Automated Energy Optimization of HTTP Requests for Mobile Applications

Ding Li, Yingjun Lyu, Jiaping Gui, and William G. J. Halfond
University of Southern California
Los Angeles, California, USA
{dingli, yingjunl, jgui, halfond}@usc.edu

ABSTRACT

Energy is a critical resource for apps that run on mobile devices. Among all operations, making HTTP requests is one of the most energy consuming. Previous studies have shown that bundling smaller HTTP requests into a single larger HTTP request can be an effective way to improve energy efficiency of network communication, but have not defined an automated way to detect when apps can be bundled nor to transform the apps to do this bundling. In this paper we propose an approach to reduce the energy consumption of HTTP requests in Android apps by automatically detecting and then bundling multiple HTTP requests. Our approach first detects HTTP requests that can be bundled using static analysis, then uses a proxy based technique to bundle HTTP requests at runtime. We evaluated our approach on a set of real world marketplace Android apps. In this evaluation, our approach achieved an average energy reduction of 15% for the subject apps and did not impose a significant runtime overhead on the optimized apps.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Energy optimization, Mobile apps, HTTP requests

1. INTRODUCTION

Mobile devices, such as smartphones and tablets, allow users to download and run millions of different apps. These apps combine sensor data and live access to the Internet to provide both entertainment and crucial services to end users. However, the overall usability of mobile devices is limited by their battery life. Therefore, optimizing the energy consumption of these apps can allow users to more fully utilize their mobile devices and take advantage of the services provided by their apps.

ACM ISBN 978-1-4503-3900-1.
DOI: 10.1145/1235

Network communication is one of the primary energy consuming operations in mobile apps. On average, network communications can consume over 40% of the total non-idle state energy of an app [31]. Among all kinds of network operations, those related to HTTP are the most energy consuming, representing almost 80% of all network related energy consumption [31]. Therefore, reducing HTTP related energy consumption can have a significant impact on the overall energy consumption of an app and improve the overall user experience by increasing the underlying device's battery life.

Traditionally, optimizing network communication energy usage has been seen as a hardware or OS level concern. However, over the past couple of years, a growing body of work has begun to investigate ways to optimize network energy consumption at the application layer. One of the more promising ideas has focused on ways to "bundle" HTTP requests – combining multiple small HTTP requests into a larger HTTP request. This has been very successful in the web browser domain for improving performance [14, 21, 43]. In this domain, bundling has targeted multiple parallel asynchronous HTTP requests, such as AJAX calls, to reduce the average network latency of HTTP requests. However, these automated bundling techniques developed for web browsers cannot be used to reduce network energy consumption for mobile apps because they only target asynchronous HTTP requests, which are common in AJAX based web apps, but are far less common in mobile apps. Our recent work has shown that bundling HTTP requests can also decrease energy consumption for mobile apps [29, 30], but did not provide any way to automatically detect when apps should have their requests bundled nor define a way to automate the bundling process.

In this paper, we present a comprehensive approach for detecting when certain types of HTTP requests can be bundled and for rewriting the app so that the bundling can happen at runtime. Our approach can be roughly broken down into three phases: detection, bundling analysis, and runtime optimization. In the first phase, we employ static analysis based techniques to identify HTTP requests that can be safely bundled together. In the second phase, we use string analysis to identify relationships between requests in bundles and configure a set of bundling rules. These rules are then used in the third phase, at runtime, by a proxy server, which detects the start of sequences that can be bundled and uses the bundling rules to carry out the optimization.

We performed an extensive empirical evaluation of our technique and ran it against a set of real-world mobile apps.

In our experiments we found that our technique could reduce, on average, 15% of the apps' total energy consumption. Our approach was also fast, it analyzed and optimized each mobile app in under ninety seconds and did not impose any runtime overhead on the execution of the apps. Lastly, we ran our detection algorithms on 7,878 marketplace apps and found that 4.6% of the apps with more than one HTTP API invocation could potentially be optimized by our technique. This implies that over 40,000 apps in the Google Play app store could benefit from using our technique. Overall these results are very positive and indicate that our approach can significantly decrease energy consumption for a large number of mobile apps.

The structure of this paper is as follows. In Section 2, we introduce background information about HTTP requests in mobile apps and summarize our previous findings. We give an overview of our approach in Section 3. We introduce the details of the three phases of our approach in Section 4, Section 5, and Section 6. The results of our evaluation are reported in Section 7 and we discuss threats to the validity of our evaluation in Section 8. Finally, we have related work in Section 9 and the conclusion in Section 10.

2. BACKGROUND

The process of making an HTTP request consumes a large amount of energy due to the underlying operations such a request entails. HTTP is part of a multi-layer network protocol stack, which includes TCP, IP, and various hardware level protocols. When an HTTP request is sent, it is necessary to encapsulate it in packets of lower level protocols. This process often involves calculating checksums, copying data, and referencing data buffers. Many such operations result in high energy consumption for even a single HTTP request.

HTTP also requires a significant overhead of extra data and messages to be sent when it makes a request. In other words, the size of the HTTP data sent is not the only data transmission cost. The overhead of an HTTP request comes in three forms. First, each packet must contain a set of headers. Although each one of these is small by itself, in total, a typical HTTP packet may contain anywhere from 200B to 2KB worth of headers. Second, the establishment or disconnection of each HTTP connection requires a multi-message handshake protocol at the underlying TCP layer. To establish a connection requires a 3-way TCP handshake and to disconnect requires a 4-way handshake protocol. For both handshake protocols, the packets are typically empty so no useful HTTP data is sent. Third, each HTTP API request also has tail energy, which is independent from the size of the request. Tail energy occurs when the system keeps the network radio in the active state after an HTTP request is finished. This is typically done to attempt to reduce the high energy overhead of starting and shutting down the wireless radio.

Although seemingly small, the overhead of an HTTP request can have a significant impact on its energy efficiency. In prior work [29], we found that downloading one byte of data via HTTP consumed the same amount of energy as downloading 1,024 bytes of data, and downloading 10,000 bytes of data only consumed twice the amount of energy as downloading 1,000 bytes of data. Compounding the problem is that many modern apps only need to send small amounts of information to the server per request. For example, a previous study found that 75% of non-video requests were below 10K bytes [26]. These insights motivate our decision to focus on HTTP optimization and, in particular, our decision to focus on reducing the amount of unnecessary HTTP connections.

3. OVERVIEW OF THE APPROACH

The goal of our approach is to reduce the number of HTTP requests made by a mobile app. To do this we developed an approach to detect and bundle HTTP requests that can be made together. More specifically, our approach first detects *Sequential HTTP Requests Sessions (SHRSs)*, which are sequences of HTTP requests in which generation of the first request implies that the following requests will also be made, and then merges these requests into one request. An example of an SHRS is shown in Program 1. Here, h_1, h_2, and h_3 represent an SHRS since after the generation of the first request, the other two will always be executed. Our approach attempts to detect such situations and rewrite the client side code to combine the requests, where possible, so that there are overall less HTTP requests.

