APE: An Annotation Language and Middleware for Energy-Efficient Mobile Application Development

Nima Nikzad
Dept of Computer Science and Engineering
Univ. of California, San Diego
La Jolla, CA USA
nnikzad@cs.ucsd.edu

Octav Chipara
Dept of Computer Science
University of Iowa
Iowa City, IA USA
octav-chipara@uiowa.edu

William G. Griswold
Dept of Computer Science and Engineering
Univ. of California, San Diego
La Jolla, CA USA
wgg@cs.ucsd.edu

ABSTRACT

Energy-efficiency is a key concern in continuously-running mobile applications, such as those for health and context monitoring. Unfortunately, developers must implement complex and customized power-management policies for each application. This involves the use of complex primitives and writing error-prone multithreaded code to monitor hardware state. To address this problem, we present APE, an annotation language and middleware service that eases the development of energy-efficient Android applications. APE annotations are used to demarcate a power-hungry code segment whose execution is deferred until the device enters a state that minimizes the cost of that operation. The execution of power-hungry operations is coordinated across applications by the APE middleware. Several examples show the expressive power of our approach. A case study of using APE annotations in a real mobile sensing application shows that annotations can cleanly specify a power management policy and reduce the complexity of its implementation. An empirical evaluation of the middleware shows that APE introduces negligible overhead and equals hand-tuned code in energy savings, in this case achieving 63.4% energy savings compared to the case when there is no coordination.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*;
D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints*

General Terms

Experimentation, Languages, Performance

Keywords

Mobile applications, energy-efficiency, hardware monitoring, annotations, programming

1. INTRODUCTION

The rapidly advancing capabilities of modern smartphones have enabled the development of a new generation of continuously-running mobile (CRM) applications, such as those for personal health and user-context monitoring. Such applications may periodically wake to collect and process sensor data, check with a remote server for updates, or provide reports to a user. Examples include CenceMe [17], SurroundSense [6], AudioSense [12], and CitiSense [21]. Even though these applications operate at low duty cycles, cumulatively they have a large impact on the battery life of a device due to their periodic use of power-hungry system resources, such as the cellular radio for networking or the GPS for localization.

Even though mobile operating systems like Android provide control over these power-hungry resources, developing an energy-efficient CRM application is challenging. Beyond the expected algorithmic and systems challenges of designing a power management policy (see Section 2), there are also significant software engineering challenges:

- *The code for power management tends to be complex.* Not just because the application must actively manage which resources are required or not, but also because it must manage nuanced tradeoffs between the availability of resources and desired battery life. Users are often willing to accept delays in processing or approximations in measurement to increase battery life. Additionally, as will be seen in Section 5.1, power management code is often event-driven and multithreaded.
- *Power management optimizations should be postponed until the application's requirements are set.* Mobile developers often depend on tight, "agile" development cycles in order to elicit feedback from early adopters on basic application behavior. Developing (and cyclically revising) complex power management code early-on would slow this cycle.

Thus, while many power-management techniques have been developed, there is no high-level way to access these as language primitives to specify or implement an application-specific policy for a new application. This paper makes three contributions:

- We present our system, *Annotated Programming for Energy-efficiency* (APE) — a small declarative annotation language with a lightweight middleware runtime for Android (Sections 3.2 and 4). APE enables the developer to demarcate power-hungry code segments (e.g., method calls) using annotations. The execution of these code segments is deferred until the device enters a state that minimizes the cost of that operation. Policies have a declarative flavor, allowing the developer to precisely trade-off delay or adapt sensing algorithms to reduce power. The policies abstract away the details of event and multi-threaded programming required for resource monitoring.

- We introduce an abstract model of the APE approach based on timed automata. With this model, a wide variety of existing power management techniques can be described, demonstrating both the scope and simplicity of the approach (Section 3.1). The model guided both our language and middleware design.

- We provide a first evaluation of the APE approach. We evaluate the expressiveness of the language through (a) a series of policy examples (Section 3.2), and (b) a case study of introducing power management into the CitiSense CRM application, both without and with APE (Section 5.1). For the middleware runtime, we show that an APE-annotated implementation of CitiSense saves as much power as the hand-tuned implementation, while requiring fewer changes to the original source code and having negligible runtime performance overhead (Section 5.2).

2. BACKGROUND AND RELATED WORK

In recent years, energy-saving optimizations for mobile devices have been an active area of research. Such work typically falls into one of two categories: low-level optimizations and system-level optimizations.

Low-level optimizations save energy by judiciously controlling the power state of hardware components such as the CPU, radio, flash memory, and sensors (see [7] for a review). Low-level optimizations are implemented as part of device drivers, or even in hardware, to ensure that they interact with the device at time scales comparable to the transition times between power states (typically sub-millisecond). Examples of such policies include dynamic voltage and frequency scaling (see [28] for a review), tickless kernel implementations [27], low-power listening [24] and scheduled transmissions for radios [8, 31], and batching of I/O operations for devices such as flash [30].

System-level optimizations interact with the hardware components on longer time scales and, as a consequence, may be implemented as part of applications or middleware services. Three common approaches to system-level optimizations include workload shaping, sensor fusion, and filtering. An example of workload shaping is that of delaying large network operations until a Wi-Fi connection is available, as Wi-Fi is typically more efficient than cellular data transmission. Such a policy is found in applications such as Google Play Market, Facebook, and Dropbox. Sensor fusion is used to combine data from multiple potentially heterogeneous sources that have diverse energy costs. Sensor fusion has found applications in state-of-the-art localization techniques that combine information from GPS, cell towers, Wi-Fi, and Bluetooth (e.g., [15]). Power savings are achieved by shifting the sensing burden from the high-power GPS sensor to the other sensors that consume significantly less energy. In the same vein, context-recognition frameworks adapt sensing and networking operations based on the user's context. Optimizations found in such frameworks include adjusting the amount of processing applied to sensor data [16], selecting a minimal set of sensors to power [29], and avoiding sensing altogether by inferring context based on already-known information [20]. Recent techniques for environmental and user-context sensing have examined the use of models to filter out collected data, incurring the high cost of processing and transmitting collected measurements only if they deviate from expected values [19, 22].

Challenges. In spite of the significant progress made in developing power management policies, energy-efficient applications remain difficult to develop. A one-size-fits-all approach fails to yield desirable results as minute differences in hardware specifications or quality of service requirements makes power management policies suboptimal or, in pathological cases, leads to energy being spent ineffectively. This reality leads developers to include highly customized power management code in their applications. Unfortunately, such code is prone to errors [23] because, even when written at the application level, the developer must still use complex, low-level power management primitives and handle the concurrency required by hardware monitoring code.

