higher benefits. By comparing the persistent bidirectional to the transient unidirectional strategy, we observe that even with high communication power of 2.2 W, the persistent bidirectional approach provides 71% higher energy reduction. The gains increase to 125% when the communication power consumption drops to 0.55 W. On the other hand, the speedup varies modestly across the three communication power settings. The higher benefits in energy reduction are mainly due to the enlarged scope of offloading in bidirectional approaches.

These results suggest that the bidirectional strategies can exploit the lower power communication links more effectively than the unidirectional strategies.

**Effects of network link latency.** First, we perform a sweep on link latency with Wi-Fi connectivity. Figure 16 shows the average speedup for all strategies across a range of link latencies from 50 ms to 0.5 ms. In these experiments, we keep the bandwidth fixed to 5 MB/s and communication power fixed to 1.1 W. As Figure 16(a) illustrates, the benefits decrease as the network latency increases, which is expected. However, interestingly, the benefit gap between our strategy and the existing transient unidirectional strategy grows significantly as the network latency decreases. With the 50 ms link latency, our strategy provides only 2% improvement over transient unidirectional. When the link latency decreases to 0.5 ms, this improvement grows to 68%. The trends are similar for energy. Another observation is that unidirectional strategies exploit the reduced latency with significantly shallower slope than the bidirectional strategies.

As the network latency decreases (e.g., due to advances in the networking technologies), our proposed strategy becomes more effective and yields larger benefits in speedup and energy efficiency. The existing strategy, on the other hand, exploits the reduced latency at a lower rate.

**Effects of network bandwidth.** Figure 16(b) shows the average speedup for all strategies across a range of bandwidth. In these experiments, the link latency is fixed to 1.5 ms and communication power is fixed to 1.1 W. When the bandwidth increases from 0.1 MB/s to 10 MB/s, the speedup of our strategy increases by 58% while the speedup of the existing strategy increases by 28%. The benefits gap between these is smaller than the case where the communication latency improves. The lower benefits from increasing bandwidth compared to improved latency is due to latency-sensitivity of the apps in our workload pool.

In general, the bidirectional strategies benefit more from improved connectivity while latency is more critical for the applications in our workload pool.

**Sensitivity to application inputs.** We compare the results of our strategies on traces generated by different inputs to the same application. For this purpose, we generate three different traces for
We study the practicality of our offloading strategies by conducting a qualitative evaluation and a case study.

**Detailed statistics.** To better understand the behavior of the four strategies, we provide more detailed statistics. Figure 17(a) shows the average number of pairs of suspend() and resume() instructions introduced by the different strategies. Each pair corresponds to a remote invocation either from the device to the remote system or vice versa. As expected, the `persistent bidirectional` strategy enables more method invocations to be offloaded. However, as Figure 17(b) shows, these remote invocations only map to a small set of unique methods in the application’s source code. In particular, the 473 remote invocations under the `persistent bidirectional` strategy only correspond to 18 methods. This is in line with the commonly observed 80-20 rule in program optimization: 80% of the execution time is spent in 20% of the code. Finally, Figure 17(c) shows the average amount of total data transferred under the different strategies. As the flexibility of our strategies increases, more data is transferred. However, recollect from Figure 12(b) that the offloadable fraction of the execution is also enlarged by the increasing flexibility. Consequently, the reduced computation cost offsets the increase in the communication cost.

**Case study.** To understand why a certain application benefits from our strategies, we closely examine ametro, which gains the highest benefits. A navigation app, ametro calculates the route between two subway stations in a city. The running time of each run of ametro is dominated by three methods: loadView, performFiltering and findRoute. Figure 18(a) shows its execution on the mobile device with no offloading. For clarity, we elide other method invocations and use solid boxes to represent code fragments that perform I/O. In this application, loadView loads the compressed map file from the disk and decompresses it. Then, the performFiltering method searches through the map data and suggests a list of stations based on the user’s query, which contains a source and a destination station. As the user enters the station names letter by letter, performFiltering is invoked repeatedly to make these suggestions. The findRoute method calculates the actual route after the user confirms the source and destination stations from the lists returned by performFiltering.

After inspecting the source code and the collected execution trace, we find that performFiltering takes more time than loadView and findRoute. In our collected trace, performFiltering is invoked 26 times and these invocations take 1.5 seconds in total. The loadView and findRoute methods are invoked only once and they take 0.3 second and 0.01 second, respectively. By applying our Min-Cut optimization framework, we generate offloading solutions under the different strategies. We illustrate the solution for the `transient unidirectional` strategy in Figure 18(b) and the solution for the `persistent bidirectional` strategy in Figure 18(c).
As Figure 18(b) shows, under the transient unidirectional strategy, only invocations of performFilter are offloaded to the remote system. For each invocation, 17 KB of map information is repeatedly uploaded to the remote system and 100 bytes of search results are returned to the device. Under the transient unidirectional strategy, loadView is not offloaded to the remote system as it invokes subroutines that perform I/O. Also, findRoute is not offloaded as the communication cost of offloading it to the remote system would offset the improvement in computation cost. As a result, the transient unidirectional strategy yields a $2.4 \times$ speedup and $3.1 \times$ energy reduction on anemetro.