Our approach can be roughly described as having three phases. In the first phase, SHRS detection, our approach uses static analysis to identify all of the SHRSs in an Application Under Test (AUT). Once these are identified, the second phase, Bundling Analysis, analyzes the SHRSs to generate code that, at runtime, will be executed to facilitate the bundling of the HTTP responses of an SHRS. The third phase, optimization, occurs at runtime. In this phase, a proxy intercepts the HTTP requests and runs the bundling code to return all of the corresponding SHRSs' responses. We now explain each of these phases in more detail.

4. SHRS DETECTION

The first phase of our approach is responsible for detecting the SHRSs in an AUT. The input to the phase is the AUT and the output is the set of identified SHRSs. To perform the detection, we first define an intra-procedural static analysis to identify SHRSs within a method, and then use per-method summaries to perform the analysis inter-procedurally.

4.1 Definition of an SHRS

We define an SHRS as a sequence of HTTP or HTTPS API invocations, $S = \langle h_1, h_2 ... h_n \rangle$ that satisfy the following conditions:

1. For any h_i and h_j, if $i < j$, h_i is post dominated by h_j in the app's Control Flow Graph (CFG).

2. For any h_i and h_j where $i < j$, if there is another HTTP API invocation h' on a path from h_i to h_j in the CFG, then $h' \in S$.

The first condition guarantees that the execution of the first HTTP API invocation in S implies that the remaining invocations will be executed sequentially. Referring to the example SHRS of h_1, h_2, and h_3 in method `print_html` of Program 1, this represents the relationship that executing h_1 implies h_2 will also be executed. The second condition ensures that the execution of HTTP APIs in an SHRS will maintain the original server-side state transitions when the requests are bundled. To illustrate, consider the `print_html`

```
1  public void main()
2  {
3     URL url0,url6;
4     //initialize the session
5     url0 = new URL("http://init");
6     urlConnection0 = url0.openConnection();
7     Parse(urlConnection0.getInputStream());//h0
8     print_html(GetCity());
9     //close the session
10    url6 = new URL("http:/close");
11    urlConnection6 = url6.openConnection();
12    Parse(urlConnection6.getInputStream());//h6
13 }
14 public void print_html(String city)
15 {
16    URL url1, url2, url3, url4, url5;
17    URLConnection urlConnection1,urlConnection2,
            urlConnection3;
18    //query current weather
19    url1 = new URL("http://weather?city="+city);
20    urlConnection1 = url1.openConnection();
21    Parse(urlConnection1.getInputStream());//h1
22    //query weather forcast
23    url2 = new URL("http://daily?city="+city);
24    urlConnection2 = url2.openConnection();
25    Parse(urlConnection2.getInputStream());//h2
26    //query location info
27    url3 = new URL("http://location?city="+city);
28    urlConnection3 = url3.openConnection();
29    Parse(urlConnection3.getInputStream());//h3
30    //query the information about the city
31    if(Cond())
32    {
33       url4 = new URL("http://information?city="+city);
34       urlConnection4 = url4.openConnection();
35       Parse(urlConnection4.getInputStream());//h4
36    }
37    //get the rate of the city
38    url5 = new URL("http://rate?city="+city);
39    urlConnection5 = url5.openConnection();
40    Parse(urlConnection5.getInputStream());//h5
41 }
```

Program 1: Example code containing SHRSs.

method in Program 1. Assume that each h_n causes the server side state to be s_n. If line 31 is true, then server-side state transition will be $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5$. If line 31 is false, then the transitions are $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_5$. Without the second condition, an SHRS containing h_1, h_2, h_3, h_5 could be identified. If these requests were bundled and line 31 was true, the server-side would transition $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_5 \rightarrow s_4$. Here the transition $s_5 \rightarrow s_4$ is incorrect.

Note that our definition of an SHRS includes both HTTP and HTTPS request invocations. As we explain in Section 6.2, as long as the proxy is configured with the correct cryptographic key, our approach can properly handle HTTPS as well as HTTP traffic.

4.2 Intra-procedural Analysis

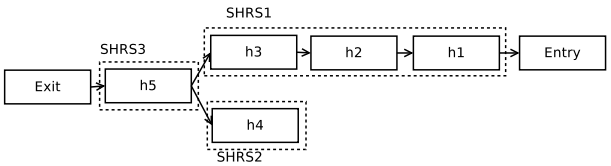

Figure 1: The post dominator tree in Program 1

We defined an intra-procedural analysis to detect SHRSs within a method. The input of the analysis is a method m of the AUT and the output T_m is the set of SHRSs defined within m. Our approach begins by generating the post dominator tree, P, of m. The approach then creates a projection of P, which we call P', that contains only nodes that make HTTP API invocations and edges that represent these invocations' post dominance relationships. Figure 1 shows the projection of the post dominator tree for method `print_html` in Program 1.

The approach then identifies the SHRSs by analyzing P'. To do this, the approach identifies maximal sequences of consecutive nodes in P' in which an edge enters at the beginning of the sequence and exits at the end without the possibility of branching except at the end. These sequences are analogous to basic blocks in control-flow graphs, but are defined over the projection of the post dominator tree. For the purpose of defining these sequences, we consider all calls/invocations to be non-branching and ignore exceptional control-flow. The sequences identified by this analysis are returned as T_m, the SHRSs in the AUT. Figure 1 shows the SHRSs identified for Program 1, $\langle h_1, h_2, h_3 \rangle$, $\langle h_4 \rangle$, and $\langle h_5 \rangle$, in dotted boxes. These are denoted as $SHRS_1$, $SHRS_2$, and $SHRS_3$, respectively. Although $SHRS_2$ and $SHRS_3$ are of size one and cannot be optimized based on intra-procedural information, they may be part of an SHRS defined inter-procedurally, so they are still tracked.

The sequences identified in this analysis satisfy both of the SHRS conditions. First, since all HTTP API invocations are in a sequence in the post dominator tree, they maintain the first condition. Second, since a sequence of nodes (i.e., a basic block) contains no branches except at the last node, it guarantees that all nodes along the sequence are not interrupted by other HTTP requests on the CFG.

4.3 Inter-Procedural Analysis

Our approach also analyzes the AUT to identify SHRSs that are defined inter-procedurally. An example of such an SHRS is $\langle h_5, h_6 \rangle$. To perform the analysis inter-procedurally, we extend our intra-procedural analysis to use and generate per-method summaries. We analyze all methods of the AUT in reverse topological order with respect to the AUT's Call Graph (CG). This ensures that a method m_i is summarized before the approach analyzes another method m_j that calls m_i. Cycles in the CG are handled by merging the individual methods' CFGs and treating them as one method.

The summary for each method m in the AUT represents the sub-tree of P' that represents SHRSs that post dominate the entry of m. The approach uses these SHRSs as the summary because they are guaranteed to be executed when m is invoked. This sub-tree, which we call P'', is identified by analyzing P', which was identified in the intra-procedural analysis. The approach identifies all SHRSs in P' on the path from the entry node to the exit node of m, which, by definition, are the SHRSs that post dominate the entry node. Referring to Figure 1, this is $SHRS_1$ and $SHRS_3$.

