Efficient and Responsive: Satisfying Delay Constraints in Energy-Efficient Mobile Applications

Nima Nikzad, Octav Chipara†, and William G. Griswold
Department of Computer Science and Engineering
University of California, San Diego, La Jolla, California 92093-0404
Email: {nnikzad, wgg}@cs.ucsd.edu
†Department of Computer Science
University of Iowa, Iowa City, Iowa 52242-1419
Email: octav-chipara@uiowa.edu

***Abstract*—A growing number of smartphone applications are always on and must employ sophisticated power management to avoid excessive battery drain. Previously we introduced APE, an annotation language and runtime for Android that helps developers realize application-specific power-management policies by delaying potentially power-hungry operations until a time when their negative effects are minimized. However, a developer using APE must have a deep understanding of their code's component interactions, such as how delaying an energy-intensive network call might interfere with the timing of user interface calls. Our investigations show that this can indeed happen.**

To address this problem, we introduce a new annotation paradigm that focuses on labeling timing-sensitive computations in lieu of choosing sites where it is safe to introduce delays. Specifically, we introduce a new annotation primitive, @APE_DelayableUpTo, that allows developers to specify delay bounds on computations. Static analysis determines both where delay annotations should be inserted and where additional instrumentation is required to ensure that no delayable execution path exhausts the specified delay allowances at runtime. A wizard guides the user in the formulation of power-saving guards for the delays. Our experiments show that this new approach removes delay problems while still achieving significant power savings.

I. Introduction

In recent years, a growing class of always-on mobile applications have helped turn smartphones into a critical part of our everyday lives. For example, services like Dropbox [**?**] and BitTorrent Sync [**?**] keep important documents synchronized across multiple devices. Personal health and context-monitoring applications, such as Fitbit [**?**], CitiSense [**?**], AudioSense [**?**], BeWell+ [**?**], and Ohmage [**?**], often collect sensor data that is periodically uploaded to a remote server.

However, these continuously-running mobile (CRM) applications often have a disproportionate impact on a device's battery life. While they may be operating at low duty cycles, these applications often rely on power-hungry resources, such as the cellular radio. To ensure their application does not kill a user's battery before the day is through, a developer must ensure their application is as energy-efficient as possible without compromising user experience. Unfortunately, power-management code tends to be complex as it must actively monitor device state and the availability of resources using low-level interfaces.

To address the challenge of building energy-efficient CRM applications, we previously developed and presented Annotated Programming for Energy-efficiency, or APE [**?**]. APE allows a developer to demarcate power-hungry segments of their application with quality-of-service and device state requirements using a simple, declarative annotation language. A runtime service then uses this information to defer the execution of costly operations until the device enters a state that minimizes their energy consumption. For example, a CRM application can reduce the cost of networking operations by deferring their data uploads until another application turns on the radio. If no connection is established within a developer-defined period of time, the application turns on the radio and proceeds with the data uploads. A brief overview of the APE language and runtime is provided in Section II-A.

Although APE dramatically simplifies power-management code while achieving significant power savings, a major unaddressed challenge is that the developer had to deftly place the delay annotations in order to preserve the expected user experience. As a simple example, a developer may not realize that their application has an execution path from a delayed operation to an UI operation. Such a path may cause the UI thread to be blocked while waiting on a delayed operation. Mentally reasoning about such execution paths in a large, object-oriented, multi-threaded application is taxing, at best.

To overcome this problem, we introduce a new paradigm for introducing power-saving delays into applications that separates the concerns of managing power consumption and user experience. Specifically, we introduce a new annotation into the APE language that allows a developer to mark delay-sensitive and delay-intolerant code, and then let the APE compiler and runtime perform the complex tasks of inserting appropriate delays and regulating the delays at runtime to ensure delay-sensitive code is not adversely impacted. The developer still needs to specify the conditions and length of delays, but not where it is safe or effective to insert those delays. This new approach comprises four contributions:

- We present a new power-management annotation paradigm and associated language primitive, `@APE_DelayableUpTo(t)`, that enables developers to specify that an operation cannot be delayed more than

`t` seconds (Section II).

- We present compile-time and runtime support that enforces the delay constraints specified by the developer (Section IV). A static analysis identifies all paths to operations that have been marked by the developer as delay-sensitive. Code is then generated along these paths to keep track of the delays introduced by postponing the execution of power-hungry operation. When the delay allowance of a path is exhausted, power-management delays are suspended along that path until the delay-limited resource is reached and executed.
- We present an analysis and tool that automatically identifies operations in a CRM application that (a) make use of power-hungry resources, (b) generates APE-based power-management policies for them, and (c) provides feedback to the developer regarding available options for customization of the policies (Section V).
- We present a study of six CRM applications that demonstrates that (a) the straightforward instrumentation of CRM applications introduces behavioral problems, (b) the use of our new annotation and the associated analysis solves these problems, and (c) this new paradigm of introducing power management into a CRM application still provides significant power savings (Section VI).

In addition to the above, we discuss related work in Section VII, and conclude in Section VIII.

II. Background and Approach Overview

We previously developed and presented Annotated Programming for Energy-efficiency, or *APE*, to reduce the challenges of building energy-efficient continuously-running mobile (CRM) applications. In this section, we provide a brief overview of APE and its use. (Further details can be found in [**?**]). We then discuss the challenges remaining with this approach, and present our new paradigm for specifying power management policies.

A. APE Language and Runtime

APE's power management model is based on the observation that two (or more) applications, processes, or threads can concurrently use a power-hungry hardware resource at roughly the same energy cost as one of them alone. Thus, significant power savings are possible by delaying, say, a thread's use of the cellular radio until another one starts using it.

