Power-Efficient and Fault-Tolerant Distributed Mobile Execution

Young-Woo Kwon and Eli Tilevich
Dept. of Computer Science
Virginia Tech
Blacksburg, VA 24060
Email: {ywkwon,tilevich}@cs.vt.edu

***Abstract*—Although battery capacities keep increasing, the execution demands of modern mobile devices continue to outstrip their battery lives. As a result, battery life is bound to remain a key constraining factor in the design of mobile applications. To save battery power, mobile applications are often partitioned to offload parts of their execution to a remote server. However, partitioning an application renders it unusable in the face of network outages. In this paper, we present a novel approach that reduces the power consumption of mobile applications through server offloading without partitioning. The functionality that consumes power heavily is executed in the cloud, with the program's state checkpointed and transferred across the mobile device and the cloud. Our approach is portable, as it introduces the offloading functionality through bytecode enhancement, without any changes to the runtime system. The checkpointed state's size is minimized through program analysis. In the case of a network outage, the offloading interrupts and the application reverts to executing locally from the latest checkpoint. Our case studies demonstrate how our approach can reduce power consumption for third-party Android applications. Transformed through our approach, the applications consume between 30% and 60% fewer Joules than their original versions. Our results indicate that portable offloading can improve the battery life of modern mobile applications while maintaining their resilience to network outages.**

I. Introduction

Today's mobile devices feature hardware capacities that often surpass those of the desktops from the recent past. Multicore CPUs, large RAMs, high resolution displays, fast cellular networks—all are becoming standard for the majority of today's smartphones, tablets, and e-readers. These powerful hardware facilities enable mobile applications that are increasingly complex in terms of their computation and communication patterns. Unfortunately, battery capacities have become a major constraint of modern distributed mobile execution.

Indeed, battery capacities are known to increase quite moderately [1]. Mobile application developers must remain mindful of how computation- and communication-intensive pieces of functionality affect the overall battery life. As a result, power consumption is not only a major resource constraint for modern mobile devices, but it also impedes the mobile programmer's creativity and productivity.

Distributed execution has been proposed as a means to reduce power consumption in mobile applications [2], [3], [4]. These solutions partition a mobile application into local and remote parts, so that power intensive functionality is executed remotely at a server. Because modern mobile devices are network-enabled, offloading functionality to a remote server presents a promising avenue for reducing power consumption. Unfortunately, this optimization also makes the application vulnerable to network outages, as mobile networks are characterized by high volatility.

In this paper, we present a novel approach that combines the advantages of the prior state of the art in partitioning mobile applications with the goal of reducing their power consumption. In particular, our approach offloads power-intensive pieces of functionality to a remote server, similarly to CloneCloud [3], while using checkpointing to synchronize the state between the client and server computations, similarly to MAUI [2].

A key difference of our approach is that it can execute power-intensive functionality on the server without having to partition the application. Instead, it efficiently replicates state to switch between local and remote executions, both to reduce client power consumption and to tolerate network outages. When the network is operational, power-intensive functionality is offloaded to the server by transferring only the program's state needed for the remote execution. Efficient checkpointing synchronizes local and remote executions. If the network becomes disconnected during the offloading, the remote execution is redirected back to the mobile device. Thus, network outages only inhibit power optimization rather than rendering the application unusable. Finally, our approach is portable, as it relies on bytecode enhancement, with the enhanced mobile applications executing on unmodified runtime systems.

Hence, our approach matches the power consumption benefits of the partitioning-based approaches, while keeping mobile applications resilient in the presence of network disconnections. Because our approach heavily relies on checkpointing, we employ sophisticated program analysis to reduce the amount of the checkpointed state that must be transferred across the network. The necessity to transfer large data volumes across the network can quickly negate the power consumption benefits afforded by remote offloading. The research literature shows that the average size of a Java heap commonly exceeds hundreds of MBs [5]. To avoid

having to checkpoint the entire heap, our approach leverages forward dataflow and side effect analyses to reduce the checkpointed state by orders of magnitude, thus rendering state transfer practical for power optimization.

Thus, our approach enables execution offloading as a means of reducing power consumption of mobile applications, without sacrificing performance. Furthermore, our experimental evaluation sheds light on the following fundamental questions:

1) How can one determine optimal execution units that can be offloaded to a remote server to reduce power consumption?
2) How can one reduce the checkpointed state's size to make state transfer a pragmatic power consumption optimization technique?
3) How can a system that relies on remote execution to reduce power consumption remain operational in the presence of network outages?

In our experiments, we have applied our approach to optimize power consumption of five third-party, real-world Android applications. Transformed through our approach, four of these subject applications reduced their power consumption; and one application maintained its original power consumption. All the subject applications maintained their original performance characteristics. To help the programmer determine when our approach can optimize power consumption, we also introduce a program analysis that can identify those application patterns that render our approach inapplicable.

Based on our results, the technical contributions of this paper are as follows:

1) **Fault-tolerant execution offloading**—an effective power consumption optimization approach that efficiently and fault-tolerantly synchronizes execution state between the mobile device and remote server to provide power-efficient and reliable mobile execution.
2) **Execution offloading analysis**—a new static program analysis technique that safely identifies the exact state that must be transferred across the network during execution offloading.
3) **Execution offloading benefit analysis**—a heuristic that can guide the programmer if execution offloading can reduce power consumption.
4) **An evaluation of execution offloading**—an empirical evaluation of execution offloading using five real-world, third-party applications.

The rest of this paper is structured as follows. Section II motivates our approach. Section III introduces the technologies we used to implement our approach. Section IV describes our approach and reference implementation. Section V evaluates our approach with third-party applications. Section VI compares our approach to the existing state of the art. Finally, Section VII presents concluding remarks.

