UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Annotated Programming for Energy-Efficiency in Mobile Applications

Permalink
https://escholarship.org/uc/item/3t56h5hb

Author
Nikzad, Nima

Publication Date
2015

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Annotated Programming for Energy-Efficiency in Mobile Applications

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Nima Nikzad

Committee in charge:

Professor William G. Griswold, Chair
Professor Octav Chipara
Professor Ryan Kastner
Professor Sorin Lerner
Professor Kevin Patrick
Professor Ramesh Rao

2015

Copyright

Nima Nikzad, 2015

All rights reserved.

The Dissertation of Nima Nikzad is approved and is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____

____

Chair

University of California, San Diego

2015

TABLE OF CONTENTS

Signature Page iii

Table of Contents iv

List of Figures vi

List of Tables x

Acknowledgements xi

Vita xii

Abstract of the Dissertation xiii

Chapter 1 Introduction 1
- 1.1 Case Study: Composing a Power-Management Policy in a CRM Application 4
- 1.2 Related Work 7
- 1.3 Annotated Programming for Energy-Efficiency 12

Chapter 2 Annotated Programming for Energy-Efficiency 18
- 2.1 APE Design Overview 18
  - 2.1.1 The APE Policy Model 19
  - 2.1.2 The APE Annotation Language 25
- 2.2 The APE Runtime Service 30
- 2.3 Evaluation 34
  - 2.3.1 Case Study: CitiSense 35
  - 2.3.2 System Evaluation 40
- 2.4 Conclusion 44
- 2.5 Acknowledgments 45

Chapter 3 Satisfying Delay Constraints 46
- 3.1 Annotation Semantics 46
  - 3.1.1 `Wait` 46
  - 3.1.2 `DelayableUpto` 46
  - 3.1.3 Example 48
- 3.2 Static Analysis and Run-time Monitoring 50
  - 3.2.1 Algorithms for Static Analysis and Monitoring 50
  - 3.2.2 Run-time Optimizations 53
- 3.3 Policy Generation Engine 54
- 3.4 Evaluation 58
  - 3.4.1 Accuracy of Policy Generation Engine 58

3.4.2 Interaction of User Experience with Power Management 61
3.4.3 Runtime Overhead 64
3.5 Conclusion 65
3.6 Acknowledgements 66

Chapter 4 Ensuring Timely Delivery of Delay-Sensitive Objects 67
4.1 Annotation Semantics 69
4.2 System Design and Implementation 74
4.2.1 Budget Tracking 76
4.2.2 Policy Evaluation 83
4.3 Case Study 84
4.3.1 NPR News 84
4.3.2 CitiSense 88
4.4 Experiments 91
4.4.1 Power-Timeliness Trade-off 92
4.4.2 Budget Tracking Overhead 103
4.5 Conclusion 106
4.6 Acknowledgements 107

Chapter 5 Conclusion 108

Bibliography 111

LIST OF FIGURES

Figure 1.1. Power consumption (mW) trace of a smartphone device running six CRM applications that each periodically download a resource using the cellular radio without any form of coordination. 2

Figure 1.2. Power consumption (mW) trace of a smartphone device running six CRM applications when each application attempts to piggyback transmissions by watching for the cellular radio to be woken by another application. 3

Figure 1.3. 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. . . . 4

Figure 1.4. An improved implementation of uploading in CitiSense that is both battery-life and cellular radio state aware. 5

Figure 1.5. Using APE to implement the same power-management policy found in Figure 1.4. 13

Figure 1.6. Using Tempus to specify a policy that delays the processing of images *except* when the image is going to be used to update the user interface. 17

Figure 2.1. Defer uploads, for up to 30 minutes, until the Wi-Fi radio has connected to a network. 21

Figure 2.2. Defer sensor sampling until movement is first detected by the accelerometer and driving is confirmed using the GPS. 22

Figure 2.3. 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. 25

Figure 2.4. The boolean expression tree representation of a particular Wait request. All leaf nodes in the tree represent primitives in the expression, while all non-leaf nodes represent operators. 33

Figure 2.5. 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. . . . 35

Figure 2.6. An improved implementation of uploading in CitiSense that is both battery-life and cellular radio state aware. 37

Figure 2.7. The equivalent timed automaton for the policy implemented in Figure 2.6. 38

Figure 2.8. 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. 42

Figure 2.9. Increase in system power-consumption due to the introduction of additional applications that periodically make use of network resources. 43

Figure 2.10. Power consumption (mW) traces from a smartphone device running a variety of naive (top) and APE enhanced (bottom) CRM applications. 44

Figure 3.1. A thread attempts to download data to displayed within the application. The `@DelayableUpTo` annotation ensures that the downloading and displaying of the data is not delayed by APE by more than one minute. 49

Figure 3.2. Algorithm for instrumenting application code with delay allowances. 52

Figure 3.3. Instrumentation for assigning, clearing, and using delay allowances. 53

Figure 3.4. 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. 56

Figure 3.5. A trace of power consumption on a smartphone device while using the `NPR News` application to listen to an audio story. 63

Figure 3.6. Power consumption in various versions of the `NPR News` and `AndStatus` applications. 64

Figure 4.1. Screenshots from the CitiSense environmental air pollution monitoring application: the most recent Air Quality Index score (left) and detailed pollutant report (right). 68

Figure 4.2. The execution of three concurrent `@Wait` annotations on threads τ_1, τ_2, and τ_3 that include the same object o in their scope. 72

Figure 4.3. The @DelayBudget annotation (line 7) ensures that the upload of HealthReport objects related to critical health events are not subject to any Tempus-introduced delays (line 25) when uploaded (line 33). 75

Figure 4.4. @DelayBudget annotations are translated into runtime calls to the Tempus service that that assign a budget to a particular object and begin tracking it. The first parameter is an automatically-generated label. 77

Figure 4.5. Translation of the @Wait annotation. See text for explanation. . . . 77

Figure 4.6. Formalization of the single-threaded behavior related to object budget tracking in the Tempus runtime. 78

Figure 4.7. Algorithm for static program analysis that returns the set of object labels to be considered at each @Wait annotation in the program. 83

Figure 4.8. Screenshots from the NPR News application: looking at the list of stories (left) and reading a story (right). 85

Figure 4.9. The download of images is deferred until the radio or WiFi are on to save energy. 86

Figure 4.10. Budgets for the @Wait annotation in Figure 4.10 are refined to ensure that the first 8 stores are not delayed and the budget of the remaining stores is computed dynamically based on their position in the news story list. 88

Figure 4.11. The application has two power management policies: (1) data acquisition is deferred until the user changes his location and (2) sensor readings are classified as either NormalReading or UrgentReading. 90

Figure 4.12. Impact of various policies on the NPR News application when piggybacking opportunities are available once every three minutes. 96

Figure 4.13. Impact of various policies on the NPR News application when piggybacking opportunities are available once every one minute. . 98

Figure 4.14. The state of budgets assigned to various images during a run of the NPR News application with the Light Prefetch policy. 99

Figure 4.15. Average power consumption in the Citisense application when processing traces from stationary and mobile users. 102

Figure 4.16. Overhead associated with the tracking of budgets for various numbers of objects. Note the logarithmic scale. 104

Figure 4.17. Garbage collection overhead when Tempus is tracking objects that are either all referenced elsewhere in the program or are all garbage collected. 106

LIST OF TABLES

Table 3.1. Searching for opportunities to reduce network-use related energy-consumption 60

ACKNOWLEDGEMENTS

I would like to thank Professors William Griswold and Octav Chipara. Working with these two men has been the highlight of my career. There was something about the different perspectives and backgrounds we each brought to the table that kept working together fresh and, frankly, a great deal of fun. I know that without the support of these gentlemen, I would never have reached this milestone.

I would also like to thank my friends and family for their love and support. Without all of you, I would have gone absolutely and completely mad years ago. Thanks to your support, I am instead only moderately insane.

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of the 36th International Conference on Software Engineering. Nikzad, Nima; Chipara, Octav; Griswold, William G. 2014. The dissertation author was the primary investigator and author of this material.

Chapter 3, in part is currently being prepared for submission for publication of the material. Nikzad, Nima; Chipara, Octav; Griswold, William G. The dissertation author was the primary investigator and author of this material.

Chapter 4, in part is currently being prepared for submission for publication of the material. Nikzad, Nima; Chipara, Octav; Griswold, William G. The dissertation author was the primary investigator and author of this material.

VITA

2009	Bachelor of Science, University of California, Los Angeles
2009–2014	Research Assistant, University of California, San Diego
2011–2012	Teaching Assistant, Department of Computer Science and Engineering University of California, San Diego
2012	Master of Science, University of California, San Diego
2015	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

"Model-driven Adaptive Wireless Sensing for Environmental Healthcare Feedback Systems." International Conference on Communications. IEEE, 2012.

"CitiSense: Improving Geospatial Environmental Assessment of Air Quality Using a Wireless Personal Exposure Monitoring System." Proceedings of the Conference on Wireless Health. ACM, 2012.