What is needed is a general mechanism of encoding power management policies that allows a developer to easily customize them for new applications. A general model for specifying power management policies may be difficult to derive as it must encode complex trade-offs between quality of service and energy consumption. We will show that a timed automata model meets these criteria: (1) The model allows simple policies to be composed into more complex ones. (2) More importantly, the complex trade-offs in quality of service can be effectively expressed as state transitions in the timed automaton which are triggered by hardware events and the passage of time.

Our Approach and Related Work. APE employs an annotation language and middleware runtime that allows developers to annotate their code to declaratively compose power-management policies from low-level primitives. APE is not a replacement for existing power-saving techniques, but rather facilitates the design and implementation of such techniques in real-world applications. APE targets the implementation of system-level power optimizations.

The design of APE was inspired by OpenMP [10], an API and library that facilitates the development of parallel C, C++, and Fortran applications. As an alternative to low-level thread management, OpenMP allows a developer to specify—using preprocessor directives placed directly in the code—how tasks should be split up and executed by a pool of threads.

Several other projects have examined the use of annotations for code generation [1, 2, 4], verification [14], and driving optimizations [11, 25]. 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 [9]. However, this approach requires developers to structure their application into discrete phases and would likely require significant effort to apply to an existing application later in the development cycle. EnerJ allows a developer to specify which pieces of data in their application may be approximated to save energy and guarantees the isolation of precise and approximate components

[26]. These systems are complementary to our own and may be used alongside APE. APE is, to our knowledge, the first such system to provide a general and highly-programmable way of expressing runtime power-management policies for mobile applications without the need for significant refactoring.

3. APE DESIGN OVERVIEW

APE is designed to provide developers a simple yet expressive mechanism for specifying power management policies for CRM applications. Three basic principles underline the design of APE:

- APE separates the power management policies (expressed as Java annotations) from the code that implements the functional requirements of an application. This enables developers to focus on correctly implementing the functionality of an application prior to performing any power optimizations.
- APE does not propose new power management policies, but rather it allows developers to compose simple power management policies into more complex ones using an extensible set of Java Annotations. APE annotations are both simple and sufficiently flexible to capture a wide range of power management policies.
- APE insulates the developer from the complexities of monitoring hardware state and provides a middleware service that coordinates the execution of power management policies across multiple applications for increased power savings (compared to when power management is not coordinated across applications).

APE includes an annotation preprocessor and a run-time environment. The preprocessor validates the syntax of annotations and translates them into Java code. The generated code makes calls to the run-time environment that coordinates the execution of power management policies across multiple APE-enabled applications.

The remainder of the section is organized as follows. First, we will introduce the formal model that is used by APE. Then, we present the set of Java Annotations that APE provides to the developer.

3.1 The APE Policy Model

APE builds on the following key insight: *power management policies defer the execution of expensive operations until the device enters a state that minimizes the cost of that operation.* For example, CRM applications 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 user-defined period of time, the application turns on the radio and proceeds with the data uploads. Similarly, an application that maps road conditions (e.g., detect potholes) would collect data only when it detects the user to be driving. An energy-efficient mechanism for detecting driving may be to first use the inexpensive accelerometer to detect movement and then filter out possible false positives by using the power-hungry GPS sensor.

To our surprise, this insight holds across diverse power-management policies that involve different hardware resources and optimization objectives, as illustrated by the examples in this section. Nevertheless, the examples also illustrate the difficulties associated with developing a general model for expressing power-management policies. (1) The model must capture both static properties of hardware resources that may be queried at run-time (e.g., radio on/off) as well as user-defined states that must be inferred using complex algorithms (e.g., driving). Henceforth, we refer to changes in hardware states or in inference results as *application events.* (2) The model must also incorporate a notion of time. The first example illustrates the use of timeouts to trigger a default action. More interestingly, the second example defines a policy where the application should monitor for potholes [18] as a sequences of application events (evolving over time): first the accelerometer must detect movement that is then confirmed by GPS.

APE adopts a restricted form of timed automata to specify power management policies. The automaton encodes the precondition when an operation O should be executed as as to minimize energy consumption. At a high level, a power management policy is encoded by the states and transitions of the timed automaton. The timed automaton starts in the start state and performs transitions in response to application events and the passage of time. Eventually, a timed automaton reaches an accepting state that triggers the execution of O.

Formally, APE's restricted timed automata is a tuple:

$$TA = (\Sigma, S, s_0, S_F, C, E) \tag{1}$$

where,

- Σ is a finite set of events,
- S is a finite set of states, state $s_0 \in S$ is the start state, $S_F \subseteq S$ is a set of accepting states,
- C is a finite set of clocks, and
- E is a transition function.

The transition $e \in E$ is a tuple (c, σ) where c is a clock constraint and σ is a boolean expression consisting of application events. The automaton transitions from state s_i to s_j ($s_i \xrightarrow{c:\sigma} s_j$) when both c and σ hold. In contrast to standard timed automata [5], in our model, transitions from the current state are taken as soon as the required clock constraints and inputs are satisfied. Additionally, we also restrict the expressiveness of clock constraints. Clock constraints can only refer to a single global clock c_G or to a single local clock c_L that is reset each time a transition is taken to a new state. The local clock can be used to impose constraints on transitions outgoing from a state, while the global clock can be used to impose a time constraint on the total delay before an operation O is allowed to execute. The above restrictions ensure that the automaton can be executed efficiently on resource constraint devices such as mobile phones.

To clarify our APE's formal model, let us return to the examples introduced in the beginning of the section.

Example 1: Defer uploads, for up to 30 minutes, until the Wi-Fi radio has connected to a network. Figure 1 shows the automaton associated with this policy. It includes only two states: a start state and an accepting state. Transitions from the start state to the accepting state occur in two cases: (1) when the radio is connected and the global clock is less than 30 minutes or (2) the global clock exceeds 30 minutes. Note the expressive power of the automaton to compactly

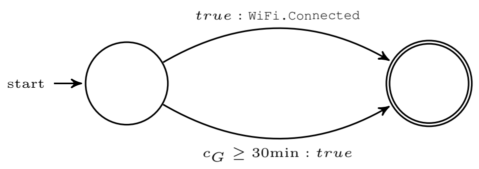

Figure 1: Defer uploads, for up to 30 minutes, until the Wi-Fi radio has connected to a network.

capture conditions that depend both on applications events and time constraints.

Example 2: Defer sensor sampling until the user is driving. Driving is detected by first verifying movement using the accelerometer and then waiting for up to 30 seconds for driving to be confirmed using GPS. Figure 2 shows the automaton associated with this policy. The automaton includes three states, transitioning from the start state to state `Acc` when movement is detected based on readings from the accelerometer, which is captured by predicate `Accel.Move`. The automaton transitions to the accepting state from `Acc` when driving is confirmed based on readings from the GPS, which is captured by the predicate `GPS.Drive`.