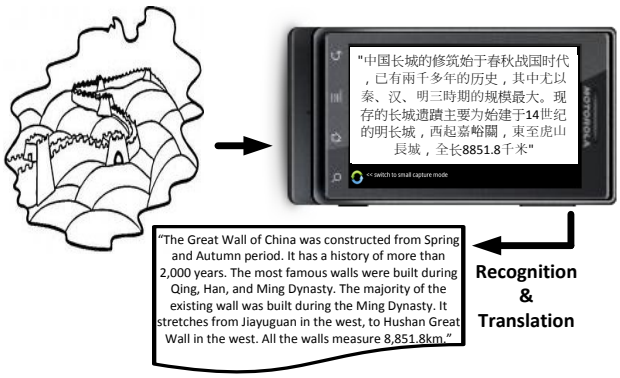

Figure 1. A Mezzofanti use case.

II. Motivating Example

Consider Mezzofanti—a third-party, augmented reality application that runs on the Android platform. This application guides travelers visiting foreign countries. Language differences is a major source of inconvenience when traveling internationally, particularly to the locales that use writing systems different from that of the traveler (e.g., English speakers visiting China). Mezzofanti enables the traveler to use a camera to capture the image of printed text in any language and obtain a free translation. The use case depicted in Figure 1 shows how Mezzofanti can decipher a Chinese language sign displayed somewhere at the Great Wall of China. To that end, Mezzofanti uses optical character recognition (OCR) and automatic language translation. Unfortunately, recognizing signs optically is highly computationally intensive, consuming battery heavily. Therefore, a foreign visitor using Mezzofanti frequently is likely to quickly run out of battery power, rendering this electronic translation aid unusable.

OCR is by far the most computationally intensive piece of this application's functionality; it takes about 1 minute to recognize a picture of 200 characters. Because power consumption is roughly proportional to execution time [6], a CPU-intensive task such as OCR executed on a mobile device is likely to drain the device's battery life. An additional source of OCR's energy inefficiency is that it requires that the device's display stay lit during the execution. Since powering a mobile device's screen is known to constitute one of the largest sources of power consumption [7], one can see why OCR is a promising offloading optimization candidate.

We wish to execute the OCR functionality of Mezzofanti at a server remotely and then just display the computed results on the mobile device. Traditionally, the application would have to be partitioned, identifying local and remote parts, which would be communicating with each other across the network. If the network is spotty or simply unavailable, such a partitioning would render the optimized application inoperable.

With our approach, the power consumption of Mezzofanti can be optimized as follows. The programmer would use the `@OffloadCandidate` annotation to mark the program's method that implements the OCR functionality[1]. Our code enhancer analyzes the functionality to be offloaded and inserts two checkpoints: one at the client before the execution is to move to the server, and one at the server before the execution is to return back to the mobile device. The enhancer also generates code to efficiently transfer state between the client and the server. Finally, the enhancer also inserts a fault handler at the client that catches network disconnection events and handles them by continuing the execution locally from the latest checkpoint.

Thus, when Mezzofanti is operated in the presence of a fully covered network, the OCR functionality is offloaded to a server, thereby reducing the application's power consumption. If the network is unavailable or experiences a disconnection, Mezzofanti can continue to deliver its functionality to the user, albeit without optimizing power consumption.

III. Background

Our approach combines program distribution, checkpointing, program analysis, and heap synchronization. We describe these technologies in turn next.

Program distribution: The approaches that can distribute a program to run across the network include automated partitioning, replication, and migration. Automated program partitioning uses a compiler-based tool to introduce distribution to a centralized program [8]. To that end, the tool changes the application's structure (e.g., introducing proxies) and inserts middleware calls. As an alternative to partitioning, a centralized program can be replicated on remote nodes, with the replicas' states synchronized according to a given consistency protocol [9], [10]. The advantage of replication is that the application's structure does not need to change, but synchronizing replicas may cause high network traffic. Finally, migration leverages mobile computing to move execution between remote nodes. Unlike replication, migration moves around a single copy of the executed application's image. Because efficient migration requires runtime system support, a customized runtime environment must be provided for each participating node. Applications, however, can migrate without any changes to their source code [4], [11].

Mobile checkpointing: Checkpointing saves a program's intermediate state, so that the program can be restarted from the latest checkpoint rather than from the beginning, in cases as diverse as recovering from crashes, debugging, or transferring state remotely. As an implementation technique, checkpointing has been used for fault tolerance, debugging, security, dynamic analysis, etc. Checkpointing can be implemented at different levels, including hardware, virtual machine monitor, operating system, application, and language.

In mobile computing, checkpointing is used to transfer state across remote nodes. Checkpointing can help reduce power consumption and save battery life [12], [13]. For checkpointing to be effective toward that end, however, the checkpointed state transferred across the network must be optimized for size, lest the network transfer becomes another bottleneck.

Program analysis: Program analysis infers various facts about the program that can be leveraged for optimization and transformation. Dataflow analysis determines which particular program variable is assigned to which variables [14]. Dataflow analyses operate on a method's control flow graph (CFG) to calculate reachable variables at each statement. Because typical dataflow analysis algorithms are intra-procedural, a whole program must be analyzed to calculate a single variable's flow.

Another program analysis used in this work is side-effect free analysis [15], which determines whether a method changes the program's heap. Side-effect analysis inspects all observable state, including static and heap objects. A method has side-effects if it changes the observable state. For this work, we used Soot [16], a powerful framework for implementing various kinds of program analyses.