"APE: An Annotation Language and Middleware for Energy-Efficient Mobile Application Development." Proceedings of the 36th International Conference on Software Engineering. ACM, 2014.

"Ensuring Timely Object Delivery in Energy-Efficient Mobile Applications." *Under Review.* Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems. ACM, 2015.

FIELDS OF STUDY

Major Field: Computer Science

Studies in Software Engineering
Professor William G. Griswold

Studies in Embedded Systems
Professor Octav Chipara

ABSTRACT OF THE DISSERTATION

Annotated Programming for Energy-Efficiency in Mobile Applications

by

Nima Nikzad

Doctor of Philosophy in Computer Science

University of California, San Diego, 2015

Professor William G. Griswold, Chair

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. Not only does this require a developer to have a strong understanding of hardware and how various operations impact resource usage, but it involves the use of complex primitives and writing error-prone multithreaded code to monitor hardware state.

To address this problem, this dissertation presents Annotated Programming for Energy-efficiency (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. The APE Policy Generation Engine can automatically identify operations in an application that utilize power-hungry resources, generate an APE-based power-management policy, and provide feedback to the developer regarding available options for policy customization. Various language constructs and static program analyses allow a developer to specify constraints for delay-sensitive operations and data in the application, while runtime support ensures that specified constraints are satisfied.

Several case studies of applying APE to real mobile sensing applications demonstrate the expressive power of the APE approach and show 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.

Chapter 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. For example, services like Dropbox [8] and BitTorrent Sync [6] keep important documents synchronized across multiple devices. Personal health and context-monitoring applications, such as Fitbit [9], CitiSense [31], AudioSense [24], CenceMe [28], SurroundSense [19], BeWell+ [27], and Ohmage [14], often collect sensor data that is periodically uploaded to a remote server.

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. The cellular radio is an example of such a resource. When not in use, the radio remains in a low-power, idle state where power consumption is on the order of 5 mW. Initiating a transmission wakes the radio, which in turn requests a dedicated channel from the cellular network to allow communication. While in this connected state, the radio can consume on the order of 1 to 2 W of power. The radio remains in this connected state for approximately five to ten seconds after the *last* transmission has been completed, in hopes of catching other upcoming transmissions and avoiding the network overhead of

Figure 1.1. Power consumption (mW) trace of a smartphone device running six CRM applications that each periodically download a resource using the cellular radio without any form of coordination.

establishing a connection. If no other transmissions occur, the radio then transitions to an intermediate power-state (consuming on the order of 500 mW of power) for another ten to fifteen seconds before finally transitioning back to the low-power, idle state. Due to this behavior of the cellular radio, a single, small transmission can drastically increase the power-consumption on a smartphone device for fifteen to twenty seconds, even though the actual transmission of data was completed in a fraction of a second.

In reality, a user is likely to be running several such CRM applications at a time: an e-mail client, news feed reader, social networking applications, and so on. In the worst-case scenario, each application will begin to download or upload data *just as* the radio has finished transitioning back to the idle state after handling another applications transmission request. Even though very little data may be actually transmitted during each operation, the timing of the requests can have a drastic impact on power-consumption, and therefore battery life, on a device. Figure 1.1 visualizes the power-consumption on a smartphone in such a situation: a number of applications, each utilizing the cellular radio, are causing the component to frequently wake up and consume large amounts of power.

Ideally, the transmission requests of multiple applications would be coordinated such that they occur at, or near, the same time, as this would minimize the number

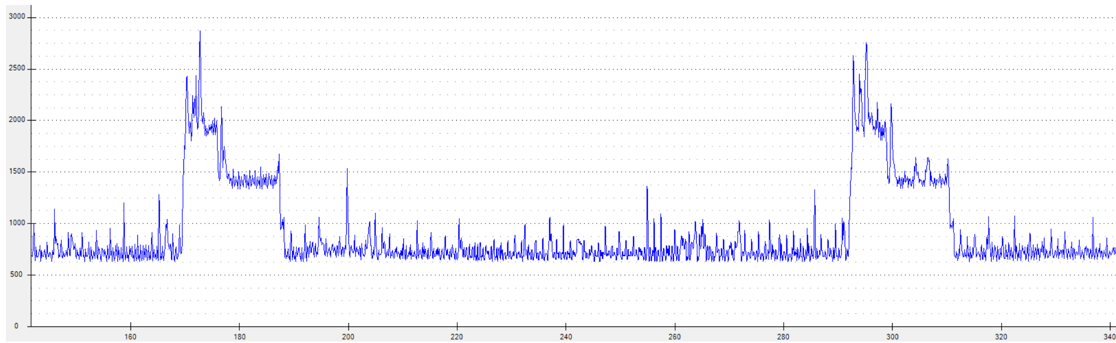

Figure 1.2. Power consumption (mW) trace of a smartphone device running six CRM applications when each application attempts to piggyback transmissions by watching for the cellular radio to be woken by another application.

of times the radio has to be woken and maximizes the time that the radio spends in a low-power, idle state. Figure 1.2 presents a trace of power-consumption for modified versions of the six CRM applications previously presented, where each application waits up to two minutes for another application to wake the radio before proceeding with transmissions. While batching transmissions in such a way increases the amount of data being transmitted at once, and therefore increases the amount of time needed to complete the transmissions, the long tail of power consumption associated with waking the radio is amortized across multiple applications. In fact, mobile application development 'best practice' guides from both Google [2] and AT&T [7] highly recommend the use of batching and piggybacking as a means of significantly reducing the power consumption of applications with periodic workloads.

Unfortunately, the development and introduction of power-management policies that monitor and react to changes in hardware state, especially in large and mature applications, can be extremely challenging for a developer. To demonstrate this challenge, a brief case study of introducing such optimizations into the CitiSense application is provided below.

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

Figure 1.3. 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.

1.1 Case Study: Composing a Power-Management Policy in a CRM Application

The author has 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 [31]. 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, the author shares his experience in implementing a policy for uploading sensor data from the CitiSense mobile application to a remote server.