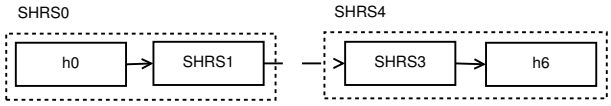

Figure 2: The summary of main in Program 1

To use the summaries, the approach (1) adds a preliminary step to the intra-procedural analysis and (2) modifies the sequence definition. First, before generating P, the approach replaces all invocations to summarized methods with

the target methods' summary. The replacement process for an invocation i first connects the predecessor of i to the entry node of the summary and the exit node of the summary to the successor of i, then removes i from m's CFG. Next, the replacement process reverses all edges within the summary and marks them as having come from the summary. Second, the definition of a sequence is modified as follows. No two nodes may be present in a sequence if they are joined by an edge that came from a summary. For all other purposes the nodes from the summary are treated as instances of an atomic HTTP API invocation. The purpose of these specially marked edges is to prevent SHRSs, such as s_3 and s_1, from being joined together once they are used in the calling method's context. Figure 2 shows the result of the summary substitution on the CFG of the `main` method. Here the summary edge is shown as a dashed line and the two SHRSs that will be identified by the intra-procedural analysis are shown as dotted boxes.

Once the analysis has finished analyzing the root method in the AUT's CG, it takes the union of the T_m for all methods m in the AUT. Then it expands all summary nodes (e.g., $SHRS_3$) to their constituent HTTP API invocations. For the example in Program 1, the reported set of SHRSs is:

$$\{\langle h_0, h_1, h_2, h_3\rangle, \langle h_5, h_6\rangle\}$$

5. BUNDLING ANALYSIS

In the second phase of our approach, the Analyzer examines each SHRS to calculate the needed bundling information and rewrites HTTP API invocations so they will use the response bundles. The inputs to the Analyzer are the AUT and the SHRSs identified in the detection phase. The output is a Bundler for each SHRS in the AUT and AUT$'$, which is the original AUT transformed to carry out the bundlings at runtime. A Bundler is a function that decides which HTTP requests should be bundled for the SHRS and is invoked at runtime by the Proxy. The Bundler has two components, the **Tester** and the **Operator**. The Tester is a set of regular expression patterns that match the URL of the first request in an SHRS. The Operator calculates which HTTP requests should have their responses bundled and is called when the Tester matches a request. The Tester and Operator are separated so that the Proxy does not have to call the more heavyweight Operator every time a new request arrives. The Analyzer first rewrites the HTTP API invocations in the AUT and then analyzes the information provided as arguments to the invocation to generate both components of the Bundler. We explain details of the underlying analyses in the rest of this section.

5.1 String Analysis

An HTTP request is composed of a target URL, a set of headers, and parameters. HTTP is a string based protocol, so if the developer provides these values, then they are provided as strings or via functions that are ultimately mapped to strings. Therefore, to identify the components of the HTTP requests, we use a string analysis tool, Violist [33], to identify the values of the variables used as arguments to the various HTTP API invocations. Violist uses a two-phased analysis. First it generates an Intermediate Representation (IR) of string operations for a given string variables at a point in an AUT, and then it applies custom built interpreters to the IR to generate a model of the possible string values it can have at runtime. The Analyzer needs two different types of string interpretations, which can be easily handled by Violist. We summarize these below, but more details are provided in the Violist paper [33].

The first type of interpreter, which we call the Safe Interpreter (SI), is one that provides a safe approximation of a string variable's possible values. This means that the interpreter generates models that could be described as an over-approximation of the variable's possible values. The SI handles loops by using the `Widen` operation proposed by Yu and colleagues [53] to generate a safe Finite State Automaton (FSA) based model of the possibly infinite string values in a loop. The SI models any substring that cannot be resolved statically, such as user input or files, as any string (i.e., ".*").

The second type of interpreter, which we call the Concrete Interpreter (CI), provides precise approximations that may not necessarily be safe. For example, the CI unravels loops assuming an upper bound n on the loop's iterations. As with the SI, the CI also identifies substrings that cannot be resolved statically, such as user input or files, but represents these using placeholders with unique IDs that correspond to the variables that could not be resolved. These unique IDs facilitate later comparisons of requests at runtime by the Operator (Section 5.4)

5.2 Rewriting HTTP API Invocations

The first step of the analyzer is to rewrite the AUT so that all HTTP API invocations can make use of the bundling invocation. This is done by replacing each HTTP or HTTPS API invocation in the AUT with an Agent HTTP API (AHA), which is a wrapper for the original invocation. The AHA is a static method call that takes the same parameters as its wrapped invocation, along with a Call Site ID (CSID) that uniquely identifies each original invocation, and returns the same type of response object as the wrapped invocation. This design ensures that there is no impact of the optimization on other network related functionality. At runtime, the AHA sends requests to the Proxy instead of the original server and manages the unpacking and distribution of the bundled responses. More details on the runtime behavior of the AUT$'$ are given in Section 6.1.

5.3 Generating the Tester

To generate the regular expressions for the Tester, the approach analyzes the arguments for the first invocation in each SHRS. Due to the definition of the Android network API, string values are not supplied directly to the APIs that make the network requests. Instead, the string values are used to initialize URL, header, and data objects that are then used as arguments for the network request. To address this the Analyzer uses standard alias analysis techniques [42] to identify the allocation sites for the objects provided as arguments to the network requests. Then once the allocation or initialization site for the object has been found, the Analyzer uses the SI to solve for the possible string values. For example, at line 21 of Program 1, *UrlConnection.getInputStream* is the API that makes HTTP requests, but its parameter is the object urlConnection2 rather than the string. The approach uses the alias analysis to figure out that the URL for the HTTP request at line 21 is defined at line 19, and then performs the

string analysis on the variables at line 19. After that our approach uses the generated regular expression as the Tester. A safe approximation is preferred instead of a precise one so that the Tester can automatically generate a pattern that will be guaranteed to recognize the incoming request. Since the models produced by the SI are FSA based, it is straightforward to convert them to regular expressions.

5.4 Generating the Operator

To generate the bundling code of the Operator, the Analyzer identifies the values of the arguments to the remaining invocations in the SHRS. The identified requests will be sent by the Proxy when it receives the first request in an SHRS. This process is analogous to the analysis done for the Tester, but uses the CI to solve for the string values. The Analyzer uses the CI because only SHRSs for which all requests can be solved precisely will be optimizable. An over-approximation is undesirable because it means the Proxy will return additional spurious responses, reducing energy savings. The Operator can solve requests precisely in two cases, which we explain below. For four of the five apps used in the evaluation, these two cases were sufficient for all of the SHRSs.

Case 1 - Constant HTTP Request: An HTTP request is constant when there is only one possible value for the request information. Note that any of these values may itself be defined by expressions (e.g., concatenation) over multiple constant substrings, as Violist is able to evaluate string operations. In general, string values defined via loops or unknown string data (e.g., files) are not constant. If a request is constant, then the CI can precisely identify the string representation of the request and this is saved by the Operator.