APE is designed to ease the development of such power-management policies for CRM applications by providing developers with a high-level annotation language to express quality-of-service requirements and desired device state. These annotations are translated at compile time into Java code that makes calls to the APE runtime service, which coordinates the execution of power-management policies across multiple APE-enabled applications. APE has the benefit of separating policies from the code that implements the functional requirements of the application, while insulating the developer from the low-level complexities of monitoring hardware state.

APE includes a collection of built-in terms that correspond to the states of various hardware components. These terms can be joined together using `AND` and `OR` to compose a boolean expression that describes potential device states. These boolean expressions can then be used along with the `@APE_WaitUntil` annotation to specify operations that should be delayed until the provided expression has been satisfied or some `MaxDelay` number of seconds has passed. Additionally, the `@APE_If`, `@APE_ElseIf`, and `@APE_Else` annotations allow for conditional policy selection at run-time. New terms can be added to the APE language using the `@APE_DefineTerm` annotation and entire policies can be saved and reused via the `@APE_DefinePolicy` annotation.

```
while(true) {
  @APE_If("Battery.Level > 70%")
     @APE_WaitUntil("WiFi.Connected", MaxDelay=1800)
  @APE_Else()
     @APE_WaitUntil("WiFi.Connected", MaxDelay=7200)
  uploadSensorData();
}
```

Fig. 1. An example of a policy implemented using APE: *Delay the upload of sensor data until a Wi-Fi connection is available*

Figure 1 provides an example of how these constructs are combined to compose a power-management policy. The policy expressed in the figure can be interpreted as follows: *If more than 70% of battery life remains, wait up to 1800 seconds (30 minutes) for a Wi-Fi connection before uploading sensor data. Otherwise, wait up to 7200 seconds (2 hours) for a Wi-Fi connection.* These annotations are replaced with corresponding Java code at compile time and used to control the behavior of sensor data uploads at runtime.

Our evaluation of APE found it to be both concise and expressive for a variety of power-management policies. In a case study, APE introduced negligible overhead and equaled hand-tuned code in energy savings, in our study achieving 63.4% energy savings compared to when there is no coordination among threads.

B. A New Approach that Eases Demands on Developer

APE was a significant advance because of its unique approach to power management, as well as abstracting away vast amounts of technical detail in specifying policies and achieving them at runtime. Still, as discussed in the introduction, it was difficult to figure out where best to place an `@APE_WaitUntil` and what to populate its arguments with.

To aid in describing this challenge, we introduce some concepts and terminology. On the whole, the timeliness of operations in mobile applications is an issue of user experience rather than one of correctness. That is, if each thread in an application is guaranteed to eventually make progress, the addition of a bounded delay to any thread does not change this guarantee. However, the user experience may be compromised. We classify operations of mobile applications as *delay-insensitive* or *delay-sensitive* depending on the impact that delaying an operation has on the user experience. Examples of delay-insensitive operations include the download of email or

prefetching of images that may be required at a later point in the application. Delay-sensitive operations typically interact closely with the user interface. For example, introducing a delay on a callback from a UI widget may degrade user experience and even result in "Application Not Responsive" dialogs prompting the user to kill the application. We call such highly sensitive operations *delay-intolerant*.

With these concepts in mind, it is now apparent that the core challenge for the developer is achieving the intended balance between power savings and user experience. Our previous solution encoded both of these concerns into use of `@APE_WaitUntil` annotations. In particular, the developer needed to identify paths of execution that both (a) were time insensitive, and (b) could produce power savings if delayed. This is problematic. For reasons of clarity, the developer would like to place annotations directly on, say, network calls. However, it is common for such calls to be hidden inside helper methods that are widely used in the application. Inevitably, some but not all of its uses would involve, say, retrieving data that would eventually be displayed in the UI. As a result, instead of directly annotating network calls, the developer would need to search up the call graph from the helper method to the callers of that method, and so forth, identifying paths to the network call that are delay-insensitive (or at least not delay-intolerant), and then writing a customized `@APE_WaitUntil` annotation on each path that met the user experience requirements of that path. For delay-intolerant call paths, no annotation would be placed at all. The result, then, is multiple, path-specific `@APE_WaitUntil` annotations that are time-consuming, at best, to place, and to compound matters, are not near the network call that is meant to be delayed.

The contribution of this paper is to decouple the concerns of specifying power management from the concern of preserving user experience. Specifically, we introduce a new APE primitive `@APE_DelayableUpto(t)` and its special case `@APE_Undelayable` that allow the developer to directly annotate operations that should have bounded delays. A compile-time analysis does the call-path reasoning that the developer previously performed. However, instead of inserting customized `@APE_WaitUntil` annotations along these paths, it inserts the power-optimal annotation at the network call site, and then inserts monitoring code along all its call paths (including paths through threads) that tracks the maximum delays allowed on the executed path `@APE_WaitUntil`. The maximum delay of the `@APE_WaitUntil` is then capped at the minimum of its own maximum delay and the delay allowances specified along incoming path.

It is important to note at this point that because this new paradigm allows `@APE_WaitUntil` annotations to be located at the access of resources, it is now possible to automate the siting of these annotations. Identifying the locations where resources are activated or acquired is straightforward as Java and Android use well-known interfaces for such operations. Since writing the specific guard conditions and maximum delay cannot be automated, we employ a wizard to support the creation of policies by presenting the developer with a list of APE terms relevant to the resource in question (Figure 5). The programmer composes the application-specific policy by joining the APE terms with `AND` and `OR` operations. Another benefit of this new approach to specifying power-management policies is that a static program analysis can detect and warn when all the call paths to a `@APE_WaitUntil` have a smaller maximum delay than the `@APE_WaitUntil` itself.

With this overview in mind, the next two sections present our approach in detail.