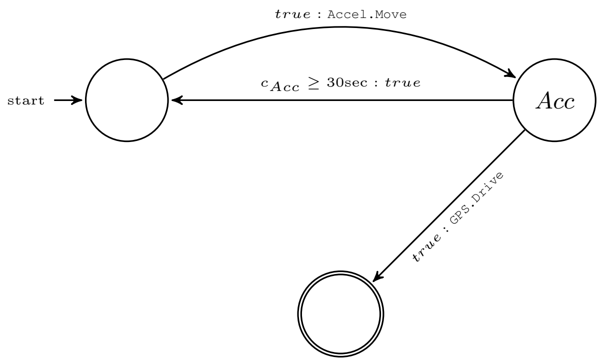

Figure 2: Defer sensor sampling until movement is first detected by the accelerometer and driving is confirmed using the GPS.

The described formal model allows us to capture a wide range of power management policies. However, a disadvantage of using timed automatons as a specification language is that they are hard to define using simple literals that may be included in Java annotations. While experimenting with expressing power management policies in APE, we observed that most of policies have a regular structure that can be captured using a simpler model. The execution of an operation O is deferred until a finite sequence $(\sigma_1, t_1), (\sigma_2, t_2)...(\sigma_n, t_n)$ of states holds:

$$P = (\{(\sigma_1, t_1), (\sigma_2, t_2)...(\sigma_n, t_n)\},\ t_{MaxDelay}) \qquad (2)$$

Sequences of conditions (when $n > 1$) are resolved in order, optionally rolling back and rechecking the previous condition (σ_{i-1}, t_{i-1}) if the current condition being checked, (σ_i, t_i), is not satisfied before t_i time has passed. Additionally, a policy may provide an upper bound, $t_{MaxDelay}$, on the delay introduced by the policy. Constructing a timed automaton from the simple model is a straight-forward process that we omit due to space limitations.

Using this concise notation, the previous two policies can be expressed as:

$$P_1 = (\{(\texttt{WiFi.Connected}, \infty)\}, 30\text{ min})$$

$$P_2 = (\{(\texttt{Accel.Move}, \infty), (\texttt{GPS.Drive}, 30\text{ sec})\}, \infty)$$

Most policies found in literature and real-world applications can also be expressed using APE's simplified model. For example, a policy implemented by applications such as Evernote, Google Play Market, and YouTube, is to delay syncing data and downloading updates until a Wi-Fi connection is available and the device is charging:

$$(\{(\texttt{WiFi.Connected AND Battery.Charging}, \infty)\}, \infty).$$

While advertisements in mobile applications are typically fetched over the network whenever one is required, an advertisement framework could instead display ads from a locally stored corpus that is updated periodically [13]. The policy that manages when updates to the corpus are fetched could be described as:

$$(\{(\texttt{Ads.NeedUpdate}, \infty),$$
$$(\texttt{Net.Active AND WiFi.Connected}, \infty)\}, \infty).$$

Batching write requests to flash memory is yet another example of a power-saving technique for mobile applications. An email client may store newly received emails in memory, writing them out to flash in batches periodically or when the number of emails in memory exceeds some threshold [30]. Such a policy could be described as:

$$(\{(\texttt{Batch.Threshold}, \infty)\}, 60\text{ min}).$$

While these examples show that the simplified timed automata model of Equation 2 is able to express a diverse set of real-world policies, this model is not as expressive as the full model of Equation 1. For example, consider an extension of the driving detection policy in example 2 (See Figure 3). In the extended policy, driving is still detected by first monitoring for movement using the accelerometer and then verifying that the user is driving by using the GPS. However, the policy additionally requires that driving be observed continuously for 30 seconds before allowing execution to continue. During this 30 second period, if the accelerometer fails to detect motion or the GPS to detect driving, the policy immediately returns to the initial state of checking the accelerometer for motion. This policy cannot be expressed in the simplified model because the transition from the state `Acc` to the start state occurs on an event ($\neg$`Accel.Move OR` $\neg$`GPS.Drive`) rather than on a timeout, as required by the simplified model.

APE annotations, further discussed in the next section, build on the simplified model as it has a simple textual representation and captures most of the policies we have encountered. APE may be further extended in the future to provide a more complex syntax for expressing power-management policies using general timed automata.

3.2 The APE Annotation Language

APE realizes the above model in a small and simple language implemented using Java annotations. A preprocessor translates the annotations at compile time into Java code that makes calls to the APE middleware runtime (Section 4).

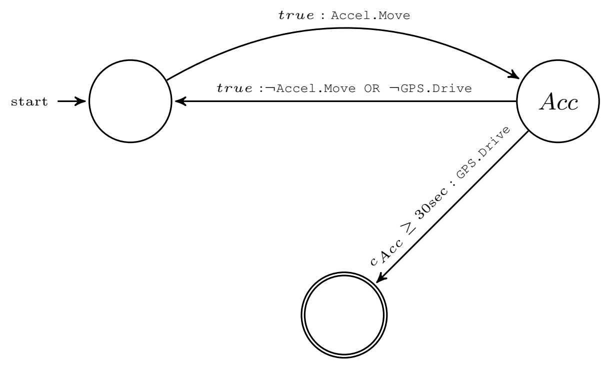

Figure 3: Defer sensor sampling until the user is driving. Driving is detected by first verifying movement using accelerometer for 30 seconds and then confirmed using GPS.

APE_WaitUntil: The *APE_WaitUntil* annotation is the direct realization of the model syntax and semantics specified above in Equation 2. As such, it prefaces a code segment, and specifies the sequence of application events that must be satisfied before execution proceeds to the prefaced code. For example, the policy from example 1 can be expressed as:

```
while(true) {
  @APE_WaitUntil("(WiFi.Connected, inf)", MaxDelay=1800)
  uploadSensorData();
}
```

WiFi.Connected is an APE-recognized application event that APE monitors on behalf of the application; inf says that the local clock constraint is *true*, i.e., $c_s < infinity$; and MaxDelay=1800 asserts global clock constraint as $c_G \geq 1800$ seconds. The parentheses can be dropped when there is no local clock constraint:

```
while(true) {
  @APE_WaitUntil("WiFi.Connected", MaxDelay=1800)
  uploadSensorData();
}
```

Similarly, the policy from example 2 can be expressed as:

```
void startDrivingLogging() {
  @APE_WaitUntil("({accMove()},inf),({gpsDrive()},30)",MaxDelay=inf)
  beginSensorSampling();
}
```

where accMove() and gpsDrive() are local functions that encompass logic specific to the application.

With APE_WaitUntil, it is also possible to designate Java code that must be executed before the thread begins waiting or after waiting has ended:

```
while(true) {
  @APE_WaitUntil("WiFi.Connected", MaxDelay=1800
    PreWait="log('Started waiting for Wi-Fi...')",
    PostWait="log('Finished waiting for Wi-Fi!')")
  uploadSensorData();
}
```

The optional *PreWait* and *PostWait* parameters are useful when an application has to prepare for, or recover from, blocking the annotated thread.