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 transmission to fail, then the data would be preserved until the next upload window. Although timed batching saves energy,

```
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 1.4. An improved implementation of uploading in CitiSense that is both battery-life and cellular radio state aware.

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 1.4). 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 the radio already being active, CitiSense would no longer be forcing the radio out of a low-power idle state: the application is taking 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 of complex 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.

Challenges. Beyond the expected algorithmic and systems challenges of designing a power management policy, 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, 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.

1.2 Related Work

A great deal of research exists on how to build more energy-efficient software. In recent years, this area has been subject to increased interest as smartphones and other battery-powered computing devices have grown more prevalent. 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. Such optimizations must be implemented at a low-level in the system as they operate at sub-millisecond frequencies and are, therefore, typically the responsibility of device and operating system vendors. Examples of such techniques include dynamic voltage and frequency scaling (see [39] for

a review), tickless kernel implementations [37], low-power listening [33] and scheduled transmissions for radios [20, 41], and batching of I/O operations for devices such as flash memory storage [40].

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 than low-level optimizations and include techniques such as workload shaping, sensor fusion, and filtering. A workload shaping policy, such as delaying large network operations until a Wi-Fi connection is available, is found in applications such as Google Play Market, Facebook, and Dropbox. While low-level optimizations typically impact *how* a particular sub-component operates, system-level optimizations attempt to save power by adjusting *when* and on *what* sub-components operate. For example, while a low-level optimization may control how the Wi-Fi radio in a device determines when it may transmit packets, a system-level optimization may control the amount of data being transmitted by an application and at what frequency. Such optimizations are typically the responsibility of application developers and are built into individual applications, as these policies are highly dependent on the requirements of the applications that they impact.

While effective, such optimizations can be challenging to understand and implement for application developers with little understanding of hardware and low-level system primitives. To address this challenge, several projects have attempted to provide developers with more accessible, high-level means of expressing power-management policies.

Energy Types is a type system based approach that aides the development of energy-efficient applications by allowing a developer to specify discrete *phases* and *modes* of application behavior, which are in turn used to manage CPU voltage scaling and application fidelity at runtime [21]. A developer uses *phases* to describe the workload

characteristics of various operations in their application, such as whether they are CPU- or I/O-bound. This phase information is then used by a compiler to generate instructions that manage CPU voltage scaling at various points in the application. For example, the CPU may be scaled down to a low-power, low-frequency state whenever the application enters a I/O-bound phase of execution. *Modes*, on the other hand, are used to specify different implementations of or input parameters to a particular operation, such that each mode has different energy-consumption characteristics and quality-of-service implications. For example, a developer may provide multiple implementations of image processing and specify that the runtime choice of implementation should be dependent on the remaining battery-life of a device.

One of the claimed benefits of the Energy Types approach is that it encourages more energy-efficient development by requiring developers to think about how their application breaks down into various phases of execution and to structure their application in a way that reflects this. However, this benefit is also one of the approaches greatest weaknesses: reasoning about phases, and structuring an application to reflect these phases, can be extremely burdensome for large applications and attempting to apply Energy Types to an existing application would likely require significant refactoring. Additionally, CPU frequency scaling affects not only the application built using Energy Types, but also affects all other workloads on the system. While Energy Types would allow a developer to specify different network-use policies as *modes* that are selected based on the available battery life on a device, it does not ease the implementation of device state monitoring or of controlling the timing of network accesses. It would not be appropriate for the Citisense application to scale down CPU frequency while uploading sensor data in the background as this may negatively impact the performance of any application the user is currently interacting with in the foreground. Such system-wide impacting decisions are

poor suited for CRM applications, which typically run in the background while the user may be utilizing other applications.

EnerJ employs a type system that allows developers to specify which pieces of data in their application may be subject to approximate storage and processing to save energy [36]. Broadly speaking, approximate computing involves the use of imprecise, but efficient, algorithms and hardware resources to come up with solutions that are 'good enough' for a particular problem. For example, an approximate storage unit may occasionally flip a bit and alter a stored value, but require less power than a precise storage unit. Approximate algorithms may return a sub-optimal result when compared to a precise solution for a problem, but they do so with fewer iterations and energy-consumption. Within particular problem domains, such as image processing, such errors are acceptable and worth the trade-off between accuracy and power.

EnerJ guarantees the isolation of precise and approximate components and requires the developer to explicitly specify when it is acceptable to use approximate data to affect a precise piece of data or state. In other words, EnerJ allows a developer to specify which data in their application must be precise, which data may be approximated, and the rules that dictate how the two types of data interact. While EnerJ is designed to take advantage of unreliable and energy-efficient storage and processing elements, which are not currently available on consumer devices, it also allows a developer to specify approximate versions of algorithms that can be called when operating on approximate pieces of data.

The primary benefit of EnerJ is that its type system statically enforces that a developer does not inadvertently use approximate data to impact the state of precise data. It provides a simple and clean way of specifying which operations may be subject to any hypothetically available efficient-but-error-prone hardware components. While EnerJ ensures that approximate data makes use of approximate versions of algorithms when

possible, it requires these approximate methods to be implemented by the developer. EnerJ is also limited in that it is best suited for applications where both approximate results are acceptable and where precise computation is a source of significant power-consumption. In many applications, precise results are a requirement and the trade-off between precision and energy-efficiency is not appropriate. While many CRM applications may be able to take advantage of approximate computation in certain cases, this approach does not address the impact of using other power-hungry components, such as the cellular radio or display. EnerJ is ill-suited for handling the earlier Citisense example: sensor data should be precise and approximate computing does not address the impact of network communication. In applications where either precision is a requirement or many non-CPU components are utilized, the appropriate trade-off is not of precision versus energy, but rather timeliness versus energy.

Procrastinator is a tool that automatically delays the prefetching of network resources in Windows Phone applications so as to reduce costly network data usage [35]. 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. A runtime component makes a decision as to when to fetch requested resources by considering the available network interfaces, any data plan constraints, and whether the data is needed to populate an in-view UI element.

Procrastinator has the advantage of being fully-automated: developers do not have to annotate their UI elements with quality-of-service requirements, or to even understand the potential benefits of delaying prefetch requests, to build a more energy-efficient application. Procrastinator focuses only on networking operations, and only those related to the user interface. However, this automated approach lacks flexibility, as there is no

mechanism for specifying different constraints for different requests in an application. The focus on managing the prefetching of UI elements also limits the applicability of the approach to CRM applications, which typically include components that run in the background. For example, Procrastinator would not be applicable to the Citisense application as network accesses are not dependent on usage patterns or the position of UI elements, but rather are periodic and occur in the background. While this approach provides a quick and simple way of introducing savings to an application, its lack of developer-facing controls limit its applicability.

1.3 Annotated Programming for Energy-Efficiency

While the aforementioned systems have provided new mechanisms for introducing energy-saving optimizations into applications, they do not completely address the challenges faced by developers of CRM applications. Specifically, these systems suffer from a lack of generality and either require prohibitively burdensome refactoring (as is the case with EnerJ and Energy Types) or provide a one-size-fits-all approach that lacks sufficient control over policy behavior (as is the case with Procrastinator). An ideal solution would provide fine-grained control over power-management policy behavior, while minimizing the amount of refactoring and major structural changes required to enable those policies. The purpose of this dissertation is to demonstrate that:

A high-level annotation language can be used to effectively and safely describe and implement the diverse set of energy-management policies typically found in mobile applications. These annotation-based policies can be efficiently evaluated at runtime to reduce the power-consumption of an application while improving the maintainability of a code base through an improved separation of concerns.

Annotated Programming for Energy-efficiency (APE) is a small declarative annotation language with a lightweight middleware runtime for Android. APE enables the

```
Thread uploadThread = new Thread(new Runnable() {
  while(true) {
    @If("Battery.Level > 70%")
        @Wait(UpTo=1200, For="Network.Active")
    @Else() @Wait(UpTo=3600, For="Network.Active")
    attemptUpload();
  }
});
uploadThread.start();
```

Figure 1.5. Using APE to implement the same power-management policy found in Figure 1.4.

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. A typical policy delays execution of a segment that uses a power-hungry resource (e.g., networking) until another application or thread turns the resource on, thus allowing the segment to "piggyback" on the resource's use with little additional power consumption. 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. APE is not a replacement for existing power-saving techniques, but rather facilitates the design and implementation of such techniques in real-world applications. Figure 1.5 presents the APE-based implementation of the same power-management policy found earlier in Figure 1.4[1]

While both Energy Types and EnerJ are useful for building efficient 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

[1]Annotation syntax has evolved over the course of this project but, for the sake of clarity, has been made consistent across all chapters of this dissertation. The original syntax can be found in the first paper on APE [30].

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. While APE requires the developer to be kept in-the-loop, unlike Procrastinator, it provides fine-grained control over the behavior of the application at runtime and allows the specification of delay-constraints for individual paths of execution or objects. Each of these systems, however, would play a role in the design of APE. For example, the use of annotations in EnerJ to specify constraints for different pieces of data would influence the design of APE constructs used for tracking object-level budgets, which will be discussed in further detail later in this dissertation. Additionally, the automated nature of Procrastinator would motivate the design of the Policy Generation Engine, also presented later in this dissertation.

While annotation-based approaches have previously been utilized in code generation [4, 10, 15], verification [26], and driving performance optimizations [23, 34], APE targets the implementation of system-level power-management policies. The design of APE was inspired by OpenMP [22], 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. Much like how OpenMP allows a developer to reason about parallelism in their application at a high level, the goal of APE is to allow developers to reason about power-management in their application without being bogged down with low-level implementation details.

The APE annotation language and runtime are introduced in Chapter 2. The APE language constructs and the design of the middleware runtime are presented in Sections 2.1.2 and 2.2, respectively. An abstract model of the APE approach, based on *timed*

automata, is presented in Section 2.1.1. With this model, a wide variety of existing power management techniques can be described, demonstrating both the scope and simplicity of the approach. This model guided the design of both the APE annotation language and the middleware runtime. Section 2.3 provides a first evaluation of the APE approach. The expressiveness of the language was evaluated through a series of policy examples and a case study of introducing power management into the CitiSense CRM application, both with and without the use of APE. For the middleware runtime, it is shown 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.

Path-based Reasoning in APE. Although APE dramatically simplifies power-management code while achieving significant power savings, a challenge to its effective use is that the developer must 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 an UI operation to a delay-annotated operation elsewhere in the program, which may cause the UI thread to be blocked at runtime. Mentally reasoning about such execution paths in a large, object-oriented, multi-threaded application is taxing, at best. A secondary challenge is that, although APE bakes the resource monitoring infrastructure directly in its run-time, the developer still must make high-level power-management decisions that require deep expertise just to introduce an effective APE annotation, such as which system operations use or activate which hardware components, and what conditions to delay on and for how long.

Chapter 3 presents extensions to APE that address these challenges. Specifically, Section 3.1 presents a new annotation for demarcating *delay-sensitive* and *delay-intolerant* operations in an application. A static program analysis and runtime, presented

in Section 3.2, perform the complex tasks of inserting appropriate delays and regulating the delays at runtime to ensure delay-sensitive code is not adversely impacted by any APE-based power-management policy. To address the challenge of composing a power-saving policy in the first place, Section 3.3 presents the *Policy Generation Engine*, 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.

Object-level Delay Budget Tracking. As power-management policies impact the runtime behavior of an application, they may have unintended consequences on user experience. In fact, reasoning about the trade off between power-consumption and the timeliness of operations is one of the primary challenges of building an effective power-management policy. For example, delaying the download of stories in a news reader application may improve device battery-life, but it may also lead to a frustrated user if downloads are delayed too long. Additionally, different pieces of data in an application may have very different requirements. In a health monitoring application, for example, it may be acceptable to delay the processing of accelerometer data, but it is never acceptable to delay the processing of sensor data from a user-worn heart monitor. An object generated during the execution of a program may be expected to be "consumed" in a timely fashion, while the data it contains is still considered *fresh* and useful. In other words, it is important to be able to (1) vary the behavior of a power-management policy at runtime so that it respects the timeliness requirements of the objects it impacts and (2) provide a guarantee that certain, critical pieces of data are never subject to any power-management related delays.