III. Annotation Semantics

Power annotations either defer operations or constrain how long an operation may be deferred.

A. `APE_WaitUntil`

Consider an execution of an operation O that is annotated with `@APE_WaitUntil`(E, D_O) on a thread τ. Thread τ is blocked until the device enters a state in which expression E holds or the maximum allowable delay D_O is reached. We call the time that a thread τ is blocked while operation O is deferred as the *operation's delay* ($\Delta(O, \tau)$). The `@APE_WaitUntil` annotation imposes the constraint that

$$\Delta(O, \tau) \leq D_O \tag{1}$$

holds on all threads τ that execute O.

B. `APE_DelayableUpto`

The developer may specify quality-of-service properties by constraining an operation's delay. We consider two useful alternative semantics for this annotation, first considering just delay within the operation in question (the `METHOD` case), and then considering all the delays occurring in the operation's thread up to and including the operation (the `THREAD_ENTRY` case).

a) `METHOD` alternative: Consider an operation O annotated with `@APE_DelayableUpTo(`R_O`, METHOD)`. The annotation constrains the aggregated delay of O and all other methods invoked during O's execution to be less than R_O. Formally, we require that for any thread τ:

$$\sum_{o \in children(O) \cup \{O\}} \Delta(o, \tau) \leq R_O \tag{2}$$

where, $children(O)$ is the set of methods invoked by O.

b) `THREAD_ENTRY` alternative: The semantics for the `METHOD` alternative are attractive because they are modular – the timing constraints apply only to the annotated operation. However, in our experiments with annotating real applications, we found that we often wanted to constrain delays on the operation O from the inception of the thread that contains it. That is, in the case where all of the computations in the thread leading up to the execution of O are seen as setting up O, it makes sense to bound their delays as well. Formally, we require that for any thread τ:

$$\sum_{o \in predecessors(O) \cup \{O\}} \Delta(o, \tau) \leq R_O \tag{3}$$

where, $predecessors(O)$ is the set of operations invoked in the thread prior to invoking O.

Because the `THREAD_ENTRY` alternative is "safer" in that it bounds the delays over at least as many operations as the `METHOD` alternative, we treat it as the default in our syntax, should the second parameter be omitted. Additionallly, we define the annotation `@APE_Undelayble()` as equivalent to `@APE_DelayableUpTo(0)`.

C. Example

Consider the example of a simple application that downloads stock data and related images from a server, formats the data, and then displays the results on the UI. Energy consumption can be reduced by batching downloads. In Figure 2, the developer annotates the `openConnection` invocation with an `@APE_WaitUntil` (see line 24). The annotation allows the downloading of images and stock quotes to be delayed until the device is connected to a Wi-Fi network or if cellular activity was observed while connected to either a 3G or 4G cellular network. If no other application turns on the network within 20 minutes, the application turns on the network interface and proceeds with the download. Additional energy savings may be obtained, by also deferring the display of the analysis results for up to 5 minutes when the screen is off (see line 15).

The developer may customize the user experience by introducing `@APE_DelayableUpTo` annotations. For example, perhaps stock quotes should be displayed as soon as possible. The developer may annotate the method invocation `processData` (see line 7) with a `@APE_DelayableUpTo(1 min, METHOD)` to display the analysis results for this timely data more quickly (see line 6, in comment). Note that the annotation is specific to an execution path, applying to the `processData` call on line 7, but not the one on line 9: the call of `processData` in line 9 may delay the display of processed results for the full 5 minutes as specified by `@APE_WaitUntil`.

The developer, however, may still be unsatisfied with how fast images are processed. To further reduce this time, she may change the scope of the annotation by changing the annotation to `@APE_DelayableUpTo(1 min)` (see line 5). The annotation requires that the aggregated delay of all operations from the start of the thread until the completion of the `processData` call on line 7 be less than a minute. Accordingly, both the `openConnection` and `updateDisplay` function may be deferred by a total of a minute; the time each function is delayed depends on the state of the device. As before, the execution path through the other call of `processData` is not constrained, and the `@APE_WaitUntil` may delay processing up the the full 5 minutes.

```
1 class ProcessDataFromServer implements Runnable {
2  public void run() {
3    Data data = NetworkUtils.downloadData();
4    if (data.isStockTicker()) {
5    @APE_DelayableUpTo(1min)
6    // @APE_DelayableUpTo(1min, METHOD)
7      processData(data);
8    } else {
9      processData(data);
10   }
11 }
12
13 void processData(Data data) {
14    ...
15    @APE_WaitUntil(Display.ON, 5min)
16    updateDisplay(data);
17    ..
18 }
19 }
20
21 class NetworkUtils {
22  public static Data downloadData() {
23   URL url = new URL(SERVER_ADDR);
24   @APE_WaitUntil("WiFi.Connected OR Network.Active
25    and (Cell.4G OR Cell.3G)", MaxDelay= 20min)
26   HttpURLConnection conn =
27    (HttpURLConnection) url.openConnection();
28    ...
29   return data;
30  }
31 }
```

Fig. 2. A thread attempts to download data to displayed within the application. The `@APE_DelayableUpTo` annotation ensures that the downloading and displaying of the data is not delayed by APE by more than one minute.