APE_If, APE_ElseIf, and APE_Else: Example 1 can be further extended to *conditionally* wait up to 30 minutes for a Wi-Fi connection if the battery level is greater than 70%, or otherwise to wait up to two hours:

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

The *APE_If*, *APE_ElseIf*, and *APE_Else* annotations allow developers to specify multiple energy-management policies for the same segment of code, selecting one at run-time based on application and device state at time of execution. Unlike APE_WaitUntil expressions, which block until they evaluate true, any expressions provided to an APE_If or APE_ElseIf annotation are evaluated immediately by the APE runtime, and the selected branch is taken to invoke the appropriate policy.

State Expressions and Transitions: The APE_WaitUntil, APE_If, and APE_ElseIf annotations each take as a parameter a boolean expression, consisting of application events, that represents a potential state of the application and device. Each term used in an expression must be either a recognized primitive in the APE language or a valid Java boolean expression. An APE term refers to a property of a hardware resource. For example, WiFi.Connected refers to the status of Wi-Fi connectivity. A Java expression included in an annotation is surrounded by curly braces. Terms are joined using AND and OR operators. As an example:

```
{requestsPending() > 0} AND
     (WiFi.Connected OR Cell.3G)
```

describes the state where the client application has requests pending and it is either connected to a Wi-Fi or 3G network. The requestsPending() method must be in scope at the location of the annotation.

Each of the expressions provided to an APE_WaitUntil annotation is a predicate controlling the transition to the next state in a timed automaton. Consistent with Android's event-driven model, APE treats these predicates as events: When in a given state, APE monitors the events necessary to trigger a transition to the next state. As events arrive, their containing predicate (expression) is reevaluated, and if true, it triggers a transition to the next state. Arriving in a new state causes APE to unregister for the last expression's events, and to register for the events required to trigger the next transition. Likewise, event triggers are set up for a state's local clock; the global clock is its own event trigger, set up when the automaton enters the start state.

The APE compiler must handle two special cases when compiling an APE annotation, both related to the dichotomy between events and method calls. (1) When an APE expression includes a local method call, the method is periodically polled until its containing expression evaluates to true, or until the evaluation becomes irrelevant due to another event trigger. For example, in the case of the expression {requestsPending() > 0}, the method requestsPending() is periodically polled until its containing boolean expression evaluates to true. (2) When APE initially registers for an event, it also makes a direct query to the resource of interest to determine if the resource is already in the desired state. For the expression WiFi.Connected, for example, APE both registers for events regarding changes in Wi-Fi status, and queries Wi-Fi to determine if it is already connected. If so, APE immediately evaluates the expression to true. Otherwise, APE waits for a callback from the system regarding a change in Wi-Fi status and rechecks for the Connected condition.

The APE preprocessor performs syntactic checking and error reporting, with APE terms being checked against an extensible library of terms. The device state primitives supported by APE are specific to each device, but there is a standardized core that encompass the display, cellular, Wi-Fi, and power subsystems of a device and their various features and states.[1] Additional details regarding the efficient implementation of timed automata semantics and the evaluation of state expressions are discussed in the next section.

APE_DefineTerm: The APE_DefineTerm annotation allows a developer to define a new term that can be used in APE annotations throughout the application. Defining new terms not only provides for reuse, but also allows non-experts to utilize policies constructed by others. For example, a new term `MyTerm` may be defined by:

```
@APE_DefineTerm("MyTerm", "Battery.Charging AND
      (WiFi.Connected OR Cell.4G)")
```

and used to construct APE annotations, such as

```
@APE_WaitUntil("{requestsPending()} AND MyTerm", MaxDelay=3600)
```

Any APE recognized primitive or valid Java boolean expression may used in the definition of a new term. Terms do not encode any notion of timing, and are thus not complete policies in themselves, but rather building blocks for higher-level policies.

APE_DefinePolicy In addition to defining new terms, developers may define reusable, high-level policies with the use of the APE_DefinePolicy annotation. The difference between a term, defined using APE_DefineTerm, and a policy is that a policy may encode timing constraints and transitions. Unlike terms, which can be joined together with other terms to form state expressions, a defined policy represents a complete state expression. Transitions may be used to chain policies together. For example, a new policy `MyPolicy` may be defined by:

```
@APE_DefinePolicy("MyPolicy", "(Display.Off,inf),(MyTerm,10)")
```

and used to contruct APE annotations, such as

```
@APE_WaitUntil("MyPolicy,({dataReady()},30)", MaxDelay=3600)
```

The timing parameters in a defined policy act as defaults and may be optionally replaced when referencing a policy:

```
@APE_WaitUntil("MyPolicy(inf,60),({dataReady()},30)",MaxDelay=3600)
```

For many power-management policies, such as those without transitions, simply defining a new term is sufficient.

Annotations are translated at compile-time into runtime requests to the APE middleware service, discussed in Section 4, which is responsible for monitoring device state and resolving policies on behalf of APE-enabled applications. Java annotations are simply a form of metadata added to source code, and thus have no impact on application behavior without the relevant processor interpreting them during the build process. This feature of annotations means that a developer can experiment with a power-management policy and then quickly disable it during testing by simply removing the APE annotation processor from the build process.

[1]APE builds upon Android's Java hardware API specification, which standardizes the names and low-level states of many components.

4. THE APE RUNTIME SERVICE

The APE runtime is responsible for executing APE annotations from multiple APE-enhanced applications. In this section we focus on the key design decisions behind the service and discuss optimizations made to reduce the overhead of executing APE annotations.

The runtime consists of a client library and a middleware service. A single instance of the middleware services, implemented as an Android Service component, runs on a device. The middleware service is responsible for (1) monitoring for changes in hardware state and (2) (re)evaluating APE expressions in response to these changes. APE applications communicate with the middleware through remote procedure calls (RPCs). The details of RPC, including binding to the service, parameter encoding, and error handling, are encapsulated in the client library. Having a single middleware service instance has the advantage of amortizing the overhead associated with policy evaluation over multiple applications. More importantly, this approach allows the middleware to coordinate the activities of multiple clients for added energy savings (shown experimentally in Section 5.2.2).

APE annotations are translated into Java code by the APE preprocessor prior to compilation. Each policy is converted into an equivalent integer array representation so as to avoid string processing at runtime. The generated code relies on three functions provided by the client library: `registerPolicy`, `ifExpression`, and `waitForExpression`. The `registerPolicy` function is executed during the initialization of the application and registers each APE policy with the middleware through RPC calls. The middleware service returns a *policy handler* that can be used by `ifExpression` and `waitForExpression` to refer to a particular policy at runtime. RPCs to the middleware are synchronous, blocking the execution of the calling application thread until they return. Consistent with the model described in Section 3.1, each policy is represented as a timed automaton that is executed by the middleware. An RPC completes when the automaton reaches an accepting state. This triggers the return of the RPC and, subsequently, the execution of the deferred application code.