Case 2 - Decisive Semi-Constant HTTP Request: An HTTP request is *semi-constant* if values of the request information, such as the URL and parameters, are simple combinations of constants and unknown variables, such as user input. Once the values of all unknown variables are provided, there is no ambiguity in the request that the HTTP API invocation can make. For example, the five HTTP APIs in the method `print_html` of Program 1 are semi-constant since their values are known once the value of the unknown variable *city* is provided. We say a semi-constant HTTP API is *decisive* if its unknown variables are the same as the unknown variables of the beginning invocation in its SHRS. For example, h_2 and h_3 are decisive semi-constant requests because they have the same unknown variable, *city*, as the beginning invocation, h_1, in their SHRS.

In our approach, the Analyzer first detects decisive semi-constant HTTP APIs in an SHRS with the following steps. First, the Analyzer finds all non-beginning HTTP requests in an SHRSs that are semi-constant. This is done by parsing the IR generated by Violist. For an HTTP API, if the IR shows that the variables that represent its URL, fields in headers, and parameters have no branches and loops, the HTTP API is semi-constant. We use this rule to check semi-constant HTTP APIs because the ambiguity of the URL, headers, and parameters of an HTTP API can only be from branches, loops and unknown variables. Second, for each of the semi-constant HTTP APIs in SHRSs, we compare its unknown variables against the unknown variables in the beginning invocation of its SHRSs. If all the unknown variables are also in the beginning invocation, the request is decisive. At runtime, the Operator generates the concrete value of the decisive semi-constant requests by parsing the beginning request of the SHRS. The Operator does this by using regular expressions to retrieve the values for the unknown variables and substituting them into the subsequent requests.

We take the SHRS of h_1, h_2, and h_3 in Program 1 as an example. For the conciseness of the paper, we ignore the headers and parameters of the HTTP APIs since they are all constant in this case. In the first step, our approach checks if h_2 and h_3 are semi-constant HTTP APIs since they are not the beginning HTTP API invocations of any SHRS. The approach analyzes h_2 and h_3 and finds that their URLs contain the unknown variable, *city*, but this part is not defined in branches or loops. Thus, both of them are semi-constant HTTP APIs. In the second step, the approach compares the unknown variable, *city*, in h_2 and h_3 against the unknown variables in h_1, which is the beginning invocation in the SHRS of h_2 and h_3, and finds that they are the same. So, the approach determines that h_2 and h_3 are decisive semi-constant HTTP requests. Finally, the Analyzer generates code for the Operator that uses the regular expression **http://weather?city=(.*)** to retrieve the value at runtime of the variable *city* and puts this value into the request of h_2 and h_3.

All Other Cases: For the requests that are not constant or decisive semi-constant, the Analyzer does not generate bundling code directly. Instead, it records the regular expressions describing the request information along with relevant relationships between the variables. Namely, which ones are the same in different requests. The Analyzer generates these patterns using the same technique described in Section 5.3. The expressions and relationship information is then provided to the developer who may manually specify the form of the requests to be made and define bundles of requests.

Operator Customization: Our approach allows developers to further "tune up" the generated Operator with domain specific knowledge. This tune up is useful to verify the correctness of the generated bundling rules and to achieve specific trade-offs during the bundling. For example, developers may want to avoid bundling an SHRS if they think the bundling could introduce a noticeable latency to a certain critical HTTP request in an SHRS. Our approach exposes the rules of the generated Operator to developers and the tune up can be done by directly adjusting the rules and generated code.

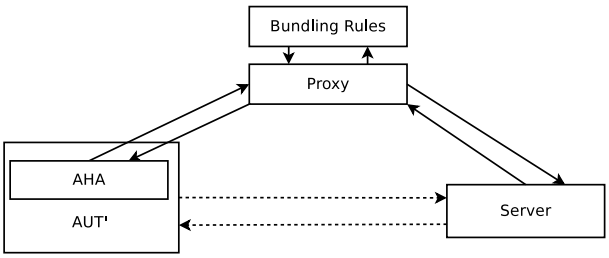

Figure 3: The runtime workflow of our approach.

6. RUNTIME OPTIMIZATION

The third, and final, phase occurs when the AUT′ is executed. At runtime, the AHAs inserted into the AUT′ redirect all requests to a Proxy, which then uses the Bundlers to determine the requests that should be bundled together.

The Proxy sends these requests to the server, bundles the responses, and returns this bundle to the AUT′, where they are unpacked and managed by the AHAs. This runtime workflow is shown in Figure 3. In this figure, the dotted lines show the old workflow and the solid lines show the new workflow.

6.1 The Agent HTTP APIs (AHAs)

The role of the AHA is to carry out the client-side operations of the optimization process while hiding the details of the optimization from the application. When an AHA in the AUT′ is invoked it carries out two steps. First, the AHA checks if the response for the current request is already cached locally. If so, this means that the response was already retrieved by a previous HTTP request in the same SHRS and the AHA directly returns the cached response. If not, the AHA generates a bundle request and sends it to the Proxy. A bundle request is an HTTP request that is identical to the original request but also contains two special header fields to identify the originating CSID and the URL of the original request. Second, when the AHA receives the bundled responses, it unpacks the bundle, returns the response for the originating request, and then caches the responses for the remaining requests in the SHRS.

6.2 The Proxy

The Proxy carries out the server-side part of the optimization process. At runtime the Proxy receives bundle requests, identifies which responses should be bundled together, and returns these to the client-side. To identify the responses that should be bundled together, the Proxy passes the URL and CSID contained in the bundle request's headers to the testers defined by the Proxy's Bundlers. If a tester determines that the request is the first part of an SHRS, then the corresponding operator is used to identify the request information (i.e., URLs and parameters) of the subsequent invocations in the SHRS. These requests are then sent, in sequence, to the app's server. The Proxy then bundles these responses and returns them to the client where they are unpacked and managed by the AHAs as described in the previous section.

The Proxy should be deployed on the same machine or in the same local network as the server. This is not a requirement for correct functionality, but ensures a low latency connection to the original server and avoids any significant slow-down due to network delays. We do not expect that this type of deployment would be difficult since the owners or developers of the AUT generally have control over the deployment of the app's server. Also to properly handle HTTPS, the Proxy also needs to use the same cryptographic signature as the original server.

6.3 Maintaining Server Side States

Our approach has to maintain the server-side states' transitions during HTTP bundling. One challenge is to ensure the order of HTTP API invocations in SHRSs maintain the server-side state transitions. This is addressed by our definition of SHRSs, which requires that requests in an SHRS must be guaranteed to execute if the first one executes (i.e., they post dominate the first request.) Another challenge is to distinguish HTTP API invocations that have the same request information, but from different HTTP API invocations. We need to address the second challenge because two HTTP API invocations may have different responses if they have the same request information but different server side states. Our approach addresses the second challenge by associating a unique CSID to each HTTP API invocation, which will be passed between the client side and the server side. Whenever our approach needs to query the response for a certain HTTP API invocation (i.e., from the cache) in an SHRS, it always uses the CSID along with the request information.

The server may impose global states for all clients. For example, two independent clients may modify a global counter of total active clients at the same time. In this case, bundling HTTP APIs in SHRSs may change the order of requests across different clients and get unexpected server side global states. Our approach does not impose any constraints in the case of global server side states because the order of requests across different clients are generally not guaranteed in most mobile apps. Even if we were to not bundle HTTP requests in an SHRSs, the requests could still return unexpected global states. Furthermore, if the order of requests from different clients is important, developers can still choose not to bundle the requests to avoid potential mistakes.