Heap synchronization: The heap is the memory space used by the runtime system to allocate memory dynamically. Heaps or their portions can be synchronized across different nodes. One of the difficult issues of heap synchronization is aliasing. When synchronized heap portions, the aliases pointing to the synchronized data items must remain in place. An effective approach to synchronizing liked data structures (e.g., linked lists, trees, and maps) is to use the copy-restore semantics for remote parameters [17]. This semantics copies all reachable data to the server and then overwrites the client copy of the parameter with the server modified data in-place (i.e., while keeping the client-side aliases intact).

IV. Offloading Power Intensive Functionality

Figure 2 describes how our approach can offload power intensive functionality. The programmer is only responsible for annotating those program methods that are known to consume power heavily. The question of how such methods are identified is orthogonal to our approach: power profiling can be used or domain knowledge can be leveraged. The execution offloading analyzer first checks whether the annotated methods can be offloaded and then determines which portion of the program's state would need to be send to the server. Only the methods that do not contain any client-only APIs (e.g., those controlling the GPS, camera, microphone, etc.) can be offloaded. This check simply traverses the program's call graph and checks the reachable statements for the presence of the known libraries that control the mobile

[1]We assume that the application exhibits the level of modularity that at least places distinct functionalities in different program methods.

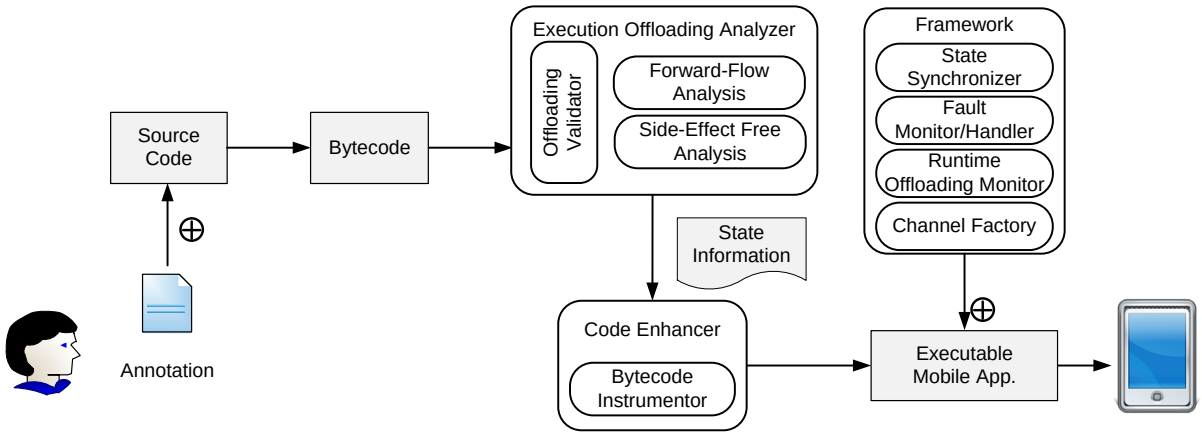

Figure 2. Execution offloading process.

device's hardware components (e.g., `android.hardware.Camera.*` controlling the camera on Android-based devices). This simple heuristic turned out to be quite effective in identifying the methods that cannot be offloaded.

To determine the portion of the offloaded method's state to be transmitted to the server, our approach combines forward dataflow and side-effect free analyses. Specifically, it conservatively determines which portion of the modified program's heap can be accessed by the offloaded method. Without these analyses, the entire heap would have to be transmitted, potentially sending gigabytes of data. Finally, based on the computed transferred state information, the code enhancer inserts the checkpoint (at the bytecode level) that captures the state as well as the state synchronization module that updates the client's heap based on the execution of the offloaded method.

To handle network outages, the inserted offloading functionality is surrounded by a `try-catch` block that catches and handles network-related exceptions. They are handled by restarting the local computation from the latest checkpoint, thus aborting the offloading attempt without disturbing the application.

Our approach to offloading power intensive functionality comprises four parts: a programming model, execution offloading analysis, a code enhancing infrastructure, and a runtime system. We describe these parts in turn next.

A. Programming Model

To demonstrate our programming model, we revisit the Mezzofanti application first introduced in Section II.

Figure 3 partially lists class `OCR` that recognizes the textual representation of a given image. Method `imgOCRAndFilter()` extracts text from its `Image` parameter, storing it in member variable `ssResult`. Having identified this methods as power intensive, the programmer annotates it with `@OffloadCandidate`.

```
1  public class OCR {
2    //member fields
3    ...
4    OCRConfig ocrConf;
5    SpannableString ssResult;
6
7    public void init() { ocrConf = new OCRConfig(); }
8
9    @OffloadCandidate
10   public void imgOCRAndFilter(Image img) {
11     String ocrResult = process(img);
12     ssResult = new SpannableString(ocrResult);
13   }
14
15   public SpannableString getParsedResult() {
16     return ssResult;
17   }
18
19   private String process(Image img) {
20     //process img using ocrConf
21   }
22 }
```

Figure 3. Motivating example revisited.

To run correctly, `ImgOCRAndFilter()` needs to have member field `ocrConf` properly initialized, an operation performed in method `init()`. Notice that `ImgOCRAndFilter()` accesses `ocrConf` indirectly by calling method `process()`.

Because method `imgOCRAndFilter` does not use any client hardware-specific API, it can be offloaded. To execute this method at the server, we need an instance of class `OCR` whose member variable `ocrConf` is initialized. No other member variables are accessed by `imgOCRAndFilter`, so that transferring them to the sever would be wasteful.