The generated code is split into initialization segments (`registerPolicy` calls) and policy segments (`ifExpression` and `waitForExpression` calls) in order to reduce runtime overhead. As the overhead associated with RPC is dependent on the size of the request, the potentially large representations of policies are only transmitted once to the middleware using `registerPolicy` calls during initialization. Runtime policy segments utilize policy handlers so as to avoid uploading policies to the middleware multiple times. As policy handlers are implemented as integers, they add only four bytes to the size of a RPC request, thus minimizing runtime overhead. The overhead associated with RPC is further discussed in Section 5.2.1.

For the middleware to execute the timed automatons efficiently, it must track changes in hardware state and update the APE expressions in response in an efficient manner. The monitoring of low-level device state primitives is implemented as components called *device state monitors.* Aside from requiring concurrent programming, device state monitors are challenging to write because they must glean information through somewhat ad hoc mechanisms. For example, the connectivity of the cellular radio is determined by periodically polling the `ConnectivityManager`.

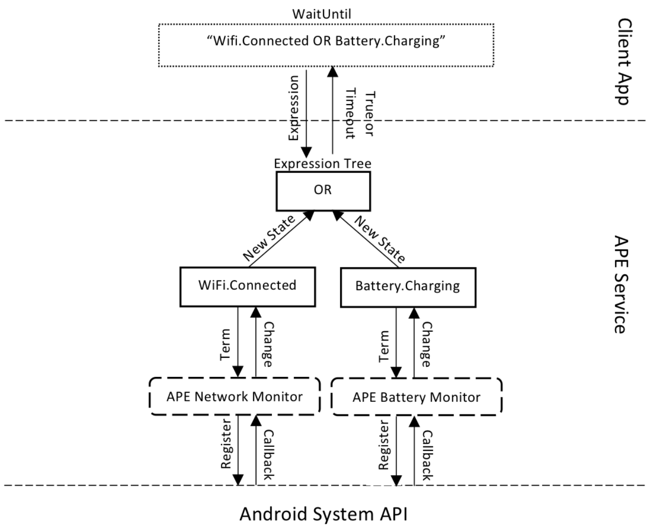

Figure 4: The boolean expression tree representation of a particular APE_WaitUntil request. All leaf nodes in the tree represent primitives in the expression, while all non-leaf nodes represent operators.

However, to determine whether data is transmitted/received, the device monitor must register callbacks with the `TelephonyManager`. Additional connectivity information may also be extracted from `sysfs` – the Linux's standard mechanism for exporting kernel-level information. Our device monitors provide clean APIs that hide the idiosyncrasies of monitoring hardware resources.

In response to a `registerPolicy` call, the middleware generates an equivalent boolean expression tree for each expression. The tree is constructed such that leaves represent terms in the expression and non-leaves are `AND` or `OR` operators. A node maintains a reference to its parent and any children. In addition, a node also maintains a boolean representing the evaluation of its subtree expression. At a high level, the expression trees are evaluated from leaves to the root. The leaves involve low-level states monitored using the device state monitors. These values are propagated up the tree and combined based on the boolean operator (`AND` or `OR`). This approach reduces the cost of evaluating expressions as changes in low-level state often do not require the entire tree to be reevaluated.

Figure 4 provides an example of a simple boolean expression tree in APE. The arrows indicate the flow of information during the evaluation of the boolean expression represented by the tree. The tree is evaluated in one of two ways. In the case of `ifExpression`, the APE service calls a method of the same name on the head node of the tree. When `ifExpression` is called on a non-leaf node, the node calls `ifExpression` on each of its children and applies its operator to the returned values. When `ifExpression` is called on a leaf node, the device monitor associated with the node returns the current state of the hardware device. The value returned by the method call on the head node is in turn returned back to the client application, where the result is used to select which policy, if any, should be applied.

In the case of `waitForExpression`, the APE service again calls a method of the same name on the head node of the tree. Rather than evaluating all nodes immediately, `waitForExpression` notifies all leaf nodes to begin monitoring changes in device state and starts any necessary timers. Leaf nodes register for callbacks from the relevant device monitor about changes regarding the node's term. If necessary, threads are created in the client application to periodically evaluate any local Java code used as part of an annotation and to communicate the result to the APE service. Whenever a node receives information that would change the evaluation of its term or operator, it notifies its parent node of the change. This lazy evaluation of the expression tree from the bottom up ensures that each term and operator in a state expression is only reevaluated when new information that may affect its result is present. To avoid unnecessary message passing and computational overhead, device state monitors only actively monitor the hardware components necessary to resolve all pending requests from leaf nodes. When the head of the expression tree evaluates to be true, or if the tree's timer expires, all leaf nodes are notified to stop monitoring changes by unregistering from their corresponding device state monitor. In the case that a policy consists of multiple expressions, the trees are evaluated in the order that they appear, moving forward to the next tree once the current tree evaluates true, or returning to a previous tree if the current expression times out. Given the synchronous nature of the remote procedure calls, calls to `waitForExpression` will block the calling thread of execution in the client application until the call returns, thus ensuring the costly operation that follows is not executed until the desired conditions have been satisfied.

5. EVALUATION

In this section we evaluate APE from two perspectives. First, we present a case study of introducing power management into the CitiSense CRM application, both with and without the use of APE. We then examine the performance of the middleware runtime and show that APE effectively reduces power-consumption by coordinating the workloads of multiple applications, while requiring fewer changes to the original source code and having negligible runtime performance overhead.

5.1 Case Study: CitiSense

The authors have developed a variety of CRM applications, notably CitiSense, which monitors, records, and shares a user's exposure to air pollution using their smartphone and a Bluetooth enabled sensor device [21]. Building an application that performed all the required tasks without depleting the smartphone's battery proved challenging, as the application depended heavily on the use of GPS for localization, Bluetooth for sensor readings, and cellular communication to upload measurements to a server for further processing. Much of the challenge in improving CitiSense arose from adding, evaluating, and iteratively revising the application's already-complex code base to implement energy-management policies. In this section, we share our experience in implementing a policy for uploading sensor data from the CitiSense mobile application to a remote server, providing both hand-coded and APE implementations.