IV. Static Analysis and Run-time Monitoring

An APE annotated application must guarantee that delayed operations satisfy any and all timing constraints specified by a developer. To do so, a combination of static analysis and run-time instrumentation and monitoring is used to ensure that constraints are satisfied during execution. Static analysis translates `@APE_DelayableUpTo` annotations into *delay allowances* that are assigned along relevant call-paths in the program. These allowances bound the delay experienced by a thread of execution: threads *spend* their allowance when waiting at a `@APE_WaitUntil` annotation site and may not spend more than their smallest assigned allowance. When a thread has zero remaining delay allowance, it simply skips any `@APE_WaitUntil` annotations. Threads without any form of `@APE_DelayableUpTo` constraint have infinite delay allowance, meaning that the time spent waiting at a `@APE_WaitUntil` annotation is bounded only by the `MaxDelay` parameter of the policy. Allowances are updated by the APE runtime after each `@APE_WaitUntil` annotation to reflect the time actually spent waiting at the annotation site.

A. Algorithms for Static Analysis and Monitoring

The static program analysis employed by APE is built upon the Soot Java Optimization Framework [**?**]. The first step in our analysis is to generate the control flow graph (CFG) of the target application. Static program analysis using the Soot framework requires that an entry point to the application be specified, and by default this entry point is the `main` method. However, unlike other Java-based programs, Android applications do not include a `main` method. Instead, an application specifies a variety of potential entry points that may

```
1: let G be the control flow graph of the application
2: let X be set of operations annotated with @APE_DelayableUpTo
3: for each operation O ∈ X:
4:   let a be the annotation @APE_DelayableUpTo(scope, R_O)
5:   if (scope == METHOD):
6:     add allowance R_O before O; clear allowance after O
7:   if (scope == THREAD_ENTRY):
8:     let E be the set of entry points in G that reach O
9:     for each e ∈ E: add allowance R_O at e
10:    clear allowance after O
11:  if (scope == THREAD_EXIT):
12:    add allowance R_O before O
13:    let F be the set of exit points in G that may be reached from O
14:    for each f ∈ F: clear allowance R_O at f
```

Fig. 3. Algorithm for instrumenting application code with delay allowances.

```
1: add-allowance(τ, O, R_O):
2:   Δ[τ, O] = R_O

3: remove-allowance(τ, O):
4:   remove (τ, O) from Δ:

5: wait-until-entry(τ, E, D_O):
6:   let R_τ = min_O Δ[τ, O]
7:   wait up to min(R_τ, D_O) for E to be satisfied
8:   let T be the time spent waiting
9:   for each (τ, O) ∈ Δ: Δ[τ, O] = Δ[τ, O] − T
```

Fig. 4. Instrumentation for assigning, clearing, and using delay allowances.

be called by the Android framework, such as the `onCreate` and `onStop` methods that are called when an application is first started or stopped, respectively. To allow Soot to properly analyze the application, a dummy `main` method must be constructed. By default, this main method includes calls to the common Android application lifecycle methods (`onCreate`, `onResume`, etc.). The developer may occasionally have to manually add other entry points in their application to this dummy main method. Further automation of this process using techniques such as those used in FlowDroid [**?**] is an area of future work.

The analysis considers each `@APE_DelayableUpTo` operation in the CFG and generates allowances as follows. Consider the annotation `@APE_DelayableUpTo(`R_O`,` *scope*`)` on an operation O. The locations where the delay allowances are inserted depends on the *scope* of the annotation. As previously mentioned, each `@APE_DelayableUpTo` annotation without the optional *scope* parameter is assumed to have a scope of `THREAD_ENTRY`. If the scope of the annotation is `THREAD_ENTRY`, then delay allowances must be assigned at each entry point with a path to O and cleared immediately after O is completed and returns. This requires computing the set of entry points in the CFG that reach O, which is a reachability problem solved using the Soot generated CFG. If the scope of the annotation is `METHOD`, delay allowances are assigned immediately before O and cleared immedately following O. If the scope of the annotation is `THREAD_EXIT`, delay allowances are assigned immediately before O and is not cleared until the thread of execution reaches its completion, again determined by utilizing the Soot generated CFG. Delay allowances are assigned and cleared via calls to the APE service at runtime, which is discussed in further detail later in this section. The pseudocode of the static analysis is included in Figure 3.

Concurrency and thread synchronization must also be considered during analysis, as a path may be potentially blocked by another path of execution by use of thread synchronization constructs like `wait` and `notify`. If a thread τ is blocked and waiting for notification from another, delayed thread, then τ must also be considered delayed. To properly handle such cases, a points-to analysis is used to identify shared locks across paths. Edges are added to the generated control flow graph from all `notify` calls to matching `wait` calls on shared objects so that any potential delays leading up to the `notify` call are considered also on the path of the `wait` call.

The pseudocode for the instrumentation code is included in Figure 4. The core of our instrumentation code is the shared map (Δ) that includes all the constraints constraints that are currently active. A constraint R_O of operation O on thread τ becomes active when τ executes **add-allowance**(τ, O, R_O) method. The constraint becomes *inactive* after τ executes **remove-allowance**(τ, O). At any point during the execution of τ, the maximum amount of time that a `@APE_WaitUntil` annotation may delay it without violating the delay constraints is:

$$R_t = \min_O \Delta[\tau, O]$$

Accordingly, a `@APE_WaitUntil(`E`,` D_O`)` annotation may wait for expression E to hold for at most $\min(R_\tau, D_O)$ Let $T \leq \min(R_\tau, D_O)$ be the time that τ is blocked. The allowance budget of τ is updated to reflect the introduced delay by subtracting T the budget of all active constraints.

B. Run-time Optimizations

The APE runtime service is primarily responsible for the monitoring of changes in hardware state and the execution of power-management policies on behalf of client applications. To ensure that constraints for delay-sensitive operations are respected at runtime, the APE service has been extended to track running threads and to make use of the code generated during static program analysis. Depending on the thread of execution and path taken through the application, power-management policy requests to the APE runtime fall into one of three categories:

1) **UI Thread Request:** Request was made from the main UI thread of the application,
2) **Constrained Request:** Request was made from a non-UI thread that has a delay constraint, and
3) **Normal Request:** Request was made from a non-UI thread with no constraints.