When method `imgOCRAndFilter` completes its power intensive execution on the server, member variable `ssResult` will be modified, so that it contains the method's result. Only this member variable needs to transferred back to the client and integrated into the client heap.

```
INPUT:  A set of methods, OM = {m_1, m_2, ..., m_n}
        A call graph, CG
        Constraints, CS_patterns, CS_resource
OUTPUT: A set of methods, M = {m_1, m_2, ..., m_n}

1  while (OM ≠ ∅)
2    m ⟵ OM.next();
3    objects ⟵ getReachableObjects(m);
4    bM ⟵ true;
5    while (objects ≠ ∅) do
6      eachObject ⟵ objects.next();
7      if (eachObject ∈ CS_resource &&
8          eachObject ∈ CS_patterns) then
9        bM ⟵ false;
10     end if
11   end while
12
13   if(bM = true) then
14     M.add(m);
15   end if
16 end while
```

Figure 4. Algorithm to validate offloaded methods.

B. Execution Offloading Analysis

The first analysis step determines whether the annotated methods can be offloaded to the remote server. Figure 4 shows our validation algorithm. As input, the validation algorithm takes a call graph and the client-only API information. All reachable objects in the call graph are transitively check for the absence of client-only APIs (e.g., controlling camera, GPS, microphone, etc.), which can only be executed on the mobile device itself. In addition, the validation algorithm checks for some common patterns that would render offloading impractical. For example, zigzag calling patterns would lead to callbacks to the client, thus negating any energy savings.

The methods to be offloaded are then analyzed for the state that should be transferred between the mobile device and remote server. To select the necessary state, we use a forward dataflow analysis. Figure 5 shows our inter-procedural analysis algorithm that identifies those member variables that have to be passed to an offloaded method and back to the mobile device. The forward dataflow analysis enables to keep tracks of local variables of the offloaded method by calculating the entry and exit of each analyzed statement in the flow graph. The algorithm examines assignment and invocation statements to determine whether local variables are changed. In case of assignments, the following cases are considered:

- If the left value is a class member variable, it is marked as a required variable at the mobile device.
- If the right value is a class member variable, it is marked as a required variable at the remote server.

In case if those member variables are accessed directly, these variables are required for execution offloading. However, if member variables are accessed through local variables by referencing the member variable, the analysis includes all transitional variables. To that end, the algorithm

```
INPUT:  Offloaded method, method
OUTPUT: Read member variables, rV = {v_1, v_2, ..., v_n}
        Written member variables, wV = {v_1, v_2, ..., v_n}

1  Select_State(method) {
2    c ⟵ method.getDeclaringClass();
3    allStmts ⟵ method.getBody();
4
5    while (allStmts ≠ ∅) do
6      stmt ⟵ allStmts.next();
7      if (stmt = assignment statement) then
8        lValue ⟵ stmt.getLeftOp();
9
10       if(lValue ∈ member fields of c) then
11         wV.add(lValue);
12       else if(lValue is a transitional variable) then
13         root ⟵ variableGraph.getRoot(lValue);
14         variableGraph.add(root, lValue); end if
15
16       if(rValue ∈ member fields of c ||
17          rValue ∈ variableGraph) then
18         rV.add(rValue); end if
19
20     else if (stmt = invocation statement) then
21       target ⟵ stmt.getInvocationTarget();
22       m ⟵ target.getMethod();
23
24       if (target ∈ system library) then
25         rV.add(values);
26         if (Side_Effect_Analysis(m)) then
27           values ⟵ stmt.getValue());
28           wV.add(values); end if
29       else if (target ∈ user library) then
30         /* recursive call */
31         values ⟵ Select_State(m);
32       end if
33     end if
34   end while
```

Figure 5. Algorithm for state selection.

maintains a variable graph, so that it can find the member variables from the currently analyzed local variables. In addition, because our forward dataflow analysis is intra-procedural analysis, it is applied to all the methods in the call graph reachable from the offloaded method.

In addition to assignment statements, invocation statements are considered to determine the required state. Invocation statements can be categorized as follows:

- If an invocation is on a member variable, the variable needs to be transferred during the offloading.
- If an invocation on a member variable changes any member variables, the changed variables must be transfered in both directions.

If an invocation is on indirectly accessed member variables, the algorithm determines the root member variable by traversing the variable graph. Because the state selection analysis' scope is limited to application classes, we cannot apply it to system classes. Instead, we employ a side-effect free analysis [15] that determines whether the invocation target changes the heap. If the invocation changes the heap, we mark the invocation's receiver object to be transferred. For example, if an invocation target method is `java.utils.HashMap.put()`, we analyze it for the absence of side-effects. Because method `put()` changes the heap, we

```
1  public class OCR {
2    @OffloadCandidate
3    public void imgOCRAndFilter(Image img) {
4      Checkpoint cp = OffloadingManager.getCheckpoint();
5      cp.addObject("ocrConf", ocrConf);
6      try {
7        cp.setMethod(getClass().getDeclaredMethod(...));
8        OffloadingManager.execute(cp);
9
10       Checkpoint serverCp = OffloadingManager.getUpdatedCP();
11       ssResult = serverCp.updateObj("ssResult", ssResult);
12     } catch (NetworkException e) { //revert to local execution
13       String ocrResult = process(img);
14       ssResult = new SpannableString(ocrResult);
15     } } }
```

Figure 6. Enhanced OCR client class.

```
1  public class OCR {
2    @OffloadCandidate
3    public void imgOCRAndFilter(Image img) {
4      Checkpoint cp = OffloadingManager.getCheckpoint();
5      ocrConf = cp.updateObj("ocrConf", ocrConf);
6
7      String ocrResult = process(img);
8      ssResult = new SpannableString(ocrResult);
9
10     Checkpoint newCp = new CheckPoint();
11     newCp.setMethod(getClass().getDeclaredMethod(...));
12     newCp.addObject("ssResult", ssResult);
13     OffloadingManager.updateCP(newCp);
14   }
15 }
```

Figure 7. Enhanced OCR server class.

determine that the entire `HashMap` member field is to be transferred in both directions.