The initial implementation of the CitiSense mobile application cached air quality measurements on the user's device and attempted to upload all stored readings once every twenty minutes. If the loss of connectivity caused a trans-

```
Thread uploadThread = new Thread(new Runnable() {
  while(true) {
    try {
     Thread.sleep(120000);
    } catch(InterruptedException e) {}
    attemptUpload();
  }
});
uploadThread.start();
```

Figure 5: Example of a naive implementation of sensor reading uploading in CitiSense. A thread thread wakes every twenty minutes to attempt uploading any stored sensor readings before returning to sleep.

```
Thread uploadThread = new Thread(new Runnable() {
  while(true) {
    Intent batt = context.registerReceiver(null,
        new IntentFilter(Intent.ACTION_BATTERY_CHANGED));
    int lvl = batt.getIntExtra(BatteryManager.EXTRA_LEVEL,-1);
    int scl = batt.getIntExtra(BatteryManager.EXTRA_SCALE,-1);
    float batteryPct = lvl / (float) scl;
    try {
      if(batteryPct > 70){ Thread.sleep(120000); }
      else{ Thread.sleep(360000); }
    } catch(InterruptedException e) {}
    attemptUpload();
  }
});
uploadThread.start();

TelephonyManager teleManager = (TelephonyManager)
  context.getSystemService(Context.TELEPHONY_SERVICE);
TransListener transListener = new TransListener();
teleManager.listen(transListener,
  PhoneStateListener.LISTEN_DATA_ACTIVITY);

private class TransListener extends PhoneStateListener {
  public void onDataActivity(int act) {
    if(act == TelephonyManager.DATA_ACTIVITY_IN
        || act == TelephonyManager.DATA_ACTIVITY_OUT
        || act == TelephonyManager.DATA_ACTIVITY_INOUT) {
      uploadThread.interrupt();
    }
  }
}
```

Figure 6: An improved implementation of uploading in CitiSense that is both battery-life and cellular radio state aware.

mission to fail, then the data would be preserved until the next upload window. Although timed batching saves energy, the approach still has several drawbacks. Uploads attempted while a user's phone had a weak cellular network signal often failed, but still incurred high energy consumption during the failed attempt. Additionally, even if connectivity was available nineteen out of every twenty minutes, the lack of connectivity at the twentieth minute mark meant the upload would be delayed until the next attempt. For some users with unreliable cellular coverage, it often took hours before their data was uploaded successfully to the server.

To improve the energy-efficiency and reliability of CitiSense uploads, the application was modified to attempt a transmission whenever the phone's cellular radio was detected to already be transmitting or receiving data (See Figure 6). The equivalent timed automaton for this policy is presented in Figure 7. The concept is that if the phone detects that another application on the phone has successfully sent or received data over the cellular radio, then CitiSense would also likely succeed. Additionally, with radio already being active, CitiSense would no longer be forcing the radio out of a low-power idle state: the application is taking

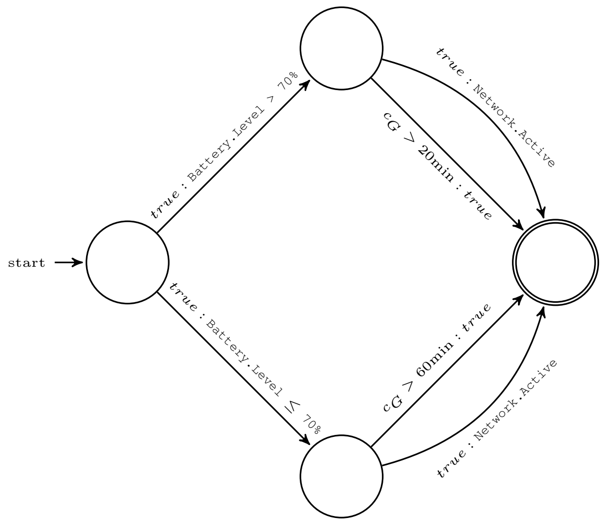

Figure 7: The equivalent timed automaton for the policy implemented in Figure 6.

advantage of other workloads waking the radio, reducing its impact on device battery life. In the event that no other application wakes the radio in a timely manner, CitiSense attempts to upload any stored readings after a timeout. However, unlike the static twenty-minute timer used in the first implementation, this one varies timeout length based on the remaining battery life of the device. If remaining battery life is greater than 70% the application waits up to twenty minutes for the radio to become active; otherwise, CitiSense will wait up to one hour.

Unfortunately, what was once a simple, nine-line implementation now requires querying the Android system for the status of the battery, registering the application for callbacks regarding changes in cellular data activity in the system, and implementing a custom `PhoneStateListener` to handle callbacks and to interrupt the sleeping upload thread if data activity is detected. Further extending the implementation to be dependent on the state or availability of other resources, such as a Wi-Fi connection, would require implementing additional listeners to handle callbacks and additional concurrency management. Given the large number and complexity of changes required, prototyping and experimenting with a variety of potential policies becomes time consuming and burdensome. Even a well-thought-out policy can perform poorly in practice and require tweaks or major changes.

When we reimplemented this policy in APE, the code collapses back to nine lines, with the 19 lines of policy code being reduced to three lines of APE annotations:

```
Thread uploadThread = new Thread(new Runnable() {
  while(true) {
    @APE_If("Battery.Level > 70%")
        @APE_WaitUntil("Network.Active", MaxDelay=1200)
    @APE_Else() @APE_WaitUntil("Network.Active", MaxDelay=3600)
    attemptUpload();
  }
});
uploadThread.start();
```

The developer-implemented `PhoneStateListener`, event handling, and thread concurrency management are now handled by the APE middleware. In this compact, declarative format, it is now possible to read the policy at a glance, and to attempt variants of the policy quickly.

The APE policy managing uploads can be rapidly extended to also consider the quality of the cellular connection and the availability of Wi-Fi:

```
Thread uploadThread = new Thread(new Runnable() {
  while(true) {
    @APE_If("Battery.Level > 70%")
      @APE_WaitUntil("WiFi.Connected OR
        Network.Active AND (Cell.3G OR Cell.4G)",MaxDelay=1200)
    @APE_ElseIf("Battery.Level > 30%")
      @APE_WaitUntil("WiFi.Connected OR
        Network.Active AND (Cell.3G OR Cell.4G)",MaxDelay=2400)
    @APE_Else()
      @APE_WaitUntil("WiFi.Connected OR
        Network.Active AND (Cell.3G OR Cell.4G)",MaxDelay=3600)
    attemptUpload();
  }
});
uploadThread.start();
```

Instead of waiting for simply any cellular network activity, CitiSense now uploads sensor readings only while connected to a Wi-Fi network or if cellular activity was observed while connected to either a 3G or 4G cellular network. The maximum time to wait for such a state was set to be 20 minutes if remaining battery life was greater than 70%, 40 minutes if between 70% and 30%, and 60 minutes if less than 30%.

With the use of APE, the new energy-management policy can be expressed in a total of six annotations. In contrast, an experienced Android developer implementing the same policy by hand required 46 lines, including five lines for suspending and waking threads, three lines to register the application for callbacks regarding changes in device state from the Android system, and 26 lines and one new class for handling the callbacks. The thread responsible for uploading sensor readings is put to sleep until it is interrupted by a different thread which handles callbacks from the Android system and determines when all required resources are available. Not only is this implementation much longer than the APE based implementation, it is significantly more difficult to understand and maintain. This shows that APE not only represents power management policies concisely, but also significantly reduces the implementation complexity by removing the need to write error-prone concurrent code.