6.4 Handling Exceptions

Exceptions may interrupt the execution of SHRSs and introduce two challenges to our approach, downloading responses for unreachable HTTP APIs and introducing unexpected server site states. We take an example to explain these two challenges. Suppose there is an exception at line 27 of Program 1. In this case, h_3 will not be executed in the SHRS. However, if we bundle the SHRS of h_1, h_2, and h_3 at the location of h_1, we will have two issues. First, the energy consumed by retrieving the response of h_3 is wasted. Second, the expected server side state should have transition as $s_1 \rightarrow s_2$, but it actually has the transition as $s_1 \rightarrow s_2 \rightarrow s_3$ since h_3 is also requested together with h_1. To address these two challenges, we provide two options for developers.

The first option, the greedy option, is to ignore the potential exception flow and bundle the SHRSs despite the possibility of them being interrupted by exceptions. This option is applicable for cases in which exceptions rarely happen and there is no hard restriction on the server side states. With the greedy option, our approach can still optimize the energy for SHRSs in most cases.

The second option, which is the safe option, avoids bundling SHRSs that may be interrupted by exceptions. This option should be used if the server side states need to be strictly maintained when there is an exception. By using the second option, our approach is safe and avoids any unexpected server side states and energy consumption of retrieving responses for unreached code, yet it may miss some opportunities to bundle larger SHRSs.

7. EVALUATION

We evaluated our approach to determine how well it could perform in practice. In our evaluation, we considered: energy savings, required developer effort for cases where the analysis could not be fully automated, analysis time, runtime overhead, and the prevalence of SHRSs in marketplace apps. Our research questions are listed below:

- RQ 1: How much energy could be saved by using Bouquet?
- RQ 2: How much manual effort is needed to augment the Bundler?
- RQ 3: How long is the analysis time of Bouquet?
- RQ 4: How much runtime overhead is introduced by Bouquet?
- RQ 5: What is prevalence rate of SHRSs in marketplace apps?

7.1 Implementation

Our approach is implemented as a tool, Bouquet, that automatically detects SHRSs, generates code to bundle HTTP requests, and rewrites the AUTs. Bouquet works for Android apps. We chose Android because it is a widely used open source system, but our approach is also applicable for other platforms, such as Windows Phones and iOS, since the mechanism of making HTTP requests is similar on these platforms.

To implement the AUT Instrumenter, we used the apktool [52] to unpack Android apps, dex2jar [1] to convert Dalvik bytecode of Android apps to Java bytecodes, and the BCEL [11] library to replace the HTTP API invocations with the AHAs. To implement the SHRS Detector and the Analyzer, we leveraged the Soot [28] framework to build analysis data structures, such as the control flow graph and the call graph. The Proxy and the Bundler in Bouquet were implemented as a Nodejs server app. We used the express framework [3] to handle the incoming HTTP requests in the Proxy.

In our experiments, we deployed the Proxy and the mock-server on the same machine, which was a DELL XPS 8100 desktop running Linux Mint 14 with an Intel Core i7@3GHz processor and 8GB memory. The machine was connected to a local WIFI router which was linked to the school network of the University of Southern California. The platform on which we ran our subject apps was a Samsung Galaxy S5 smartphone, which was also connected to the same WIFI router of the server.

7.2 Subject Apps

To measure the energy saving of Bouquet, we selected five apps that contained SHRSs. To ensure the representativeness of our subjects, we selected apps that used the two most common APIs for making HTTP requests in Android, which are URLConnection and HTTPClient, and both the GET and POST methods of HTTP requests. These five apps were selected from the 7,878 Google Play marketplace apps that we analyzed for RQ5. The descriptions of our selected apps are in Table 1, where #Bytecode represents the size of an app in terms of the number of Java bytecodes, API represents which APIs are used to make HTTP requests in the app, and Method means which HTTP method is used by the SHRSs in the app.

7.3 RQ 1: Energy Saving

To answer this research question, we used Bouquet to optimize our subject apps. Then we measured and compared the energy consumption of both the optimized and unoptimized versions. To have a full view of the energy saving of our approach, we measured two types of energy savings: the whole app energy saving and the SHRSs' energy saving. For the whole app energy saving, we compared the energy consumption between the unoptimized and the optimized version of each app. For the SHRSs energy saving, we only compared the energy of the optimized and unoptimized HTTP API invocations in each SHRS.

7.3.1 Protocol

We measured the energy saving of Bouquet when running on the five subject apps. Our approach required us to deploy the Proxy in the same network domain as the servers of the apps. However, we did not have access to the server-side since our subjects were not open source apps. To solve this problem, we built a mock-server to mimic the behavior of the servers of our subject apps, and redirected the HTTP requests of our subject apps to the mock-server. The mock-server was a Nodejs based server that accepted the HTTP requests from our subject apps and replied with previously recorded responses that were collected using a capture replay technique, Reran [20]. In our experiment, we executed our apps with the minimal workload that could trigger each of the bundled HTTP requests once. We also captured and replayed this workload with Reran to ensure that both versions of the app were run with the same interactions and timing. This enabled us to avoid any variation in the measurements due to unstable or inconsistent interactions across the versions. We handled the exceptions with the first option described in Section 6.4.

The energy savings of Bouquet depend on the underlying network bandwidth and delay. To measure the energy savings of Bouquet under realistic network conditions, we used a simulator, NEWT [4], developed by Microsoft to simulate the network conditions of WIFI (15M bandwidth, 27ms delay), LTE (6.2M bandwidth, 96ms delay), and 3G (2M bandwidth, 147ms delay) networks. The bandwidth and delay of our LTE and 3G networks were the average values of the four major mobile carriers in the US (ATT, T-Mobile, Sprint, and Verizon) [5, 7]. Unlike LTE and 3G network, which have a uniform quality of network service for all customers, the bandwidth and delay of the WIFI network is highly dependent on the customers' Internet plan. In our evaluation, the bandwidth and delay of WIFI were profiled from the standard plan of Time Warner Cable [10], with a popular network speed tester [8].

The energy consumption was collected with our previous technique, vLens [32], with the Monsoon [6] power meter. We used the vLens technique because it is able to isolate the energy consumption of individual HTTP API invocations from other parts of a subject app's execution and eliminate idle state energy, such as the energy consumed by waiting for user input. Note that eliminating idle state energy was important to have an accurate energy measurement for Android apps. The idle energy was not consumed by the the app itself but by the operating system. Including idle state energy in the total energy measurement could introduce significant inaccuracies in the measurements [31]. To account for random measurement error, each of our subject apps was executed five times, enough to achieve statistical significance, and the average number was taken as the final result. We also conducted a t-test on the null hypothesis: "the energy consumption of an unoptimized version is not larger than the optimized version," which was rejected. We could use the t-test because the random measurement error fol-

Table 1: Description of subject apps

App	Description	#Bytecode	API	Method	Generated	Provided
bobWeather	Weather Forecasting	22,517	URLConnection	GET	88	20
LIRR	Train Schedule Checker	4,408	HttpClient	POST	88	0
Tapjoy	Rewards Tracker	84,963	URLConnection	GET	28	0
ALJA	News Portal	279,114	HttpClient	GET	24	0
PCH	Lottery	216,842	URLConnection	GET	28	0

lowed a normal distribution in independent measurements.