C. Execution Offloading Code Enhancer

Once the state is selected, the offloaded methods are transformed to run on the server, with the results transferred back to the mobile device. Using the Soot API, our bytecode code enhancer transforms the offloaded methods into cloud and server versions, Figures 6 and 7, respectively.[2] The inserted code interfaces with the runtime to obtain checkpoints as well as to transfer the execution between the mobile device and the server and vice versa.

D. Execution Offloading Runtime System

Our execution offloading runtime system stores checkpoints, handles remote communication, and synchronizes object state. The system keeps track of the latest checkpoint sent to the offloaded method. In the case of a network exception, the execution simply continues locally. The device communicates with the server by means of TCP Java sockets to eliminate the middleware overhead. Object states are synchronized through an efficient implementation of the copy-restore semantics [17]. This semantics efficiently replays remote changes to linked data structures in place (i.e., preserving their aliases).

Because our approach generates the offloading functionality based on the results of static analysis, it is possible that the runtime checkpointed state may turn to be larger than it is practical to transmit across the network. To that end, our runtime system performs a practicality check on the size of the transmitted state. If the state surpasses a parameterizable size threshold in bytes, the runtime system throws a `CheckpointSizeOverflow`. Because this exception class extends `NetworkException`, it is caught by the `catch` clause on line 12 as any other `NetworkException`, causing the execution to continue locally.

[2]Although the functionality is inserted at the bytecode level, we present it in source code for ease of exposition.

V. Evaluation

We have evaluated the effectiveness of our approach in reducing power consumption and improving performance by applying it to five third-party mobile Android applications. The experimental setup has comprised a Motorola Android smartphone, Droid (600 MHz TI OMAP3430, 256 MB RAM) as the mobile device and Lenovo G560 laptop (2.5 GHz Intel i3 CPU, 4 GB RAM) as the server. The mobile device has communicated with the server through a wireless LAN with the average round trip time (RTT) of 73ms, which is representative of modern mobile networks. To measure power consumption, we used PowerTutor [18], a power measuring system for the Android platform.

A. Experimental Subjects

To ensure that our approach is applicable to real-world mobile applications, we chose our experimental subjects from the list of open source projects hosted by Google Code and SourceForge. For each application, Table I lists the name of the offloaded methods[3], the total number of member fields in the method's class, and the number of read/written fields as determined by our analysis algorithm. Without the analysis, all the member fields would have to be transmitted across the network, reducing the efficiency of optimizing power consumption.

Table I
THE BENCHMARK APPLICATIONS AND ANALYSIS RESULTS.

Application	Offloaded method	Member variables	Read variables	Written variables	Ratio
Mezzofanti[4]	OCR.ImgOCRAndFilter()	15	7	5	40%
JJIL[5]	DetectHaarParam.push()	2	1	1	50%
OSMAnd[6]	ShortestPathAlgorithm.execute()	7	1	5	92%
DroidSlator[7]	EnglishDict.translate()	0	0	0	0%
ZXing[8]	DecodedBitStreamParser.decode()	25	15	1	32%

[3]Since the methods are offloaded as specified by the programmer, we chose to offload one most power intensive method per application.

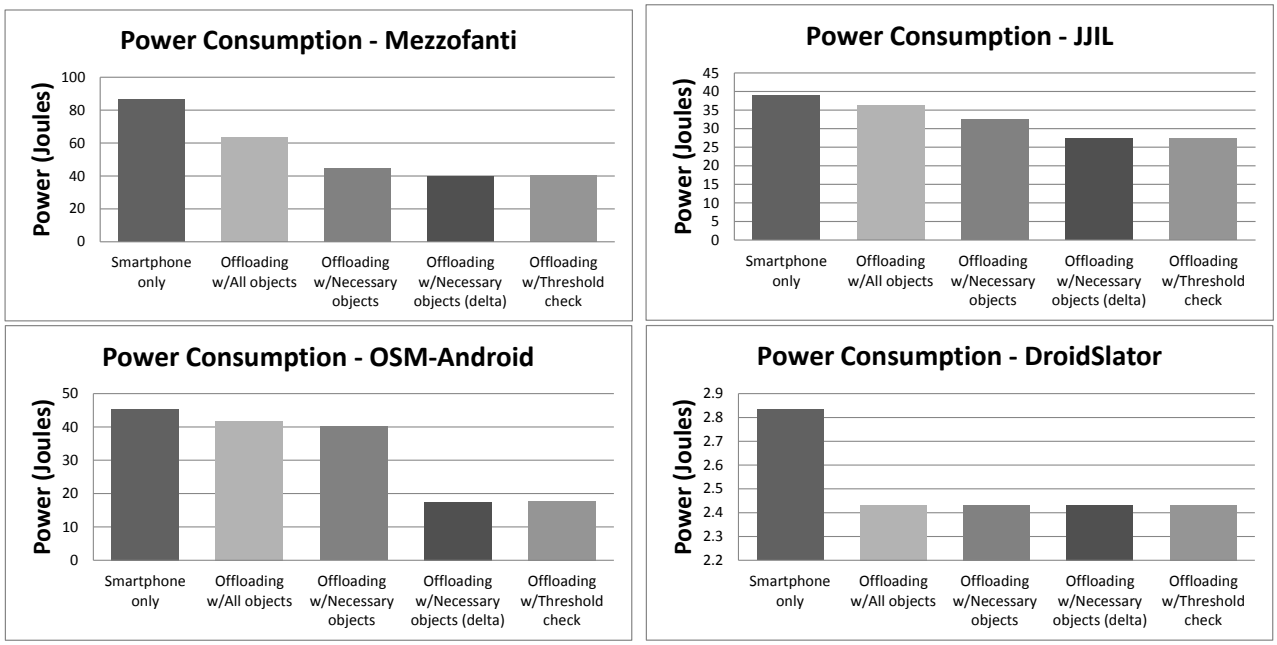

Figure 8. Power consumption.