```
void displayCameraImage(Bitmap capturedImage) {
  @DelayBudget(0, capturedImage)
  ...
  processImage(capturedImage);
  drawImageToUI(capturedImage);
}

void processImage(Bitmap rawImage) {
  ...
  @Wait(UpTo=3600, For="Battery.Charging AND Display.Off")
  rawImage.compress(PNG, 75, outputStream);
  ...
}
```

Figure 1.6. Using Tempus to specify a policy that delays the processing of images *except* when the image is going to be used to update the user interface.

Chapter 4 presents Tempus, a new annotation language and runtime that builds upon the techniques developed in APE and that supports the development of *object aware* power-management policies. New annotation constructs, presented in Section 4.1, allow a developer to specify a *delay budget* for an object, bounding any delays introduced by Tempus between the creation of the object and the sites of where the object is referenced throughout the program. While this object-level approach is presented as an alternative to the path-oriented approach to composing constrained power-management policies in APE, Tempus supports all of the features and constructs found in the earlier system. The implementation of the Tempus system, including how delay budget constraints are tracked at runtime and the static program analysis used to limit the number of budgets considered at each power-management policy, are presented in Section 4.2. Figure 1.6 provides an example of using Tempus to specify a power-management policy that delays the processing of images *except* when the image being processed would be displayed in the UI.

Finally, the dissertation is concluded in Chapter 5 with a summary of contributions and discussion of potential future work in the area.

Chapter 2

Annotated Programming for Energy-Efficiency

2.1 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 manage-

ment 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.

2.1.1 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 [29] 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:

$$TA = (\Sigma, S, s_0, S_F, C, E) \tag{2.1}$$

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 [17], 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 2.1 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 capture conditions that depend both on applications events and time constraints.

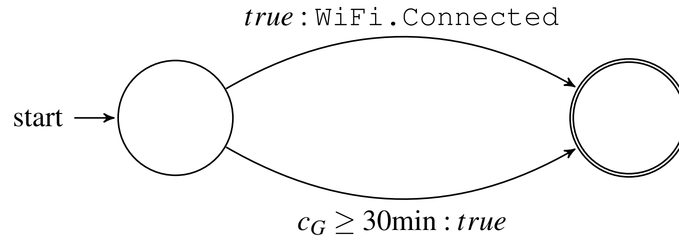

Figure 2.1. Defer uploads, for up to 30 minutes, until the Wi-Fi radio has connected to a network.

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 2.2 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`.

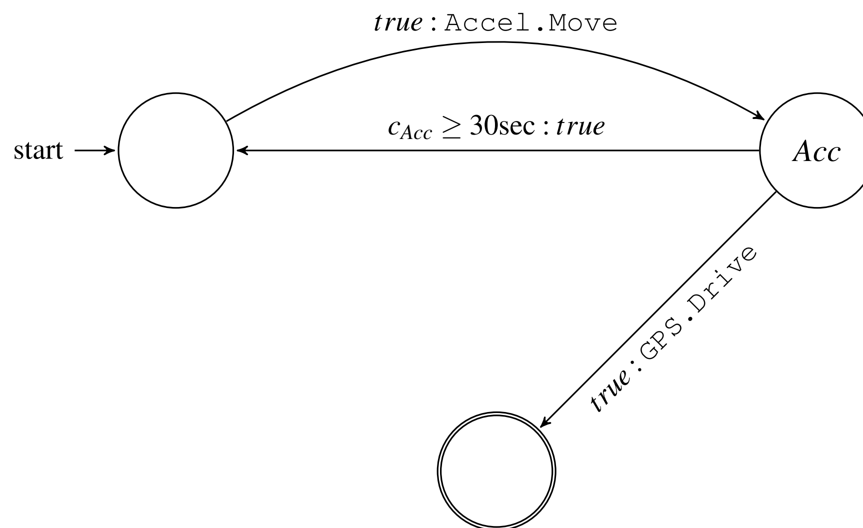

Figure 2.2. Defer sensor sampling until movement is first detected by the accelerometer and driving is confirmed using the GPS.

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:

$$P = (\{(\sigma_1,t_1),(\sigma_2,t_2)...(\sigma_n,t_n)\},\ t_{MaxDelay}) \tag{2.2}$$

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 [25]. 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 [40]. 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 2.2 is able to express a diverse set of real-world policies, this model is not as expressive as the full model of Equation 2.1. For example, consider an extension of the driving detection policy in example 2 (See Figure 2.3). 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

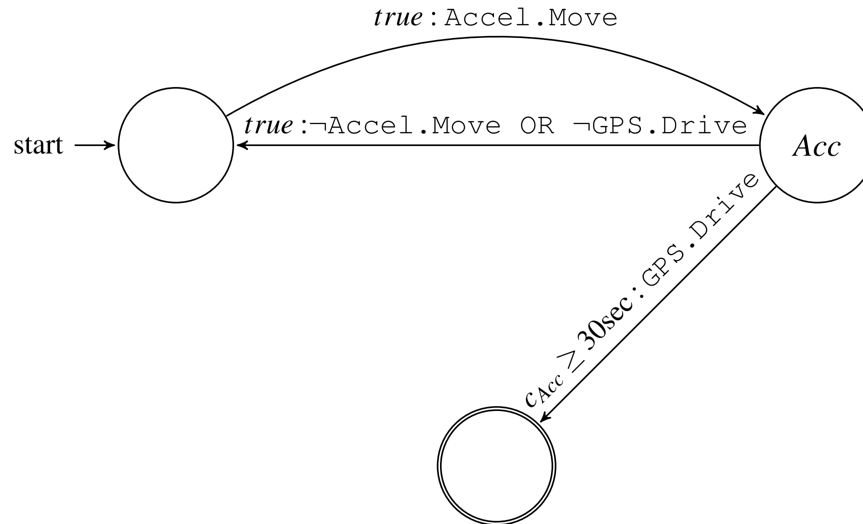

Figure 2.3. Defer sensor sampling until the user is driving. Driving is detected by first verifying movement using accelerometer for 30 seconds and then confirmed using GPS.

encountered. APE may be further extended in the future to provide a more complex syntax for expressing power-management policies using general timed automata.

2.1.2 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 2.2).

`@Wait.` The *Wait* annotation is the direct realization of the model syntax and semantics specified above in Equation 2.2. 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) {
  @Wait(UpTo=1800, For="(WiFi.Connected, inf)")
  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) {
 @Wait(UpTo=1800, For="WiFi.Connected")
 uploadSensorData();
}
```

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

```
void startDrivingLogging() {
 @Wait(UpTo=inf, For="({accMove()},inf),({gpsDrive()},30)")
 beginSensorSampling();
}
```

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

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

```
while(true) {
 @Wait(UpTo=1800, For="WiFi.Connected",
   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.

`@If`, `@ElseIf`, and `@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) {
 @If("Battery.Level > 70%")
```

```
    @Wait(UpTo=1800, For="WiFi.Connected")
  @Else()
    @Wait(UpTo=7200, For="WiFi.Connected")
  uploadSensorData();
}
```

The *If*, *ElseIf*, and *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 Wait expressions, which block until they evaluate true, any expressions provided to an If or 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 Wait, If, and 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 Wait 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

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

timed automata semantics and the evaluation of state expressions are discussed in the next section.

DefineTerm. The 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:

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

and used to construct APE annotations, such as

```
@Wait(UpTo=3600, For="{requestsPending()} AND MyTerm")
```

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.

DefinePolicy. In addition to defining new terms, developers may define reusable, high-level policies with the use of the DefinePolicy annotation. The difference between a term, defined using 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:

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

and used to contruct APE annotations, such as

```
@Wait(UpTo=3600, For="MyPolicy,({dataReady()},30)")
```

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

```
@Wait(UpTo=3600, For="MyPolicy(inf,60),({dataReady()},30)")
```

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 2.2, 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.

2.2 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 2.3.2).

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 2.1.1, 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 2.3.2.