By taking advantage of both the bidirectional and persistent aspects of our offloading strategy, it improves the benefits to $3.6 \times$ speedup and $5.0 \times$ energy reduction. As Figure 18(c) shows, the bidirectional aspect of our strategy enables offloading methods such as loadView that even perform I/O on the device. That is because the remote system is enabled to invoke a procedure on the device. Furthermore, the persistent aspect of our strategy reduces the data transfer between the two ends. Using our strategy, the offloaded loadView is able to preserve the map information alleviating the need to communicate this information every time a invocation of performFilter is offloaded. This case study suggests that the bidirectional aspect of our strategy can enlarge the offloading scope, while the persistent aspect can reduce the communication cost.

5.4 Discussion

We believe that the persistent bidirectional offloading strategy can be effectively integrated with existing and forthcoming programming models, runtime systems, and connected devices. We discuss some aspects of this integration. From the programming model standpoint, the unidirectional feature is similar to subroutines whereas the bidirectional feature is analogous to co-routines [18]. The persistent feature can be enabled when the programming model supports heap sharing between different offloading sessions. Our preliminary prototyping efforts show that these features can be implemented using Remote Procedure Calls (RPC). Using RPC is commensurate with other existing offloading systems [7, 8]. Moreover, the proposed strategies can be integrated with offline [5, 13] or online [7, 8] subsystems that monitor and control offloading. A growing body of recent work addresses reducing the communication latency at the level of the operating system [27] and networking hardware [11, 33]. As discussed in Section 5.2, the persistent bidirectional strategy can leverage these advances more effectively than the existing strategy. We did not explore other techniques like concurrent execution and weak consistency for offloading. These techniques are complementary to our approaches. Studying the benefits of these techniques in conjunction with our offloading strategies is an interesting future work.

6. Related Work

Cyber foraging [4, 29] was proposed a decade ago to use external resources to augment mobile devices in terms of performance, storage, and battery life. A significant body of work [12, 15, 21, 28, 30, 37] has since addressed the design and implementation of mobile-cloud computing systems, which motivate our approach. CloneCloud [7] and MAUI [8] are most closely related to our approach. Both perform procedure-level offloading using explicit communication (as opposed to distributed shared memory, discussed below), and both make offloading decisions dynamically guided by online profiling. Both perform transient unidirectional offloading and encode the optimal offloading problem using an ILP formulation that aims to balance computation and communication costs based on execution history. Later works [3, 19, 38] follow similar designs.

COMET [14] leverages distributed shared memory for offloading. COMET can offload a single thread at any program point and perform communication at synchronization operations. COMET’s offloading strategy is similar to our approach in that it allows communication between the device and the remote system during offloading. However, COMET immediately terminates offloading when the offloaded thread needs to access resources local to the device, and thus can suffer from fragmented offloading for applications performing frequent I/O.

Language features have been proposed to facilitate offloading. Spectra [13] and Chroma [5] rely on developers to pre-partition their programs and select an offloading scheme on the fly. Hyrax [22] ports Hadoop to the Android platform. Cuckoo [17] is an offloading framework on Android that makes coarse-grained offloading decisions based on heuristics. All these works apply transient unidirectional offloading. Sapphire [35] is a distributed programming platform which provides higher-level object-oriented abstractions for writing mobile-cloud applications.

While our work assumes stable connectivity between the mobile device and the remote system, others address intermittent connectivity issues [9, 31, 32]. In addition, [32] considers the economic costs of using network and cloud resources, besides performance and energy. These are complementary to our approach that relaxes the communication and persistence restrictions of the previous approaches.

Most existing approaches use ILP to find the optimal offloading solution [7, 8, 24, 34]. An exception is [23] which uses MinCut. Their graph encoding and optimization objective are radically different from ours: they aim to partition the classes in a program into groups to minimize interactions between these groups, whereas we seek a fine-grained bipartition of a program trace that maximizes performance or energy benefits of offloading.

7. Conclusion

As the demand for computation on mobile devices increases, there is a timely need for techniques that improve their performance and efficiency. Mobile-cloud computing aims to address this need by leveraging the ubiquitous connectivity and availability of higher performance remote systems. Existing strategies that offload parts of the computation to remote systems are restricted in both computation and communication schemes. This paper defined and explored novel strategies that relax these restrictions along the two axes of remote state and communication pattern.
We developed a unified optimization framework to find an offloading solution for a given program trace under each strategy. By allowing more flexibility in communication and preserving state on the remote system, our proposed strategy achieves $2.7 \times$ speedup and $4.4 \times$ energy reduction. Our evaluation yields new insights into programming models, runtime systems, and hardware policies that could pave the way to make mobile-cloud computing widely applicable.

References


[34] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff. Tapping into the fountain of CPUs: on operating system support for programmable devices. In ASPLOS, 2008.