5.2 System Evaluation

In this section we evaluate APE by examining the overhead associated with communicating requests to the middleware service. Additionally, we present power savings achieved by using APE to implement a simple resource-aware energy-management policy in an application that makes regular use of network communication. All experiments were run on a Pantech Burst smartphone running Android version 2.3.5. The power consumption of the device was measured using a Power Monitor from Monsoon Solutions [3]. The battery of the Pantech Burst was modified to allow a direct bypass between the smartphone and the power monitor, allowing power to be drawn from the monitor rather than the battery itself. Traces of this power consumption were collected on a laptop connected to the power monitor over USB. Measurements involving network communication were run on the AT&T cellular network in the San Diego metropolitan area. Though the Pantech Burst supports LTE, all experiments were run while operating on AT&T's HSPA+ network as LTE coverage was not available at the site of our experiments.

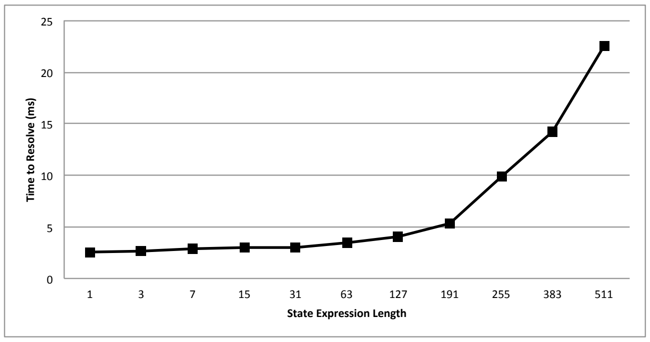

Figure 8: Time to register expressions of various lengths using `registerPolicy`. As the size of the expression grows, the size of the message passed over IPC begins to impact latency.

5.2.1 APE Overhead

To evaluate the overhead associated with using APE, we examine the time required to complete a simple request to the middleware service. Additionally, we examine the impact of expression length on the latency of `registerPolicy` requests. A simple Android application was built that performed no operations other than to execute the code being benchmarked. The time required to execute code segments was measured by taking the difference between calls to `System.nanoTime()` placed just before and after the code segment.

To evaluate the latency overhead associated with using APE, we measured the time required to check the current status of data activity on the device using the standard Android API and using APE. Checking the current status of data activity using the standard Android API was done using the following code:

```
TelephonyManager telMan = (TelephonyManager)
   getSystemService(Context.TELEPHONY_SERVICE);
int dataAct = telMan.getDataActivity();
```

The average time required to execute this code was measured to be approximately 0.79 ms. Checking the current status using APE was implemented using the following annotation:

```
@APE_If("Network.Active")
```

The average time required to check for data activity using APE was measured to be approximately 2.5 ms, meaning approximately 1.71 ms were spent sending the request to the APE service over IPC, evaluating a single-term expression tree, and returning a message to the client application over IPC. Given that a developer would use APE to shape delay-tolerant workloads, we believe that an overhead of 1.71 ms is negligible, especially when compared to the time that will be spent waiting for ideal conditions.

To evaluate the impact of expression length on the time required to register a policy, calls to `registerPolicy` using expressions of various lengths were measured using our test application. Expressions were constructed using a chain of `Network.Active` and `AND` terms. As observed in Figure 8, the time to register policies remains fairly constant at lower expression lengths. It is only when expressions begin to become longer than 127 terms that the overhead associated with passing large messages over IPC begins to take its toll. Messages are passed between processes using a buffer

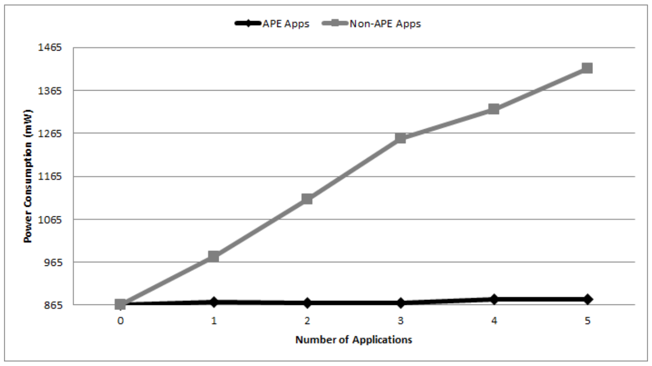

Figure 9: Increase in system power-consumption due to the introduction of additional applications that periodically make use of network resources. APE enhanced applications effectively recognize opportunities to transmit data efficiently, only marginally increasing power consumption.

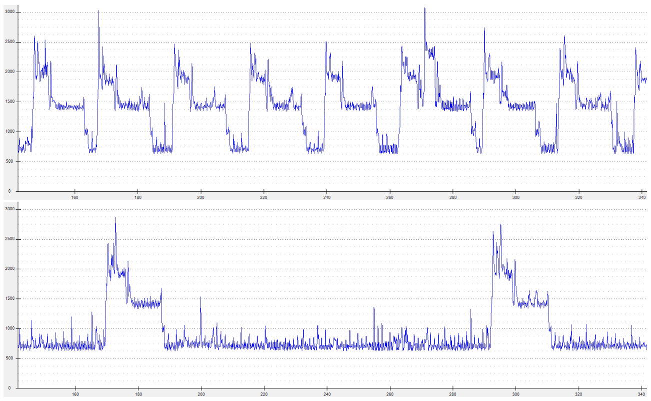

Figure 10: Power consumption (mW) traces from a smartphone device running a variety of naive (top) and APE enhanced (bottom) CRM applications.

in the Android kernel. If messages become sufficiently large, they require additional buffer space to be allocated in the kernel, thus introducing additional latency in resolving requests. However, expressions of such length are unlikely to arise in practice as realistic energy-management policies depend on significantly fewer application events. In the experience of the authors, most APE expressions tend to be between one and nine terms. As these requests are completed only once, at the start of an application, their overhead is considered acceptable, even at long expression lengths.

5.2.2 *Power Savings*

To demonstrate the potential impact of CRM applications on the battery life of a device, power measurements were collected from a smartphone running an instance of the CitiSense application, which fetched data from a remote server once every two minutes. As a baseline, we measured the power consumption of the phone, while powering its display at maximum brightness and running a single instance of CitiSense, to be 865.33 mW. Up to five additional instances of CitiSense were then introduced to the system. The power consumed by these applications when their use of network resources does not overlap, presented in Figure 9, reached as high as 1416.39 mW, a 63.7% increase. This is a worst-case scenario, as there is no coordination with existing workloads on the device to ensure efficient usage of resources.