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`. 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 2.4 provides an example of a simple boolean expression tree in APE. The arrows indicate the flow of information during the evaluation of the boolean ex-

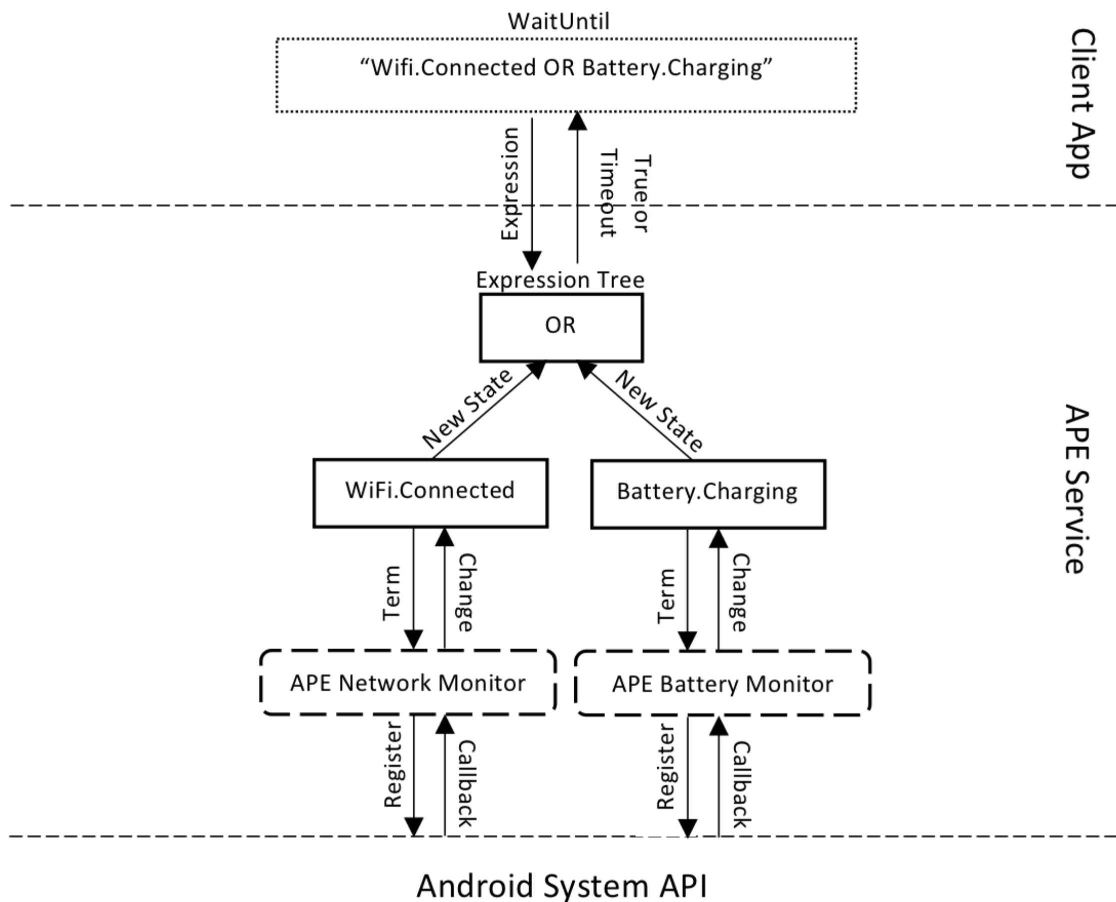

Figure 2.4. The boolean expression tree representation of a particular Wait request. All leaf nodes in the tree represent primitives in the expression, while all non-leaf nodes represent operators.

pression 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.

2.3 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

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

Figure 2.5. 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.

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.

2.3.1 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 [31]. 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 transmission 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 2.6). The equivalent timed automaton for this policy is presented in Figure 2.7. 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 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

```
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 2.6. An improved implementation of uploading in CitiSense that is both battery-life and cellular radio state aware.

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

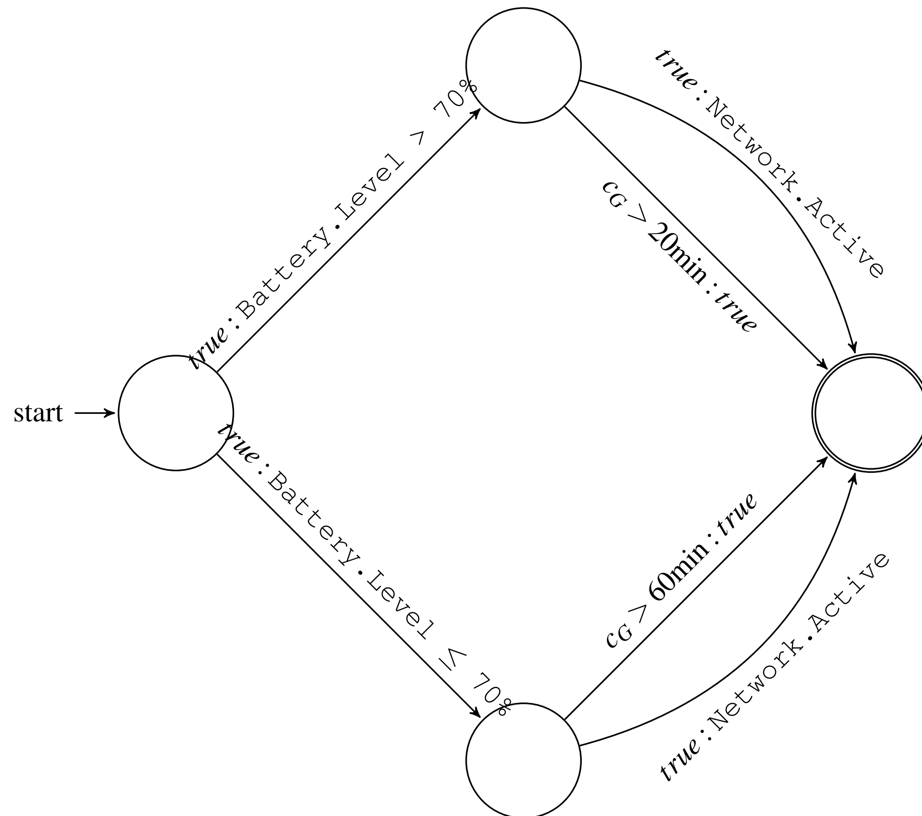

Figure 2.7. The equivalent timed automaton for the policy implemented in Figure 2.6.

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) {
    @If("Battery.Level > 70%")
        @Wait(UpTo=1200, For="Network.Active")
    @Else() @Wait(UpTo=3600, For="Network.Active")
    attemptUpload();
  }
});
uploadThread.start();
```

The developer-implemented `PhoneStateListener`, event handling, and thread concurrency management are now handled by the APE middleware. In this compact, declar-

ative 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) {
    @If("Battery.Level > 70%")
      @Wait(UpTo=1200, For="WiFi.Connected OR
        Network.Active AND (Cell.3G OR Cell.4G)")
    @ElseIf("Battery.Level > 30%")
      @Wait(UpTo=2400, For="WiFi.Connected OR
        Network.Active AND (Cell.3G OR Cell.4G)")
    @Else()
      @Wait(UpTo=3600, For="WiFi.Connected OR
        Network.Active AND (Cell.3G OR Cell.4G)")
    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.

2.3.2 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 [12]. 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.