The 'normal request' is handled as presented earlier: the thread of operation makes a synchronous request to the APE runtime to execute a particular power-management policy. The two other cases, however, require further discussion.

The main UI thread of an application is responsible for handling updates to the user interface of an application. As any long-running operation on this thread would delay updates to the interface and give the appearance of a broken application, such operations should *never* be run on the main thread. In fact, the official Android developer guide explicitly and clearly warns developers: "Do not block the UI thread" [**?**]. As APE-based power-management policies are based around the idea of delaying execution of tasks until an energy-efficient opportunity presents itself, it is clear that the main thread should never be delayed by APE. So as to avoid any unwanted stalls in the UI, whenever a thread reaches a `APE_WaitUntil` annotation, it is checked using `Thread.currentThread().getId()` to see if the executing thread's ID matches that of the main thread. If the calling thread is in fact the main thread, the synchronous request to the APE service is skipped over and execution of the application continues normally. Once a thread has been delayed a total amount of time equal to its allowance, it may no longer be delayed and simply skips all other APE-driven delays as if it was the main UI thread.

V. Policy Generation Engine

The *Policy Generation Engine*, or PGE, is designed to lower the barrier to developing power-management policies with APE by examining the source code of an Android application, identifying instructions that are known to be sources of high-power consumption, and recommending relevant APE power-management policies to apply to the program.

The tool first scans the target application source code looking for any calls to the interfaces provided by Android to power-hungry resources. The PGE does not actively measure the power-consumption of a running application to determine instructions to target. Instead, it makes use of knowledge gathered from official documentation and various best-practice guides to target instruction that are known to wake power-hungry hardware components. The current implementation of the PGE identifies instructions that may wake the smartphone display or cellular radio, as these resources are well suited for delay-based power-management policies and are commonly used in a variety of CRM applications. This list of relevant instructions, and the the hardware resource they utilize, can be found in a file called `rules.pge` that accompanies the tool. This file also includes a list of APE-based power-management policies to recommend for each such instruction and is easily extensible to support adding information about third-party libraries that provide new interfaces for interacting with hardware components.

When the PGE is run, it parses the information found in `rules.pge` and builds a list of all locations in the target application that include a call to an instruction found in the rule set. The tool then presents the developer with information about the identified operations and provides a proposed power-management policy to apply. Specifically, whenever a costly operation is identified, the developer is presented with:

- a snippet of code that provides context to the instruction,
- the APE policy that is being recommended for insertion,
- a plain language description of the recommended policy,
- a list of other APE recognized terms that are relevant to operation being modified, and
- the option to modify, insert, or discard the recommended policy.

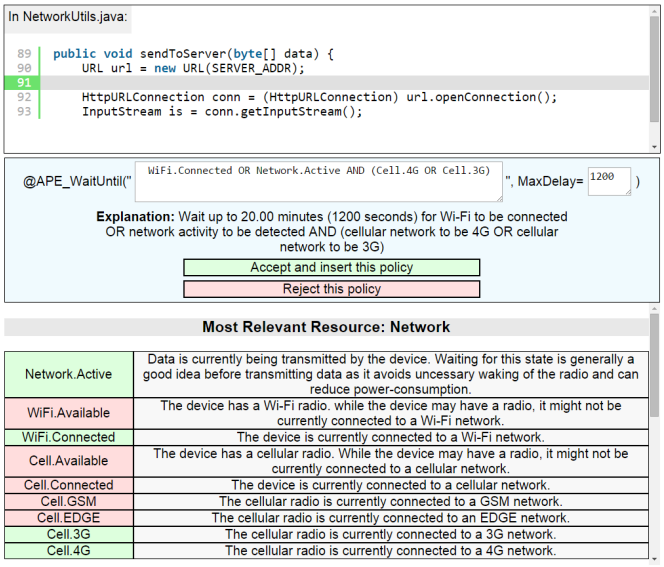

Fig. 5. An example of output from the Policy Generation Engine: a costly network operation has been identified and a general and effective power-management policy is presented and explained.

The description of the policy is intended to assist developers new to APE with understanding how to read and compose their own annotations, while the list of relevant terms is intended to assist the developer with adjusting the generated policy as they see fit.

Figure 5 provides an example of the results provided by the PGE. During analysis of the source code, it was determined that the `openConnection()` method of a `URL` object was being called. As this operation will make use of a network to open an HTTP connection to a desired address, a networking related power-management policy is recommended to the developer. At this point, the developer can insert the recommended policy, discard it, or modify it before inserting it. Clicking on any of the listed terms below the recommendation will automatically insert it into the policies boolean expression.

The authors believe that it is critical that the developer of an application be kept in the loop during policy generation and that no changes be made to the application source code without explicit approval of the developer. Quality-of-service requirements may vary greatly across different applications and domains, and these requirements may not always be inferred by examining source code. For example, the server side component of a particular sensing application may expect updates from clients at least once every twenty minutes. Delaying such an update in hopes of piggybacking on another transmission may improve energy-efficiency in a mobile

application, but it could also cause unintended consequences on the server side.

In certain cases, it is possible that delaying the use of one resource may extend the length of time another resource is powered on. For example, if an application was to force the display to stay awake by acquiring a `WakeLock` and only released that lock after a particular network operation was completed, any delays to that transmission would cause the display to be active for longer than if the network operation was undelayed. However, in the experience of the developers, such interactions are rare in practice, as the logic for managing the state of various hardware components is not typically interwoven in such a manner, nor is it commonly the responsibility of a single thread. Previous work has studied issues related to unreleased locks in mobile applications and provided techniques for identifying them [**?**]. Extending the PGE to handle such cases is an area of future work.