To demonstrate the potential savings of using APE to implement even a simple energy-management policy, the applications were each modified using a single APE_WaitUntil annotation to wait up to 120 seconds for the cellular radio to be woken before attempting transmission. As observed in Figure 9, the introduction of this annotation significantly reduced the power consumption of additional CRM workloads; adding five additional APE enhanced instances of the CitiSense application increased power consumption by only 13.49 mW, or 1.6%. As can be seen in Figure 10, APE is able to effectively coordinate the workload of the CRM applications to minimize the number of times the cellular radio is woken and put into a high-power state. If no background application had been running on the device and transmitting data, then the first APE enhanced application to timeout would wake the radio to transmit its request. The other APE applications would then have detected this event and transmitted at the same time for nearly no additional energy cost. This experiment shows that APE provides effective means of coordinating power management across applications to achieve significant energy savings.

6. CONCLUSION

Annotated Programming for Energy-efficiency (APE) is a novel approach for specifying and implementing system-level power management policies. APE is based on two key insights: (1) Power management policies defer the execution of power hungry code segments until a device enters a state that minimizes the cost of that operation. (2) The desired states when an operation should be executed can be effectively described using an abstract model based on timed automata. We materialized these insights in a small, declarative, and extensible annotation language and runtime service. Annotations are used to demarcate expensive code segments and allow the developer to precisely control delay and select algorithms to save power.

We showed our approach to be both general and expressive, in that it can replicate many previously published policies and that its use reduced the complexity of power management in CitiSense. The APE middleware's use of techniques like code generation, policy handlers, lazy evaluation, and encoding policies as integer arrays kept overhead below 1.7 ms for most requests to the service. In our benchmarks, APE provided power savings of 63.7% over an application that did not coordinate access to resources.

Tools for assisting developers in reasoning about appropriate places to apply APE annotations are currently under development. We are also exploring the use of a type system for designating delay-(in)tolerant data in an application. A user study of experienced developers will be conducted to examine how well developers new to APE adapt to using annotations to express power-management policies.

7. ACKNOWLEDGMENTS

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).

8. REFERENCES

[1] AndroidAnnotations. `http://androidannotations.org/`.

[2] Google Guice. `https://code.google.com/p/google-guice/`.

[3] Monsoon Solutions - Power Monitor. `http://msoon.com/LabEquipment/PowerMonitor/`.

[4] Roboguice: Google Guice on Android. `https://github.com/roboguice/roboguice`.

[5] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[6] M. Azizyan, I. Constandache, and R. Roy Choudhury. Surroundsense: mobile phone localization via ambience fingerprinting. In *Proceedings of the 15th Annual International Conference on Mobile Computing and Networking*, pages 261–272. ACM, 2009.

[7] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, 2000.

[8] O. Chipara, C. Lu, J. Stankovic, and G. Roman. Dynamic conflict-free transmission scheduling for sensor network queries. *IEEE Transactions on Mobile Computing*, 10(5):734–748, 2011.

[9] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. In *ACM SIGPLAN Notices*, volume 47, pages 831–850. ACM, 2012.

[10] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[11] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. *ACM SIGPLAN Notices*, 35(1):39–52, 2000.

[12] S. S. Hasan, F. Lai, O. Chipara, and Y.-H. Wu. Audiosense: Enabling real-time evaluation of hearing aid technology in-situ. In *26th International Symposium on Computer-Based Medical Systems (CBMS)*, pages 167–172. IEEE, 2013.

[13] A. J. Khan, K. Jayarajah, D. Han, A. Misra, R. Balan, and S. Seshan. Cameo: A middleware for mobile advertisement delivery. In *Proceeding of the 11th International Conference on Mobile Systems, Applications, and Services*, pages 125–138. ACM, 2013.

[14] S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In *ACM SIGPLAN Notices*, volume 37, pages 231–245. ACM, 2002.

[15] M. B. Kjærgaard, S. Bhattacharya, H. Blunck, and P. Nurmi. Energy-efficient trajectory tracking for mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 307–320, New York, NY, USA, 2011. ACM.

[16] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. The jigsaw continuous sensing engine for mobile phone applications. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, pages 71–84. ACM, 2010.

[17] E. Miluzzo, N. D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S. B. Eisenman, X. Zheng, and A. T. Campbell. Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application. In *Proceedings of the 6th ACM conference on Embedded Network Sensor Systems*, pages 337–350. ACM, 2008.

[18] P. Mohan, V. N. Padmanabhan, and R. Ramjee. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, pages 323–336. ACM, 2008.

[19] M. Musolesi, M. Piraccini, K. Fodor, A. Corradi, and A. T. Campbell. Supporting energy-efficient uploading strategies for continuous sensing applications on mobile phones. In *Pervasive Computing*, pages 355–372. Springer, 2010.

[20] S. Nath. Ace: exploiting correlation for energy-efficient and continuous context sensing. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pages 29–42. ACM, 2012.

[21] N. Nikzad, N. Verma, C. Ziftci, E. Bales, N. Quick, P. Zappi, K. Patrick, S. Dasgupta, I. Krueger, T. v. Rosing, and W. G. Griswold. Citisense: Improving geospatial environmental assessment of air quality using a wireless personal exposure monitoring system. In *Proceedings of the Conference on Wireless Health*, WH '12, pages 11:1–11:8, New York, NY, USA, 2012. ACM.

[22] N. Nikzad, J. Yang, P. Zappi, T. S. Rosing, and D. Krishnaswamy. Model-driven adaptive wireless sensing for environmental healthcare feedback systems. In *IEEE International Conference on Communications (ICC)*, pages 3439–3444. IEEE, 2012.

[23] 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 *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pages 267–280. ACM, 2012.

[24] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 95–107. ACM, 2004.

[25] D. Quinlan, M. Schordan, R. Vuduc, and Q. Yi. Annotating user-defined abstractions for optimization. In *20th International Parallel and Distributed Processing Symposium, 2006*, pages 8–pp. IEEE, 2006.

[26] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[27] V. Srinivasan, G. R. Shenoy, S. Vaddagiri, D. Sarma, and V. Pallipadi. Energy-aware task and interrupt management in linux. In *Ottawa Linux Symposium*, 2008.

[28] V. Venkatachalam and M. Franz. Power reduction techniques for microprocessor systems. *ACM Computing Surveys (CSUR)*, 37(3):195–237, 2005.

[29] Y. Wang, J. Lin, M. Annavaram, Q. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh. A framework of energy efficient mobile sensing for automatic user state recognition. In *Proceedings of the*

7th International Conference on Mobile Systems, Applications, and Services, pages 179–192. ACM, 2009.

[30] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li. Optimizing background email sync on smartphones. In *Proceeding of the 11th International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 55–68, New York, NY, USA, 2013. ACM.

[31] W. Ye, F. Silva, and J. Heidemann. Ultra-low duty cycle mac with scheduled channel polling. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pages 321–334. ACM, 2006.