(1) Mezzofanti—used as our motivating example—offloads its OCR functionality. (2) JJIL detects faces from a picture and offloads its face recognition functionality. (3) OSMAnd navigates the Open Street Map using GPS and offloads its shortest path calculation functionality. (4) DroidSlator translates languages and offloads its translation functionality. (5) ZXing encodes/decodes barcodes and offloads its decoding functionality.

B. Experimental Results

Our approach proved effective in optimizing power consumption and overall execution time for four out of five subject applications. Figure 8 and 9 show how offloading has improved the subjects' power and performance characteristics, respectively. For each subject, the power consumption and performance execution graphs display the results of executing the offloaded method three times in five configurations: (1) original centralized execution, (2) offloaded execution with all the member variables of the offloaded method's class transferred to the server, (3) offloaded execution with only the member variables that are accessed by the server transferred, (4) same as in (3) but also transferring only the delta changes on the second and third run, (5) same as in (4) but with a runtime max data transferred threshold check[9] (see Section IV-D for details).

[4]http://code.google.com/p/mezzofanti/
[5]http://code.google.com/p/jjil/
[6]http://code.google.com/p/osmand/
[7]http://code.google.com/p/droidslator/
[8]http://code.google.com/p/zxing/
[9]We used 6MB for all subjects.

The optimized versions of Mezzofanti, JJIL, DroidSlator, and OsmAnd consumed less energy than their original local versions. Even though Mezzofanti and JJIL send image data to the server, because OCR and face recognition are processing intensive, the optimization reduces power consumption. Although OsmAnd needs to transfer an entire location graph and location points to the server, this transfer takes place only once, and later only a delta needs to be transferred. DroidSlator transfers only a search string to the server, thus saving energy by searching a local server-based database. In terms of performance, the optimized versions of subject applications usually outperform their original local versions: processor-intensive computations can be executed faster by a powerful server than a mobile device. One exception is DroidSlator that transfers little data, but is not processor-intensive. So the performance of the optimized version of DroidSlator is latency-bound.

Figure 10 shows that our approach does not optimize ZXing, which sends a large barcode picture to decode. Sending large items over the network consumes more power than is saved by decoding barcodes at the server. In this case, running locally is the most power efficient option. For cases like that, our implementation performs a runtime check that reverts the execution to the mobile device. We were able to cancel the offloading for ZXing by setting the offloading threshold to 6MB, which stops the offloading before any data is transferred across the network.

To test how the optimized subjects can withstand network disconnection, we ran each application on a mobile device that was disconnected from the network. Each application

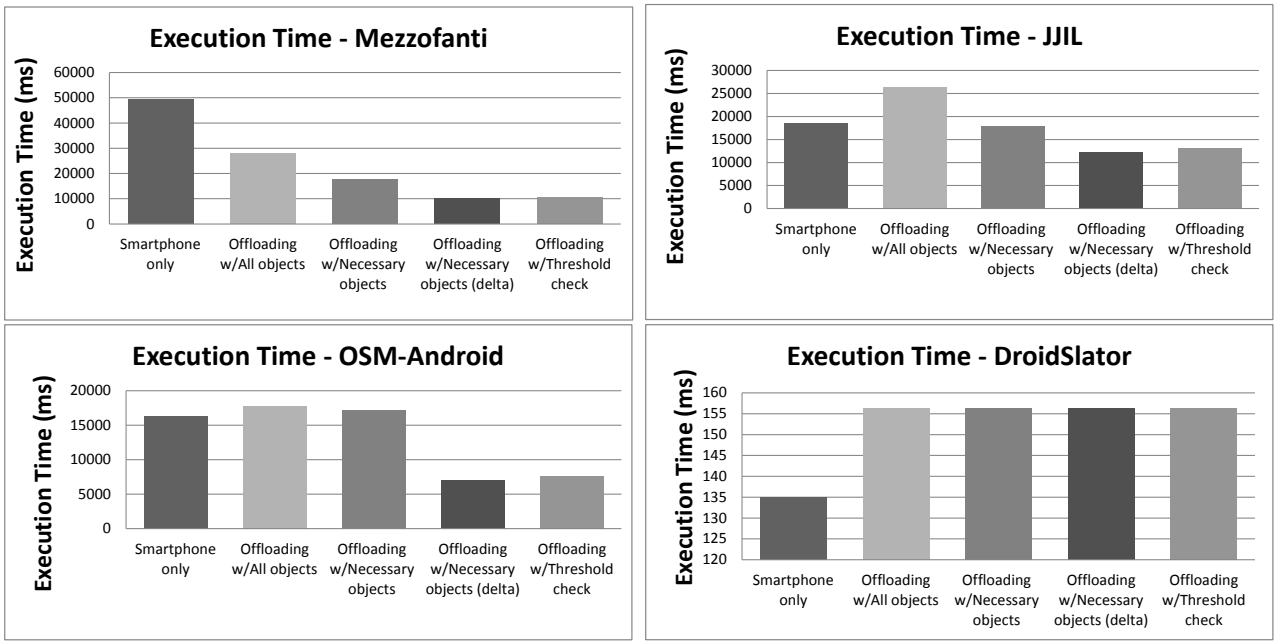

Figure 9. Total execution time.