The current implementation of the PGE does not perform a precise points-to analysis and may therefore miss identifying expensive instructions in the face of complicated aliasing. However, in the experience of the authors, this shortcoming has not impacted the tools ability to successfully identify instructions when tested on real-world applications. It is the intention of the developers to eventually pair the PGE with a precise points-to analysis to catch any uncommon issues related to aliasing.

VI. Evaluation

The core claim of our approach is that it is possible to separate the specification of timing constraints and power management policies while still achieving the goals of both. Additionally, we claim that separating the two makes it is possible to automatically identify the sites where power management policies can be inserted. Finally, we claim there is minimal runtime overhead incurred. We evaluate these claims by presenting case studies of our tool's use in real applications drawn from the open-source and research communities.

A. Accuracy of Policy Generation Engine

To evaluate the accuracy of the PGE, we compared it against the `Grep` command-line utility, a tool often used by developers to search large numbers of plain-text files. Since `Grep` can only perform searches, we do not evaluate it against the PGE's ability to guide the formulation of the actual `@APE_WaitUntil` annotation.

For the purposes of the comparison, the authors took on the role of developer for six different Android-based applications and libraries:

- `AndStatus`: a social networking client [**?**],
- `AudioSense`: a CRM application for hearing aid performance evaluation [**?**],
- `K-9`: an e-mail client [**?**],
- `NPR News`: an application for reading and listening to news stories [**?**],
- `ohmage`: a participatory sensing platform [**?**], and
- `WeatherLib`: a library for weather applications [**?**].

Prior to beginning the study, a definitive list of APIs – classes and method names – relevant to power management was derived via careful study of official Android API documentation. Then, for each application, we first ran the PGE to insert `@APE_WaitUntil` annotations. We then repeated the process using `Grep`, recursively invoking it from an application's root directory, looking for whole-word mentions of any of the 18 classes that contain methods that initiate network communication, such as 'HttpClient', 'URL', and 'Socket'. Finally, we exhaustively inspected each application to determine the ground truth. Only Java files were considered, since the PGE and APE are implemented for Java.

For the analysis, we calculated the precision and recall of the PGE and `Grep` at both the file level and the method level, for each application. We considered an insertion recommendation correct at the file level if it at least identified the correct file for insertion of a `@APE_WaitUntil` annotation. Likewise, for the method level, if a recommendation identified the right method for insertion. To make the comparison with `Grep` fair, we did not require line-level precision, as most developers could quickly identify the correct line of code to annotate once in the right method. However, if PGE identified the right method, it identified the right line as well. The results are presented in Table I.

`Grep` found all of the relevant files (i.e., 100% recall) for all of the applications. Recall was also good at the method level, achieving 67% recall or higher on all six applications. `Grep` returned no false positives (i.e., 100% precision) on the Audiology project, at both the file and method level, in part because it encapsulates all networking related code within a single method. Otherwise, method precision was low for `Grep`, often returning many results in files that contained no network operations at all. In many of these files, objects of networking-related classes are instantiated, but not used. For example, the `AvatarData` class in the AndStatus application contains a `URL` object that encodes the path to an avatar image. However, this `URL` is not used from within this class, but rather is accessed by another class that performs the actual communication. In other cases, files contained large comment blocks that discussed how an instance of the class is used in communication elsewhere in the application. Using a tool like the Eclipse IDE's search could avoid such false positives. For the cases in which method recall was below 100%, `Grep` still provided a file-level match. In other words, using `Grep` will eventually get the developer to the relevant operations for power management, but only after wading through many irrelevant results and additional searching.

In contrast, the PGE was found to be fully precise at the file and method level. However, the PGE did miss some results in the ohmage and WeatherLib projects, reducing recall. As discussed in the previous section, the PGE looks for particular method calls on objects of relevant types. In the missed case from the ohmage application, the network-utilizing method call was made directly on the return value of a getter method defined elsewhere in the application. The PGE does not currently infer the return types of method calls and therefore

TABLE I
SEARCHING FOR OPPORTUNITIES TO REDUCE NETWORK-USE RELATED ENERGY-CONSUMPTION

App		Truth		Grep							PGE					
Name	Total Files	Relevant Files	Relevant Methods	Grep Results	Num. of Files	File Precision	File Recall	Num. of Methods	Method Precision	Method Recall	Num. of Files	File Precision	File Recall	Num. of Methods	Method Precision	Method Recall
AndStatus	209	4	8	58	17	24%	100%	14	50%	88%	4	100%	100%	8	100%	100%
Audiology	151	1	1	3	1	100%	100%	1	100%	100%	1	100%	100%	1	100%	100%
K-9	263	6	6	65	12	50%	100%	7	71%	83%	6	100%	100%	6	100%	100%
NPR News	75	4	6	53	11	36%	100%	15	40%	100%	4	100%	100%	6	100%	100%
ohmage	345	3	6	80	14	27%	100%	9	44%	67%	2	100%	67%	5	100%	83%
WeatherLib	70	2	10	32	7	29%	100%	6	50%	30%	1 (2)	100%	50% (100%)	2 (10)	100%	20.0% (100%)

misses this opportunity, though this feature will now be added. The WeatherLib project, on the other hand, makes use of a precompiled external library for most of its networking-related functionality. As information about this library was not initially included in the PGE's `rules.pge` file, it failed to identify it as a source of network utilization. However, when the PGE's rules file was updated to include this library, recall improved to 100%. As an alternative analyzing API's at the source level, the PGE could analyze code at the byte-code level, thus detecting networking calls from compiled libraries without additional information from the developer.