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:

```
@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. As observed in Figure 2.8, 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 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

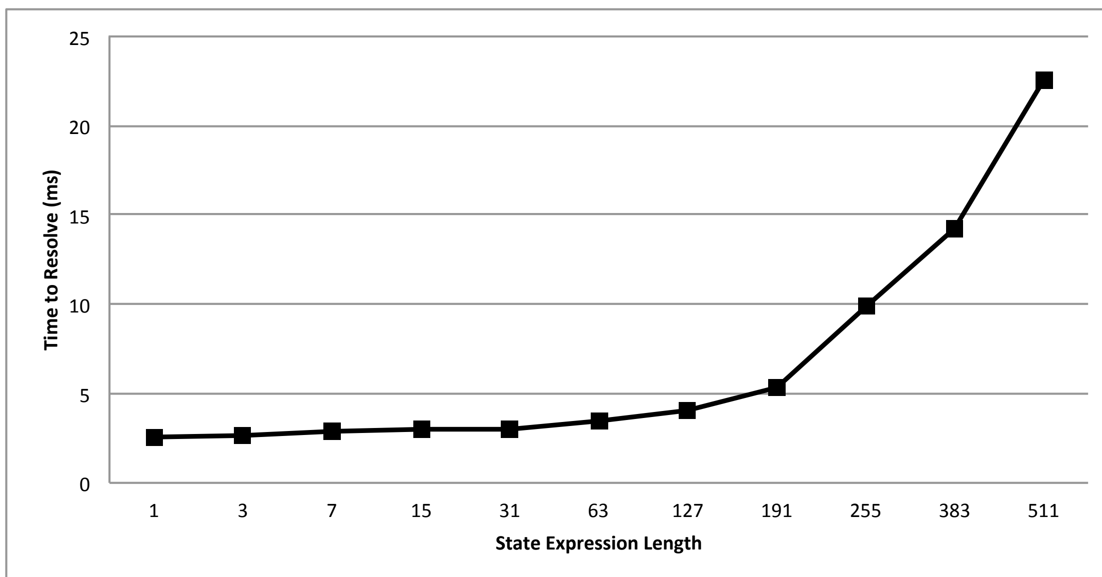

Figure 2.8. Time to register expressions of various lengths using `registerPolicy`. As the size of the expression grows, the size of the message passed over IPC begins to impact latency.

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.

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

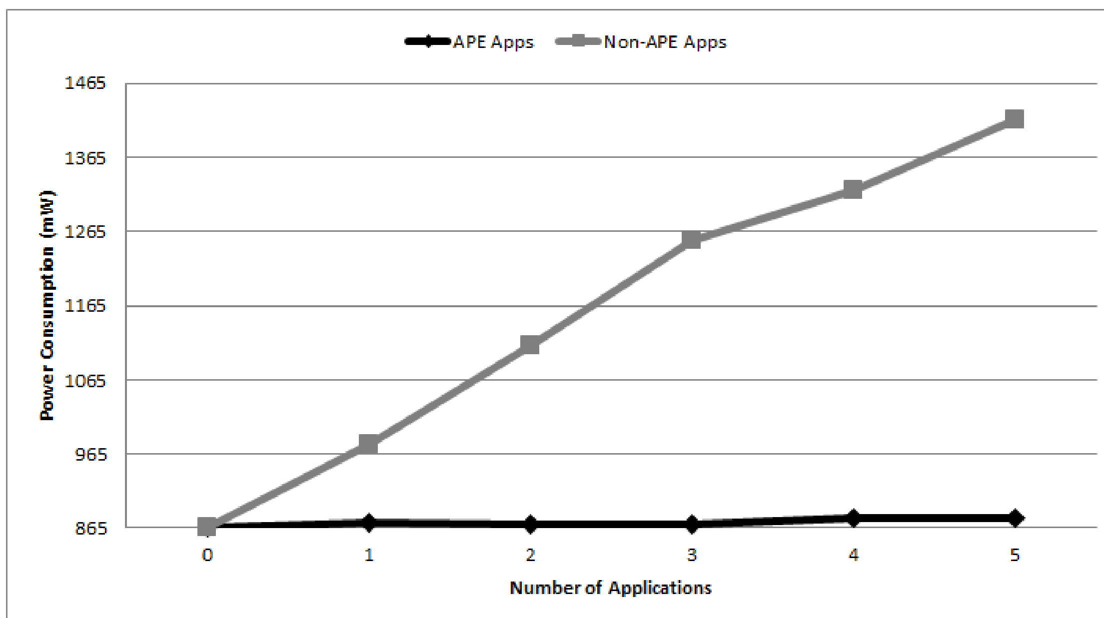

Figure 2.9. Increase in system power-consumption due to the introduction of additional applications that periodically make use of network resources. APE enhanced applications effectively recognize opportunities to transmit data efficiently, only marginally increasing power consumption.

not overlap, presented in Figure 2.9, 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 Wait annotation to wait up to 120 seconds for the cellular radio to be woken before attempting transmission. As observed in Figure 2.9, 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 2.10, 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

Figure 2.10. Power consumption (mW) traces from a smartphone device running a variety of naive (top) and APE enhanced (bottom) CRM applications.

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.

2.4 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.

2.5 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).

This chapter, in part, is a reprint of the material as it appears in Proceedings of the 36th International Conference on Software Engineering. Nikzad, Nima; Chipara, Octav; Griswold, William G. 2014. The dissertation author was the primary investigator and author of this material.

Chapter 3

Satisfying Delay Constraints

3.1 Annotation Semantics

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

3.1.1 `Wait`

Consider an execution of an operation O that is annotated with `@Wait`(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 `@Wait` annotation imposes the constraint that

$$\Delta(O,\tau) \leq D_O \tag{3.1}$$

holds on all threads τ that execute O.

3.1.2 `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).

`METHOD` alternative Consider an operation O annotated with `@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:

$$\sum_{o \in children(O) \cup \{O\}} \Delta(o, \tau) \leq R_O \tag{3.2}$$

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

`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:

$$\sum_{o \in predecessors(O) \cup \{O\}} \Delta(o, \tau) \leq R_O \tag{3.3}$$

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 `@Undelayble()` as equivalent to `@DelayableUpTo(0)`.

3.1.3 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 3.1, the developer annotates the `openConnection` invocation with an `@Wait` (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 five minutes when the screen is off (see line 15).

The developer may customize the user experience by introducing `@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 `@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 `@Wait`.

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 `@DelayableUpTo(1 min)` (see line 5). The annotation requires that the aggregated delay of all operations from the start of the thread until the

```
1  class ProcessDataFromServer implements Runnable {
2   public void run() {
3    Data data = NetworkUtils.downloadData();
4    if (data.isStockTicker()) {
5     @DelayableUpTo(1min)
6     // @DelayableUpTo(1min, METHOD)
7     processData(data);
8    } else {
9     processData(data);
10   }
11  }
12
13  void processData(Data data) {
14   ...
15   @Wait(UpTo=5min, For="Display.ON")
16      updateDisplay(data);
17      ...
18  }
19 }
20
21 class NetworkUtils {
22  public static Data downloadData() {
23   URL url = new URL(SERVER_ADDR);
24   @Wait(UpTo=20min, For="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 }
```

Figure 3.1. A thread attempts to download data to displayed within the application. The `@DelayableUpTo` annotation ensures that the downloading and displaying of the data is not delayed by APE by more than one minute.

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 `@Wait` may delay processing up the the full five minutes.

3.2 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 `@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 `@Wait` annotation site and may not spend more than their smallest assigned allowance. When a thread has zero remaining delay allowance, it simply skips any `@Wait` annotations. Threads without any form of `@DelayableUpTo` constraint have infinite delay allowance, meaning that the time spent waiting at a `@Wait` annotation is bounded only by the `MaxDelay` parameter of the policy. Allowances are updated by the APE runtime after each `@Wait` annotation to reflect the time actually spent waiting at the annotation site.

3.2.1 Algorithms for Static Analysis and Monitoring

The static program analysis employed by APE is built upon the Soot Java Optimization Framework [38]. 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 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 [18] is an area of future work.

The analysis considers each `@DelayableUpTo` operation in the CFG and generates allowances as follows. Consider the annotation `@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 `@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 3.2.

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

1: let G be the control flow graph of the application
2: let X be set of operations annotated with `@DelayableUpTo`
3: for each operation $O \in X$:
4: let a be the annotation `@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 $e \in E$: 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 $f \in F$: clear allowance R_O at f

Figure 3.2. Algorithm for instrumenting application code with delay allowances.

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 3.3. 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 `@Wait` annotation may delay it without violating the delay constraints is:

$$R_t = \min_{O} \Delta[\tau, O]$$

Accordingly, a `@Wait(`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.

1: **add-allowance**(τ, O, R_O):
2: $\Delta[\tau, O] = R_O$

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

5: **wait-until-entry**(τ, E, D_O):
6: let $R_\tau = \min_O \Delta[\tau, O]$
7: wait up to $\min(R_\tau, D_O)$ for E to be satisfied
8: let T be the time spent waiting
9: for each $(\tau, O) \in \Delta$: $\Delta[\tau, O] = \Delta[\tau, O] - T$

Figure 3.3. Instrumentation for assigning, clearing, and using delay allowances.

3.2.2 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” [3]. 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 `Wait` 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.

3.3 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.

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.

In NetworkUtils.java:

```
89  public void sendToServer(byte[] data) {
90      URL url = new URL(SERVER_ADDR);
91
92      HttpURLConnection conn = (HttpURLConnection) url.openConnection();
93      InputStream is = conn.getInputStream();
```

@APE_WaitUntil(" `WiFi.Connected OR Network.Active AND (Cell.4G OR Cell.3G)` ", MaxDelay= `1200`)

Explanation: Wait up to 20.00 minutes (1200 seconds) for Wi-Fi to be connected OR network activity to be detected AND (cellular network to be 4G OR cellular network to be 3G)

Accept and insert this policy

Reject this policy

Most Relevant Resource: Network

Network.Active	Data is currently being transmitted by the device. Waiting for this state is generally a good idea before transmitting data as it avoids uncessary waking of the radio and can reduce power-consumption.
WiFi.Available	The device has a Wi-Fi radio. while the device may have a radio, it might not be currently connected to a Wi-Fi network.
WiFi.Connected	The device is currently connected to a Wi-Fi network.
Cell.Available	The device has a cellular radio. While the device may have a radio, it might not be currently connected to a cellular network.
Cell.Connected	The device is currently connected to a cellular network.
Cell.GSM	The cellular radio is currently connected to a GSM network.
Cell.EDGE	The cellular radio is currently connected to an EDGE network.
Cell.3G	The cellular radio is currently connected to a 3G network.
Cell.4G	The cellular radio is currently connected to a 4G network.

Figure 3.4. 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.

Figure 3.4 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 [32]. 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.

3.4 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.

3.4.1 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 `@Wait` 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 [5],
- `AudioSense`: a CRM application for hearing aid performance evaluation [24],
- `K-9`: an e-mail client [11],
- `NPR News`: an application for reading and listening to news stories [13],
- `ohmage`: a participatory sensing platform [14], and
- `WeatherLib`: a library for weather applications [16].

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 `@Wait` 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 `@Wait` 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 3.1.

`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

Table 3.1. 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%)

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 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.

3.4.2 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 [12]. 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 `@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

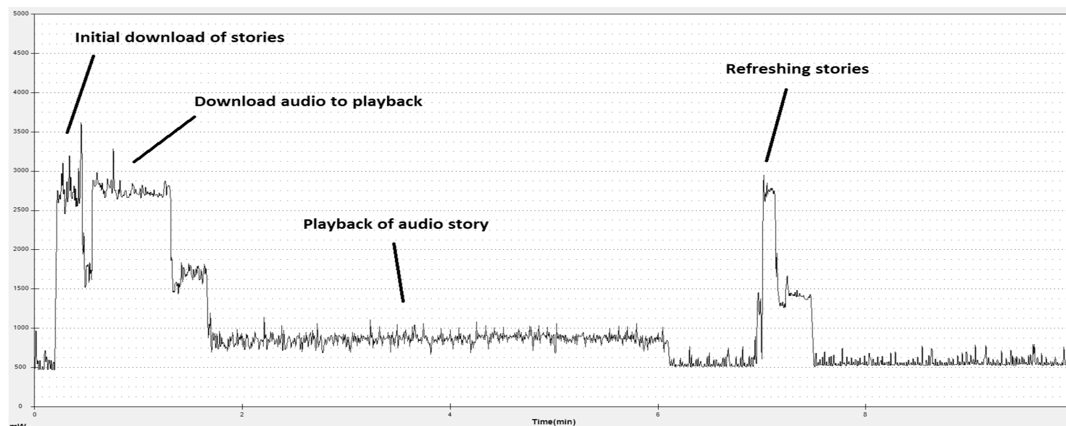

Figure 3.5. A trace of power consumption on a smartphone device while using the `NPR News` application to listen to an audio story.

responsible for fetching news stories at start up. In the AndStatus application, an `@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 3.6. 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%

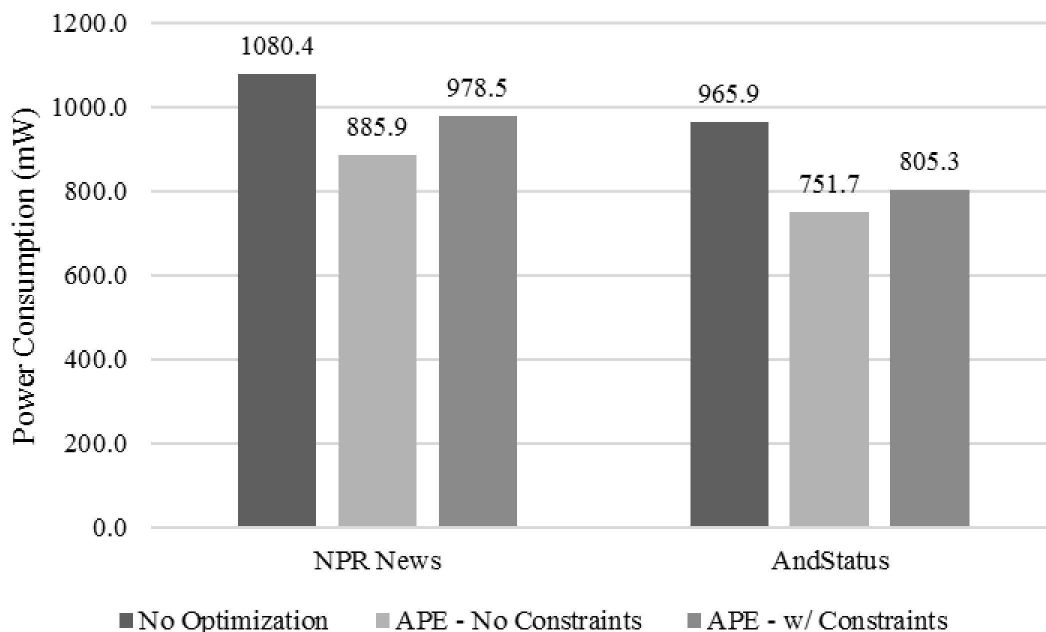

Figure 3.6. Power consumption in various versions of the `NPR News` and `AndStatus` applications.

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

3.4.3 Runtime Overhead

In our previous APE work we found that each invocation of a `@Wait` 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 `@Wait` 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 $2\mu s \times C + 1710\mu s \times L$, 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.

3.5 Conclusion

In this chapter we presented a new paradigm for introducing power-saving delays into an application. We presented the `@DelayableUpTo` and `@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.

3.6 Acknowledgements

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).

This chapter, in part is currently being prepared for submission for publication of the material. Nikzad, Nima; Chipara, Octav; Griswold, William G. The dissertation author was the primary investigator and author of this material.

Chapter 4

Ensuring Timely Delivery of Delay-Sensitive Objects

Mobile applications may save significant power by changing the timing of delay-tolerant operations. However, operations differ in their degree of tolerance to delays, so the developer must carefully balance energy savings against user experience. Writing policies that trade-off timeliness and energy savings is difficult since such policies typically crosscut components and involve multiple threads. More importantly, it is difficult to reason about the impact of delaying threads in a multi-thread application. Our goal is to develop a programming model that simplifies the writing of such policies and reasoning about their impact on user experience. In the following, we will highlight some of the challenges of writing such policies by hand and derive requirements for the programming model. The examples in this section are based on CitiSense [31], an environmental air pollution monitoring application that utilizes a body-worn sensing device to accurately measure a user's exposure to a variety of pollutants. Screenshots of the application are included in Figure 4.1.

Saving Power: A common pattern in mobile applications is to perform long-running operations in the background. For example, in CitiSense, a thread is dedicated to uploading batched sensor readings to a remote server so that they may be used to generate

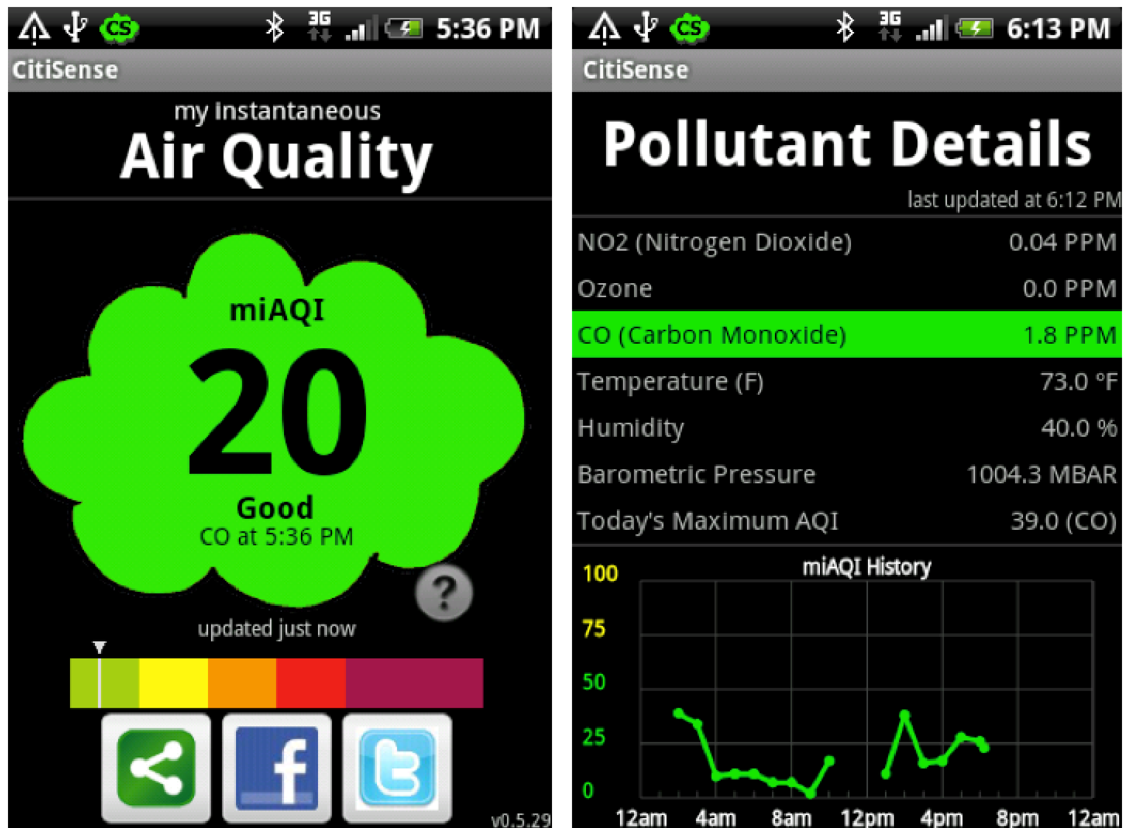

Figure 4.1. Screenshots from the CitiSense environmental air pollution monitoring application: the most recent Air Quality Index score (left) and detailed pollutant report (right).

pollution models for the region and notify users of pollution hot spots. Since this operation is *delay-tolerant*, the upload of data may be deferred until another application turns on the network interface. Implementing this policy requires mixing event-based programming that tracks hardware states and multi-threaded code that blocks (and later resumes) the network thread until the device enters the desired state. Our previous system, APE, address the mechanism of implementing such power management policies. However, APE does not address the problem of managing the impact of power management on user experience, which is the focus of this chapter.

Ensuring Timeliness: A naive approach to this problem is to add a timeout to a power policy to limit its impact on timeliness. Unfortunately, a one-size-fits-all approach

is insufficient when a component operates over different types of data. For example, in CitiSense, not all sensor readings are of equal importance. While most pollution measurements may conform to the expected and healthy levels, a user may occasionally discover a location with unusually high and harmful pollution levels. Uploading these high readings in a timely fashion is particularly important, as the server would like to warn other nearby users of the discovered pollution hot spot as soon as possible. Therefore, the developer must be able to write power policies that provide *differentiated* timing behavior based on the type of data the system processes.

A good programming model must ensure that even as the complexity of the application increases it remains easy to reason about its properties. To this end, we are interested in a model that allows the user to *isolate* the operation of policies and supports *predictable composition* semantics when an application includes multiple policies. For example, let us return to the example of sensor readings. The same sensor readings must be both displayed on the user interface and uploaded to the server. When the user is viewing the sensor readings in the application (see Figure 4.1), it is essential that the path to the user interface is not delayed. The programmer should be able to isolate the behavior of power annotations even as they may share data. Similarly, applications include multiple power management policies must have a predictable behavior. For example, a high pollution value is processed through multiple stages, each potentially including power management policies: the samples are collected, saved to flash, and uploaded to the server. The developer must be able to bound the end-to-end impact of power management policies on data processing across threads.

4.1 Annotation Semantics

A developer may use our annotations to compose power management policies that manage the trade-off between energy and timeliness. Tempus saves power by deferring

the execution of power hungry operations until the device enters a power state that minimizes the cost of that operation. Tempus ensures timeliness by guaranteeing that the processing of any object is not delayed more than a user-specified budget. The static analysis and run-time environment ensure that these properties hold even when multiple policies are executed concurrently. In this section, we formalize the semantics of Tempus annotations. Static analysis is used to minimize the annotation effort of developers. However, as static analysis is inherently conservative, annotations are designed to allow explicit and fine-grained control over the behavior of Tempus. The details of how the annotations are used in code generation, static program analysis, and runtime analysis are presented in following section.

`@DelayBudget/@ClearBudget` Tempus's core abstraction is that of a *delay budget* that is associated with an object. Consider a reference r to an object o in the application code. The annotation `@DelayBudget`(B_o, r) assigns a budget of B_o to r. If a budget was already assigned to o, its value is updated to equal B_o. The tracking of an object's budget may be stopped using annotation `@ClearBudget`(r).

Java is a garbage-collected language that provides no control over when stale objects are freed. As a consequence, Tempus may conservatively decide to stop waiting for a power-efficient state due to the budget of a stale object.[1] Some objects have a two-phase lifetime, an initial time-critical phase, and a later non-time-critical phase (or perhaps vice versa). The `@ClearBudget` annotation provides the programmer explicit control over removing such an object from consideration in determining delays. In our experience, `@ClearBudget` annotations are seldom necessary as static analysis usually determines the liveness of objects accurately.

[1] We note that even in this case Tempus does not violate the timeliness guarantees albeit the power savings are reduced.

`@Namespace/@ClearNamespace` Namespaces allow programmers to group multiple objects under a single namespace. Accordingly, an object o referenced by reference r may be added to namespace l using annotation `@Namespace`(r,l). We allow an object to belong to multiple namespaces. The object may be removed from namespace using annotation `@ClearNamespace`(r,l).

Namespaces play a crucial role in Tempus. Because the static analysis is conservative, a programmer might want to override the analysis and provide its own determination of what objects are live. Also, namespaces may also be used to isolate the behavior of policies by defining them over non-overlapping namespaces.

`@Wait` Tempus defers the execution of an operation until the device enters a power-saving state without violating the timing constraints. Consider the execution of an operation P annotated with `@Wait(UpTo=`D_P`, For=`E`)` on a thread τ. The expression E defines the power-saving state in which P has a low energy-cost. Tempus borrows from APE the mechanism used for specifying the power state E. In its simplest form, E is a boolean expression composed of one or multiple built-in terms. A term refers to the state of a variety of hardware components such as the cellular radio, display, and battery. For example, the expression