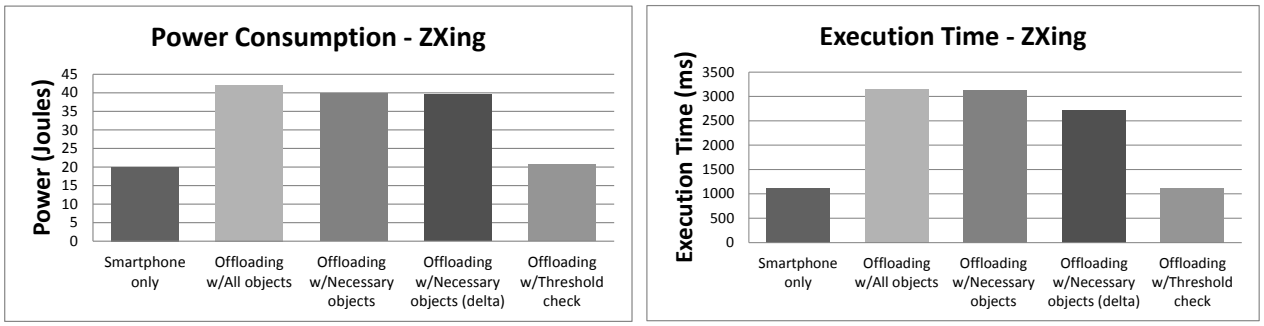

Figure 10. The experimental results of offloading with adaptation.

attempted to offload its power-intensive method to the server, but then was able to switch back to the local execution on the mobile device. We have not measured the effect of this recovery mechanism on power consumption and performance, as network disconnections are expected to be exceptional conditions that happen infrequently.

C. Execution Offloading Index

So far, we compared our experimental results using a single metrics. To obtain deeper insights, we introduce a new metrics, *execution offloading index*, represented by the following equation:

$$EOI = \frac{OET}{ET} \times \alpha + \frac{OPC}{PC} \times (1 - \alpha)$$

where PC and OPC are original and optimized power consumption, respectively; ET and OET are original and optimized execution times, respectively; α denotes a parameterizable weight value. If ET/OET is less than 1, the offloading optimization will increase the application's performance. Similarly, if PC/OPC is less than 1, the offloading optimization will reduce the application's power consumption. The programmer uses the α parameter (ranging between 0 and 1) to express whether the optimization should favor performance or power. To focus on performance, the α parameter should be greater than 0.5; to focus on power consumption, the α parameter should be less than 0.5. When α is exactly 0.5, the focus is on both increasing performance and reducing power consumption.

Figure 11 shows *execution offloading index* of our case study's subject applications. In this analysis, we set α to 0.3, which means that the optimization's focus is on energy saving. The *execution offloading index* value smaller than 1

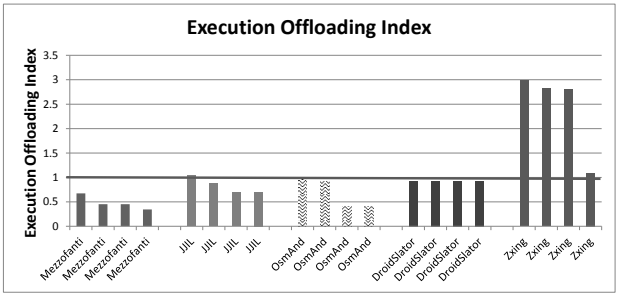

Figure 11. Execution Offloading Index.

indicates that the offloading optimization can indeed reduce power consumption.

The intuition behind the *execution offloading index* is that offloading power-intensive functionality both increases the performance and reduces the power consumption. However, in some special cases performance can be traded for power consumption and vice versa; the programmer can use the α parameter to express such special cases preferences. For example, from the five applications in our case study suite, one exception is DroidSlator, whose *execution offloading index* is greater than 1, but which still reduces its power consumption through offloading. By setting the α parameter to 0.3, the execution offload index becomes less than 1, indicating that DroidSlator can reduce power consumption at the price of slowing down its performance.

D. Limitations

Despite its benefits, our approach is not applicable to all applications. Some mobile applications are written in a monolithic style, in which functionality cross-cuts through traditional modularization program constructs such as classes and methods. Without clear offloading program points, our approach, which operates at the method boundary, would be inapplicable. In addition, mobile applications can allocate and manipulate large objects. Sending large objects across a mobile network can quickly offset any energy savings that can be achieved by offloading intensive processing functionality to the server. These cases limit the applicability of our approach, and the programmer has to be able to identify the cases in which offloading is unlikely to optimize power consumption or increase performance.

Another limitation of our approach is that we do not take multiple concurrent threads into consideration when determining whether a method can benefit from offloading. If an offloaded method can be simultaneously invoked by multiple concurrent threads, our approach currently does not ensure that the program's state remains consistent. However, extending our program analysis heuristics to work with multiple threads is a matter of engineering. Similarly, our runtime can be easily enhanced to become thread-aware. We plan to investigate how our approach can support concurrency as a future work direction.

Finally, our approach only minimally relies on the underlying middleware. As a result, the offloaded methods switch to the local-only execution mode only when the network becomes disconnected. With better middleware support, our runtime system could have recognized those instances when the network's bandwidth/latency characteristics no longer make offloading a worthy optimization and switch to executing locally.