B. Interaction of User Experience with Power Management

To evaluate the effectiveness of the policies recommended by the PGE combined with the use of timing constraints, we took a closer look at the `NPR News` and `AndStatus` applications. We ran the applications in three conditions: unannotated, annotated only with PGE annotations, and annotated with both PGE annotations and timing annotations. The policies were coded to have the effect of delaying all network operations in the two applications by up to five minutes while waiting for either a Wi-Fi connection to become available or for the cellular radio be woken by another process on the device.

All experiments were run on a Pantech Burst smartphone running Android version 4.0. To simulate the presense of other applications running on the device, a service was implemented that would request a network resource once every two minutes. Device power-consumption was measured using a Power Monitor from Monsoon Solutions [**?**]. The battery of the Burst smartphone was modified to achieve a direct bypass between the smartphone and the power monitor, allowing power to be drawn from the monitor rather than the battery itself. All networking was done over the T-Mobile cellular network in the San Diego metropolitan area.

In the PGE-only condition, the generated APE policies successfully captured each networking request made in the application and delayed the operations. In both applications, periodic, battery-draining 'refresh' attempts that polled a remote server for new content were successfully delayed. However, this had undesirable consequences on the user experience in both applications. In the case of `NPR News`, the initial loading and display of news stories was delayed, as was the downloading and playback of user selected audio stories were delayed. In the case of AndStatus, manually-requested refresh attempts by the user were also being delayed. However, these operations eventually completed, preserving the semantics of eventual progress on all threads. Delays like these would likely leave the user staring at a frozen display.

In the PGE-and-timing-annotations condition, both applications were revisited and `@APE_Undelayable` annotations were placed at relevant sites in each application. In the `NPR News` application, an annotation was added within the `AsyncTask` responsible for downloading content on the user's playlist and within the `run` method of a thread responsible for fetching news stories at start up. In the AndStatus application, an `@APE_Undelayable` constraint was added to the `AsyncTask` in the application's service component responsible for executing all requested tasks. When executed, these timing-annotated versions of the applications no longer exhibited the undesirable behavior, while the periodic refreshes that ran in the background continued to be delayed to reduce power-consumption.

We now examine the relative power savings. To simulate real-world usage patterns, each application was used three times during the course of a day (morning, midday, evening) for ten minute periods. The average system power-consumption while running each of the three different versions of the `NPR News` and `AndStatus` applications are presented in Figure 7. In the case of `NPR News`, the application was configured to update news stories automatically once every five minutes. The `AndStatus` application was configured to check for new updates on Twitter once every three minutes. As expected, the applications without any form of APE-driven power-management policy observed the highest power consumption, as operations were executed without concern for the state of the device. Adding in each of the recommended APE policies improved efficiency significantly, reducing power-consumption by 18.1% in `NPR News` and by 22.2% in `AndStatus`. However, as noted above, these versions of the applications included an undesirable user experience. When updated to avoid delaying user-requested updates and initial downloading of content, savings in `NPR News` dropped to 9.5% while `AndStatus` saved 16.6%.

Nearly any application that includes a user-facing compo-

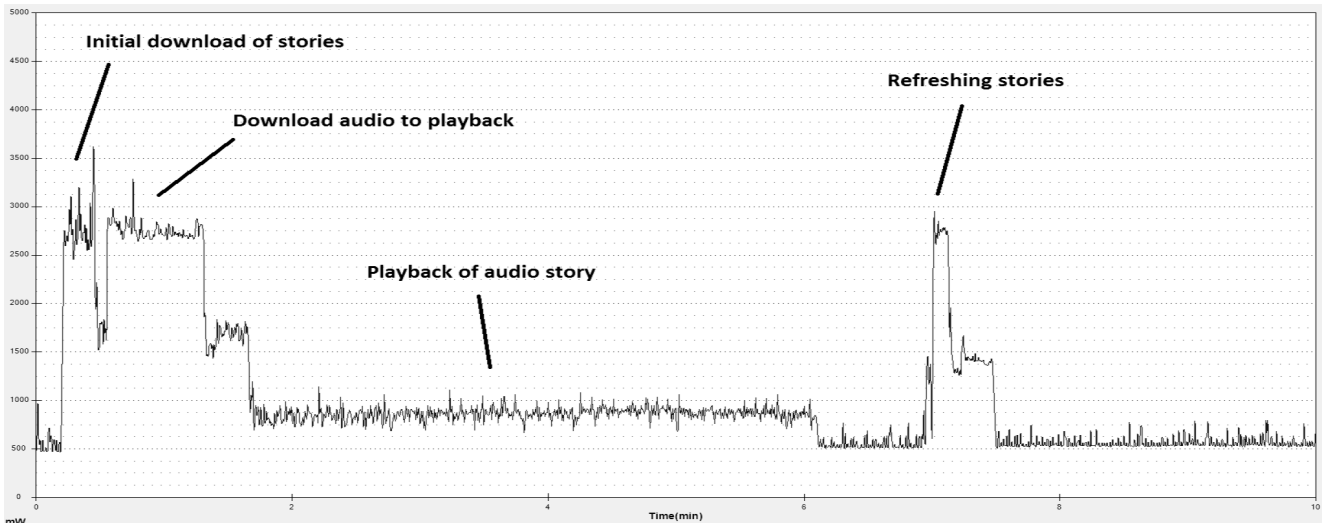

Fig. 6. A trace of power consumption on a smartphone device while using the `NPR News` application to listen to an audio story.