Finally, we manually verified each execution to confirm that the optimized version of the AUT behaved the same as the unoptimized version. To verify this we monitored each workload and verified that the two versions of the app performed the same functions and returned the same answers.

7.3.2 *Result*

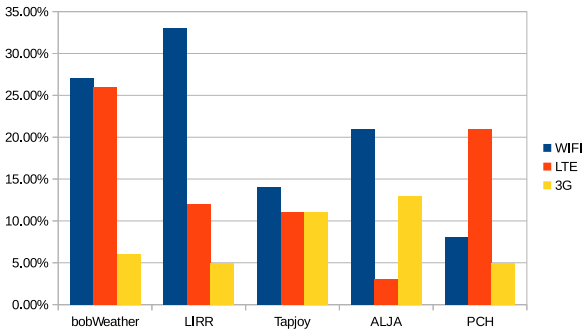

Figure 4: Energy savings at the whole-app level.

Energy savings at the whole app level are shown in Figure 4 and the SHRSs level energy savings are shown in Figure 5. On average, our approach achieved energy savings of 21%, 15%, and 8% at the whole app level for WIFI, LTE, and 3G network, respectively. For the SHRSs level energy savings, the average value for WIFI, LTE, and 3G were 37%, 43%, and 35% respectively. The p-values were 0.049 and 0.0005 for the whole app level energy saving and the SHRS level energy savings, respectively.

7.3.3 *Discussion*

The results showed that our approach was able to achieve significant energy savings across all three types of networks and at both the whole app and the SHRS level. The savings at the whole app level is particularly revealing because it emphasizes not only the energy savings that an end user could expect to see but also that these individual HTTP invocations represent a significant source of energy consumption for a typical app. So much so that the significant local reductions in energy consumption (i.e., for an SHRS) translate into large app level savings as well.

One interesting pattern in our results was that the energy savings for WIFI were higher than those for LTE and 3G at the whole app level. However, the same trend was not observed at the SHRS level. This was interesting because it indicated that, under our configuration of WIFI, the ratio of the energy consumed by HTTP requests to the total energy of the whole app was higher. We hypothesized this was because our WIFI configuration had the fastest network bandwidth, which was not as energy efficient as slower net-

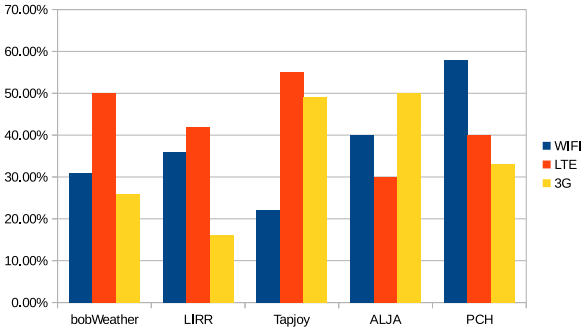

Figure 5: Energy savings at the SHRS level

work configurations for small HTTP requests. In our experiments, the size of our HTTP requests were all between one to three kilobytes. Using a very large bandwidth to retrieve such a small amount of data may not reduce the response time significantly but force the network to consume much more power. Thus, having a very fast network may in fact increase the ratio of HTTP energy to the total energy.

7.4 RQ 2: Manual Effort

In general our approach is fully automated and does not require any developer interaction except to confirm the identified optimizations. In most cases, developers using our approach will only need to check the completeness of the rules generated by the Analyzer. In some cases though, as described in Section 5.4, an optimization cannot be fully automated and the developers need to complete the generated code with the assistance of the generated comments.

As an approximate measure of developer effort, we calculated how many lines and comments were generated by the Analyzer and how many lines needed to be written by the developers. We report these metrics because the Bundlers were the only parts of the process that would require manual intervention by the developer. The number of generated lines and comments reflects the effort needed to learn and understand the Bundlers, and the number of lines needed to be written reflects the required effort to create the Bundlers. To measure how many lines needed to be written, we manually completed the bundling rules with our understanding of the test cases and reported the number of lines we had written for each test case.

The result is shown in the last two column of Table 1. The “Generated” column reports the lines of code and comments that Bouquet generated. The “Provided” column represents the lines of code that had to be created by us to complete the Bundlers. The language for these two columns was JavaScript of Nodejs. Note that the manual effort is to modify the generated code for the proxy, there is no modification required for the apps with our approach.

Table 2: Analysis Time (s)

App	Loading	Analysis	Convert	Rewrite	Total
bobWeather	1.4	4.7	29.4	2.5	38.0
LIRR	1.1	1.2	6.7	1.0	10.0
Tapjoy	2.2	5.3	20.1	7.5	35.1
ALJA	25.1	4.4	43.3	12.4	85.2
PCH	10.4	7.0	36.9	12.2	66.5

On average, Bouquet generated 51 lines of code and comments for each test case. The number of lines of written code was zero for four test cases and 20 for bobWeather. This means that for 4 out of 5 of our test cases, developers only needed to understand the generated code, which was less than 88 lines, to ensure the correctness of the generated Bundlers. For bobWeather, we found that its string values that represented the URLs of each HTTP API invocation in the SHRSs had branches. Since our Analyzer cannot predict which branch would be taken during runtime, it could not automatically generate the code to bundle the SHRSs. However, the code we wrote for bobWeather was straightforward. It consisted of several regular expression operations without any loops or branches. Therefore we believe that the required amount of manual work for developers is reasonable.

7.5 RQ 3: Analysis Time

To answer this research question, we evaluated how much time was consumed by Bouquet to detect SHRSs, generate the optimized version of each subject app, and generate the Bundler. To calculate this time, we measured the execution time of each phase during the measurements for RQ1 and then reported the average execution time.

The result of this measurement is shown in Table 2. The unit for all numbers is seconds. The "Loading" column is the time consumed by the Soot framework to load Android apps. The "Analysis" column is the time consumed by Bouquet to detect SHRSs and generate bundling rules. The "Convert" column is the time used to convert Dalvik bytecodes to Java bytecodes with dex2jar. The "Rewrite" column is the time consumed by Bouquet to replace the original HTTP API invocations in Android SDK with the AHAs. The "Total" column is the sum of all other columns.

On average, the total analysis time for our five subjects was 47 seconds. On average 61% of the total time was spent converting Dalvik bytecode to Java bytecode through dex2jar. Nevertheless, all of our test cases could be analyzed and instrumented in less than 1.5 minutes. These results suggest that the analysis time of our approach would not be a barrier to its acceptance by developers.

7.6 RQ 4: Runtime Overhead

In our approach, runtime overhead may be introduced by the AHAs. Compared to the original HTTP APIs, the AHAs may take longer to process the responses of HTTP requests, since they need to pack and unpack the bundled responses of the HTTP requests. It is important to know how much runtime overhead this may introduce.

To answer this research question, we measured the total response time of all of the HTTP API invocations in the unoptimized app and all of the AHAs in the optimized app. We only measured the total response time of HTTP API invocations or AHAs in SHRSs because AHAs were the only sources of additional runtime overhead.