VI. Related Work

Multiple prior research efforts have focused on optimizing power and performance of mobile applications. Introducing distribution to reduce power consumption of a mobile application has been a known optimization strategy [19].

Spectra [20] monitors resource usage and availability to determine whether the mobile application's power consumption can be optimized through execution offloading. To that end, Spectra requires that the programmer manually partition the application to create a proxy for calling remote functions. By contrast, our approach does not introduce remote proxies and modifies the program automatically through bytecode engineering.

Slingshot [21] leverages state replication, so that the replicas can be deployed on remote servers to increase performance. Although Slingshot shares similarity with our approach by relying on synchronizing distributed state, our approach does not require any changes to the runtime system and is portable across any JVMs. Furthermore, while Slingshot optimizes the offloading efficiency by locating the closest surrogate server, our approach relies on program analysis to reduce the size of the program's state that needs to be transferred across the network.

CloneCloud [3] leverages thread-level offloading to optimize mobile execution. Cloudlet [4] migrates the VM. These approaches require a custom runtime system. By contrast, our approach does not change the runtime system but rewrites the program instead to introduce fault-tolreant offloading. As a result, our approach works with standard systems and runtime environments and is easily portable across platforms.

MAUI [2] offloads resource-intensive functionality to remote servers through program partitioning. While MAUI, similarly to our approach, relies on the programmer annotating the source code to which methods to offload, XRay [22] partitions applications automatically. As compared to partitioning-based approaches, our approach relies on checkpointing and program analysis to transfer state across the network efficiently and fault-tolerantly.

VII. Conclusions

We have presented execution offloading, a technique that can reduce power consumption and can also improve performance. Through program analysis and runtime support, our approach reduces the amount of a program's that needs to

be transferred across the network when offloading power-intensive methods. Another advantage of our approach is that our offloading optimization is fault-tolerant—when the network becomes disconnected the optimized application seamlessly switches to the original local-only execution. We have evaluated our approach using five third-party applications, which were optimized to use less energy and run faster, without compromising their fault-tolerance. These results indicate that our approach can help achieve power-efficient and fault-tolerant distributed mobile execution.

ACKNOWLEDGMENTS

This research is supported by the National Science Foundation through the grant CCF-1116565.

REFERENCES

[1] R. Powers, "Batteries for low power electronics," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 687 –693, apr 1995.

[2] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *MobiSys '10: Proceedings of the 8^{th} international conference on Mobile systems, applications, and services*, 2010.

[3] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *EuroSys '11: Proceedings of the 6^{th} conference on Computer systems*, 2011.

[4] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing, IEEE*, vol. 8, no. 4, pp. 14 –23, 2009.

[5] S. Dieckmann and U. Holzle, "A study of the allocation behavior of the SPECJvm98 Java benchmarks," in *ECOOP'99 European Object-Oriented Programming*, 1999.

[6] D. Kirovski and M. Potkonjak, "System-level synthesis of low-power hard real-time systems," in *DAC '97: Proceedings of the 34^{th} annual Design Automation Conference*, 1997.

[7] B. Anand, K. Thirugnanam, J. Sebastian, P. G. Kannan, A. L. Ananda, M. C. Chan, and R. K. Balan, "Adaptive display power management for mobile games," in *MobiSys '11: Proceedings of the 9^{th} international conference on Mobile systems, applications, and services*, 2011.

[8] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java application partitioning," in *ECOOP '02: Proceedings of the 16^{th} European Conference on Object-Oriented Programming*, 2002.

[9] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. . Gifford, and M. F. Kaashoek, "Rover: A toolkit for mobile information access," in *Proceedings of the 15^{th} Symposium on Operating Systems Principles*, December 1995.

[10] M. Kim, L. P. Cox, and B. D. Noble, "Safety, visibility, and performance in a wide-area file system," in *FAST '02: Proceedings of the 1^{st} USENIX conference on File and storage technologies*, 2002.

[11] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI '05: Proceedings of the 2^{nd} conference on Symposium on Networked Systems Design & Implementation - Volume 2*, 2005.

[12] J. Flinn and Z. M. Mao, "Can deterministic replay be an enabling tool for mobile computing?" in *HotMobile '12: Proceedings of 12^{th} Workshop on Mobile Computing Systems and Applications*, 2011.

[13] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of Zap: a system for migrating computing environments," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 361–376, 2002.

[14] M. Sharir and A. Pnueli, *Two approaches to interprocedural data flow analysis*. Prentice-Hall, 1981, ch. 7, pp. 189–234.

[15] A. Rountev, "Precise identification of side-effect-free methods in Java," in *Proceedings of the 20^{th} IEEE International Conference on Software Maintenance*, 2004.

[16] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, ser. CASCON '99, 1999.

[17] E. Tilevich and Y. Smaragdakis, "NRMI: Natural and efficient middleware," in *ICDCS '03: Proceedings of the 23^{rd} International Conference on Distributed Computing Systems*, 2003.

[18] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the 8^{th} IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2010.

[19] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning, "Saving portable computer battery power through remote process execution," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 2, January 1998.

[20] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing performance, energy, and quality in pervasive computing," in *ICDCS '02: Proceedings of the 22^{nd} International Conference on Distributed Computing Systems*, 2002.

[21] Y.-Y. Su and J. Flinn, "Slingshot: Deploying stateful services in wireless hotspots," in *MobiSys '05: Proceedings of the 3^{rd} international conference on Mobile systems, applications, and services*, 2005.

[22] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Enabling automatic offloading of resource intensive smartphone applications," Purdue University, Tech. Rep. ECE-TR-11-3, 2011.