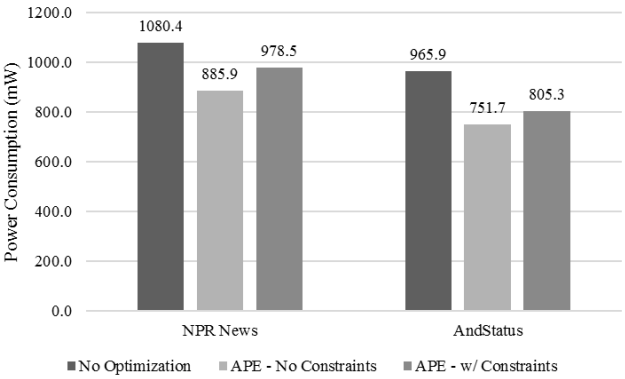

Fig. 7. Power consumption in various versions of the `NPR News` and `AndStatus` applications.

nent is likely to require some form of `@APE_Undelayable` or `@APE_DelayableUpTo` constraint, although the interaction between quality of service and power management could vary widely. In the case of the `NPR News` application, its `@APE_WaitUntil` annotations were reached by a total of 21 paths in the program, and 2 paths were constrained by timing annotations.

C. Runtime Overhead

In our previous APE work we found that each invocation of a `@APE_WaitUntil` entailed 1.71ms of overhead. Most of that cost is due to interprocess communication. Here we report on the additional overhead induced by propagating delay allowances along execution paths. This involves two basic steps: (1) identifying the current thread of execution and (2) inserting and reading allowance values from a hash map based data structure. These steps are typically performed only two to three times during the execution of a path: when an allowance is assigned at the start of the path, when a constrained operation has been completed, and if a `@APE_WaitUntil` annotation is encountered on that path.

The average overhead of these checks were measured to be approximately 2 μs, and 7 μs in the unlikely case of high contention for the synchronized hash map. These overheads are small compared to APE's other overheads because no interprocess communication is involved.

In the general case, the total overhead of allowance tracking along a path of execution is equal to $2\mu\text{s}\times\ C+\ 1710\mu\text{s}\ \times L$, where C is equal to the number of constraints on that path and L is the number of APE policies on that path. Assuming that a path has one constraint and one policy, the addition of allowance monitoring at runtime leads to an expected increase of only 6 μs, or 0.36%. A path with no constraints will access this map only once when a policy is reached, for a total increase in overhead of 0.12% compared to the original APE.

VII. Related Work

Research in energy-efficient software typically falls into one of two categories. Low-level optimizations are typically implemented at the kernel or device-driver level and manage the power state of hardware components. Examples of such techniques include dynamic voltage and frequency scaling (see [**?**] for a review), tickless kernel implementations [**?**], low-power listening [**?**] and scheduled transmissions for radios [**?**], [**?**], and batching of I/O operations for devices such as flash [**?**]. Such optimizations are typically the responsibility of device vendors. System-level optimizations, on the other hand, are implemented at the application- or middleware-level. These optimizations interact with hardware components at longer time-scales and include techniques such as workload shaping, sensor fusion, and filtering. A workload shaping policy like delaying large network operations until a WiFi connection is available is found in applications such as Google Play Market, Facebook, and Dropbox.

The difficulties of implementing such policies motivated our work, as well as that of many others, in developing

higher-level ways of expressing power-management policies. Energy Types allows developers to specify phased behavior and energy-dependent modes of operation in their application using a type system, dynamically adjusting CPU frequency and application fidelity at runtime to save energy [**?**]. EnerJ employs a type system for developers to specify which data values in their application may be approximated to save energy and guarantees the isolation of precise and approximate components [**?**]. While these approaches are useful for building CPU-intensive applications, they have limited applicability to applications that make heavy use of other power-hungry resources. These approaches also require the application to be structured into discrete phases of execution, which may necessitate refactoring when applied to an existing, mature project. In contrast, APE-based policies may be dropped into an existing application without refactoring and are well-suited for managing access to access to hardware components other than CPU, such as the cellular radio and display. Beyond expressing power-management policies, annotation-based approaches have been used for code generation [**?**], [**?**], [**?**], verification [**?**], and driving optimizations [**?**], [**?**].

Procrastinator is a tool that automatically delays the prefetching of network resources in Windows Phone applications so as to reduce costly network data usage [**?**]. Reducing network usage can also reduce power consumption. The Procrastinator Instrumenter identifies prefetching patterns in an application, modifies the relevant network calls at the byte-code level to instead be routed through the Procrastinator Runtime, and delays the fetching of content that is displayed within a UI element until that element is detected to be visible to the user. This work is related to our own in that it eases the implementation of a delay-based technique within an application, but it is narrower in scope and less flexible. Procrastinator focuses only on networking operations, and only those related to the user interface. It also provides no developer control of user experience: all identified prefetching calls are delayed. On the other hand, these limitations allow the approach to be completely automated, with no developer input. In this respect, Procrastinator and APE are at opposing ends of the spectrum of tool-assisted delay-based power management techniques.

VIII. Conclusion

In this paper we presented a new paradigm for introducing power-saving delays into an application. We presented the `@APE_DelayableUpTo` and `@APE_Undelayable` annotations, which allow a developer to demarcate delay-sensitive and -intolerant operations in their application. This information is then used by the APE compiler and runtime to generate and insert effective power-management policies within the target application while ensuring that all delay related constraints are satisfied at runtime. We demonstrated the efficacy of our approach by presenting a study of introducing power-management policies to six CRM applications. Measurements have shown that the generated policies were effective at reducing the power-consumption of a smartphone device, while a small number of constraint annotations can ensure proper application behavior while still providing savings. The addition of delay allowance checking at runtime was shown to have a minimal overhead of only 2μs, a negligible impact on runtime performance.

Acknowledgment

This work was supported by the National Science Foundation (grant nos. CNS-0932403, CNS-1144664, and CNS-1144757) and by the Roy J. Carver Charitable Trust (grant no. 14-4355).

References