We conducted the experiment with the same protocol as in Section 7.3. For each subject app, we reported the ratio of the total HTTP response time of SHRSs in the optimized version to the same metric for the unoptimized version. The result is shown in Figure 6.

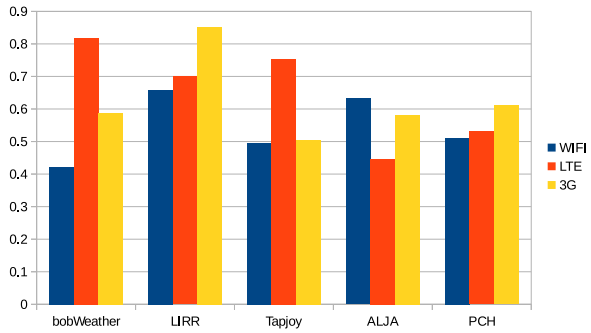

Figure 6: The runtime overhead introduced by Bouquet

On average, the HTTP response time of SHRSs in the optimized versions was 61% of the time in the unoptimized versions. The average standard deviation of our measurement was 13%. The p-values were all below 0.01. This result was counter-intuitive since it showed that the AHAs did not introduce any extra runtime overhead, but, in fact, reduced runtime overhead for the subject apps. We studied this result and found that even though using AHAs required extra time to process the bundled responses, more time was saved due to HTTP request bundling. In our approach, bundling HTTP requests reduced the amount of data that was transmitted through the network so that it also reduced the time consumed to get the HTTP responses. In general, most of the time spent making HTTP requests was consumed by the network transmission instead of processing the responses. Thus, in our experiment, the time saved in network transmission easily dominated the extra processing time for response.

7.7 RQ 5: Pervasiveness of SHRSs

To answer this research question, we carried out an empirical study on a large set of market apps to see how many of them contained SHRSs. We collected 7,878 market apps from the Google Play market, representing 23 different categories, by using the Google Play Crawler [2]. The sizes of our apps varied from several hundreds of bytes to 50 Megabytes. We used the SHRS Detector of Bouquet to parse the 7,878 downloaded apps and reported how many of them had SHRSs. We found that there were 206 Android apps containing SHRSs. This was 2.6% of the entire app pool that we downloaded. Note that for many of the apps it was not possible to have SHRSs because they had zero or just one HTTP API invocation. By excluding the apps which had less than two HTTP invocations, the percentage increased to 4.2%. As reported in a commercial report [9], the number of apps in the Google Play Store was above 1.6 million as of July 2015. 2.6% represents more than 40,000 apps that could contain SHRSs. Note that we did not manually verify each result, so these results only indicate an upper bound on the number of apps that could be optimized. Overall though, these results show that there are potentially many SHRSs in real world apps and thus the impact of our tool could be high.

8. THREATS TO VALIDITY

External Validity: Using only five test cases may introduce threats to the external validity of our evaluation, if the selected apps are not representative. To make the selected apps representative, we selected a set of apps that covered general patterns of the usage of HTTP APIs, which were: both common methods of HTTP requests (POST and GET) and common APIs that can make HTTP requests. Furthermore we also selected apps with different sizes, from 4,408 to 279,114 bytecodes. In general, we believe that our selected apps represent common usages of HTTP APIs in market apps.

The energy savings on the application level in our approach heavily depend on the network condition. To avoid bias introduced by any one particular network, we used a popular network emulator, NEWT [4], to simulate the common network speed and delay for WIFI, LTE, and 3G network. These network configurations were either set to the average value in the US or profiled from the standard network package of one of the common network service providers in the US. Therefore, we believe that the network configuration represents typical network conditions for smartphone users.

Internal Validity: The accuracy of our approach is guaranteed by the definition of SHRSs, which is based on the post domination relationship and "basic blocks" on the post dominator tree. Since the definition of an SHRS is purely based on the static information of the AUT , our approach can accurately detect each SHRS. However, SHRSs are only a subset of HTTP APIs that could be optimized, other HTTP APIs, such as HTTP APIs across different event handlers, may be also optimized with more sophisticated bundling techniques. However, these HTTP APIs are not defined as SHRSs and are not addressed in our approach. We will work on these other HTTP APIs that can be optimized in future work.

The energy measurement in our evaluation also depended on the workload for each app. For example, a longer workload can have a larger energy consumption. To have a fair comparison between the unoptimized and optimized versions of apps, we used Reran [20] to record and replay the identical workload on both the unoptimized and optimized version.

Construct Validity: In the evaluation, we used the lines of code that developers needed to read and write to approximate the manual workload of developers. These metrics did not include the effort to get the required expertise with the AUT. However, as the users of our approach are the developers of the AUT, we assume that they would already have sufficient expertise about the AUT. Thus, we believe the lines of code that developers needed to read and write are reasonable metrics to measure the expected effort of developers.

9. RELATED WORK

Optimizing the performance of HTTP requests is a classic problem in browser and network design. Many approaches have been proposed to send HTTP requests with less TCP connections, such as HTTP pipelining [43], SPDY [14], and multiplexing in HTTP 2.0 [21]. Similar to our technique, these approaches optimize the performance of HTTP requests by bundling multiple HTTP request into one TCP connection. However, these techniques have been proposed to optimize parallel HTTP requests, such as AJAX calls, in web browsers. They cannot be used directly for synchronized sequential HTTP requests, such as SHRSs, in mobile apps because they do not account for the data dependencies in synchronized sequential HTTP requests. Furthermore, unlike these techniques, our approach does not require any adaption of current protocols or system level infrastructure.

To optimize HTTP energy consumption in mobile apps, we made two preliminary studies [29, 30] , which showed that bundling HTTP requests could possibly save HTTP energy. However, those studies did not investigate how to optimize HTTP request energy automatically with static analysis techniques.

Energy optimization for mobile apps is also well studied. One group of energy optimization techniques detect resource leakage issues in mobile apps [45, 38, 12, 39, 23, 19]. Another group of techniques optimize the display energy consumption of mobile apps [18, 34, 47] by using energy efficient color schemes. Researchers have also proposed several techniques to optimize the energy consumption of test cases for in-situ testing [35, 27, 15, 40, 36]. Nikzad and colleagues proposed an annotation language and middleware service [44] that can schedule task on mobile apps in a more energy efficient way. Wang and colleagues proposed a framework to optimize energy for processors and memories [50, 51, 49, 48]. Despite the large body of research work in energy optimization of mobile apps, none of these approaches targets optimizing the energy consumption of HTTP requests.

Besides energy optimization, researchers have also proposed techniques for energy measurement [41, 32, 24, 45, 25, 17, 13] and conducted empirical studies to find potential areas to optimize the energy consumption of mobile apps [31, 22, 35, 37, 46, 16]. This paper is inspired by the results in these studies. However, the measurement approaches and the empirical studies themselves do not propose techniques to optimize energy consumption for mobile apps.

10. CONCLUSION AND FUTURE WORK

Energy is critical for mobile devices. Current approaches for energy optimization do not address a significant source of energy consumption, HTTP requests. In this paper, we propose a technique to optimize energy consumption of HTTP requests by bundling multiple HTTP requests. Our approach uses static analysis to detect requests that can be bundled and then uses a proxy based technique to optimize the sending of the requests. In our evaluation, we found that 4.2% of market apps that have at least two HTTP APIs can be optimized. We also measured the energy saving of our approach on five market apps. In our experiment, our approach averaged a 38% energy reduction for the targeted requests and 15% energy reduction at the application level. These savings show that our approach can help developers to improve the energy efficiency of their apps. In the future, we will develop techniques to optimize more types of HTTP requests.

11. ACKNOWLEDGMENT

This work was supported by NSF grant CCF-1321141 to the University of Southern California.

12. REFERENCES

[1] Dex2jar. http://code.google.com/p/dex2jar/.

[2] Google play crawler http://goo.gl/0yDL5w.

[3] http://expressjs.com/.

[4] https://chocolatey.org/packages/newt.

[5] http://www.fiercewireless.com/special-reports/3g4g_wireless_network_latency_how_do_verizon_att_sprint_and_t-mobile_compar.

[6] http://www.msoon.com/LabEquipment/PowerMonitor.

[7] http://www.pcworld.com/article/253808/3g_and_4g_wireless_speed_showdown_which_networks_are_fastest_.html.

[8] http://www.speedtest.net/.

[9] http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[10] http://www.timewarnercable.com/en/plans-packages/internet/internet-service-plans.html.

[11] Apache. Bcel library. http://bcel.sourceforge.net/.

[12] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *FSE*, 2014.

[13] R. Behrouz, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann. Ecodroid: An approach for energy-based ranking of android apps. In *Green and Sustainable Software (GREENS), 2015 IEEE/ACM 4th International Workshop on*, pages 8–14, May 2015.

[14] M. Belshe and R. Peon. Spdy protocol. 2012.

[15] B. R. Bruce, J. Petke, and M. Harman. Reducing energy consumption using genetic improvement. In *17th Annual Conference on Genetic and Evolutionary Computation. ACM*, 2015.

[16] S. A. Chowdhury, V. Sapra, and A. Hindle. Is http/2 more energy efficient than http/1.1 for mobile users? *PeerJ PrePrints*, 3:e1571, 2015.

[17] M. Dong, Y.-S. K. Choi, and L. Zhong. Power Modeling of Graphical User Interfaces on OLED Displays. In *DAC*, 2009.

[18] M. Dong and L. Zhong. Chameleon: A Color-Adaptive Web Browser for Mobile OLED Displays. *IEEE Transactions on Mobile Computing*, 2012.

[19] A. Ferrari, D. Gallucci, D. Puccinelli, and S. Giordano. Detecting energy leaks in android app with poem. In *Pervasive Computing and Communication Workshops (PerCom Workshops), 2015 IEEE International Conference on*, pages 421–426, March 2015.

[20] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 72–81, May 2013.

[21] I. Grigorik. Making the web faster with http 2.0. *Commun. ACM*, 56(12):42–49, Dec. 2013.

[22] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *ICSE*, 2015.

[23] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang. Characterizing and detecting resource leaks in android applications. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 389–398, Nov 2013.

[24] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating Mobile Application Energy Consumption Using Program Analysis. In *ICSE*, 2013.

[25] A. Hindle. Green mining: A methodology of relating software change to power consumption. In *MSR*, pages 78–87. IEEE Press, 2012.

[26] S. Ihm and V. S. Pai. Towards understanding modern web traffic. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 295–312, New York, NY, USA, 2011. ACM.

[27] E. Kan. Energy efficiency in testing and regression testing – a comparison of dvfs techniques. In *QSIC*.

[28] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[29] D. Li and W. G. Halfond. An Investigation Into Energy-Saving Programming Practices for Android Smartphone App Development. In *GREENS*, 2014.

[30] D. Li and W. G. Halfond. Optimizing Energy of HTTP Requests in Android Applications. In *Proceedings of the Third International Workshop on Software Development Lifecycle for Mobile (DeMobile) – Short Paper*, September 2015. To Appear.

[31] D. Li, S. Hao, J. Gui, and W. Halfond. An Empirical Study of the Energy Consumption of Android Applications. In *ICSME*. IEEE, 2014.

[32] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating Source Line Level Energy Information for Android Applications. In *ISSTA*, 2013.

[33] D. Li, Y. Lyu, M. Wan, and W. G. J. Halfond. String Analysis for Java and Android Applications. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, September 2015. To Appear.

[34] D. Li, H. Tran, Angelica, and G. J. Halfond, William. Making Web Applications More Energy Efficient for OLED Smartphones. In *ICSE*, 2014.

[35] J. Li, A. Badam, R. Chandra, S. Swanson, B. L. Worthington, and Q. Zhang. On the energy overhead of mobile storage systems. In *FAST*, pages 105–118, 2014.

[36] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Optimizing energy consumption of guis in android apps: A multi-objective approach.

[37] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *MSR*, 2014.

[38] Y. Liu, C. Xu, and S. Cheung. Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *PerCom*, 2013.

[39] Y. Liu, C. Xu, S. Cheung, and J. Lu. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *Software Engineering, IEEE Transactions on*, 40(9):911–940, Sept 2014.

[40] I. Manotas, L. Pollock, and J. Clause. Seeds: A software engineer's energy-optimization decision

support framework. In *ICSE*, 2014.

[41] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *ACM Mobicom*. ACM, August 2012.

[42] S. S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[43] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley. Network performance effects of http/1.1, css1, and png. In *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '97, pages 155–166, New York, NY, USA, 1997. ACM.

[44] N. Nikzad, O. Chipara, and W. G. Griswold. Ape: An annotation language and middleware for energy-efficient mobile application development. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 515–526, New York, NY, USA, 2014. ACM.

[45] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys*, 2012.

[46] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause. How does code obfuscation impact energy usage? In *ICSME*, 2014.

[47] M. Wan, Y. Jin, D. Li, and W. G. J. Halfond. Detecting display energy hotspots in android apps. In *ICST*, April 2015.

[48] S. Wang, H. Lee, F. Ebrahimi, P. K. Amiri, K. L. Wang, and P. Gupta. Comparative evaluation of spin-transfer-torque and magnetoelectric random access memory. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2016.

[49] S. Wang, G. Leung, A. Pan, C. O. Chui, and P. Gupta. Evaluation of digital circuit-level variability in inversion-mode and junctionless finfet technologies. *Electron Devices, IEEE Transactions on*, 60(7):2186–2193, 2013.

[50] S. Wang, A. Pan, C. O. Chui, and P. Gupta. Proceed: A pareto optimization-based circuit-level evaluator for emerging devices. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pages 818–824. IEEE, 2014.

[51] S. Wang, A. Pan, C. O. Chui, and P. Gupta. Proceed: A pareto optimization-based circuit-level evaluator for emerging devices. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, (99):1, 2015.

[52] R. Winsniewski. Android–apktool: A tool for reverse engineering android apk files, 2012.

[53] F. Yu, T. Bultan, M. Cova, and O. Ibarra. Symbolic String Verification: An Automata-Based Approach. In *Model Checking Software*. Springer Berlin Heidelberg, 2008.