```
Network.Active OR WiFi.Connected
```

specifies that the device should wait until either the network becomes active or WiFi is connected. More complicated expressions may be defined over time-based sequences of states as described in [30].

Thread τ is blocked until the device enters a state in which expression E holds or the annotation's timeout D_P is reached. We call the time the thread τ is blocked while operation P is deferred as the *operation's delay* ($\Delta(P,\tau)$). The behavior of the `@Wait`

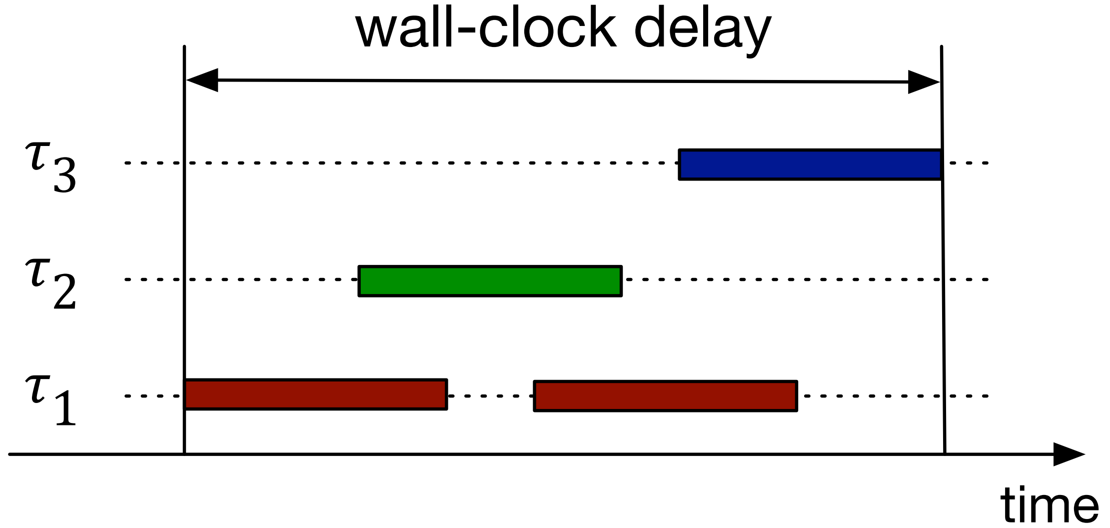

Figure 4.2. The execution of three concurrent `@Wait` annotations on threads τ_1, τ_2, and τ_3 that include the same object o in their scope. The filled boxes indicate when a thread is running. The operation delay observed by o is the *system delay* i.e., the duration of time while at least one thread is running.

annotation depends on the budgets of the objects that are in *scope*. By default, the *scope* of a `@Wait` is determined by the static analysis. Since `@Wait` blocks τ, the processing of instructions following `@Wait` is delayed. Accordingly, we defined the *default scope* to include any object o that has a budget *and* may be reference on any path starting at the annotation. We allow the programmer to override the scope of `@Wait` by adding an additional `Scope` argument. The `Scope` argument may include both object references and namespaces separated by the "—" operator. The scope is the union of the reference objects and the objects pertaining to the specified namespaces.

The static analysis and run-time environment ensure that the processing delay ($\Delta(P,\tau)$) of any `@Wait` does not exceed the minimum budget of the objects in scope and the annotation timeout D_P.

$$\Delta(P,\tau) < \min(D_P, \min_{o \in \text{scope}} B_o) \tag{4.1}$$

where, B_o is the budget of object o.

After the execution of a `@Wait`, the budget of all objects in scope is updated. In the case when a single `@Wait` annotation is executed at a time, the budget of an object is decremented by its operation delay $\Delta(P, \tau)$. However, the case when an object o is in the scope of multiple `@Wait` annotations that are executed concurrently requires more careful handling. In this case, the operations that involve o are delayed according to the *system delay* that measures the time at least one of the threads is executing. As an example, consider Figure 4.2 in which three annotations that involve o are executed on three threads τ_1, τ_2, and τ_3. In this case, the time that operations on o are delayed is the time while at least one of the three threads is executed. This time is labeled as the system delay in the figure. These semantics are consistent with the intuition that time flows in parallel on independent threads.

A key design decision was to opt against the use of real-time deadlines in favor of delay budgets. Relative deadlines are an effective abstraction for specifying the time when a task should complete in real-time systems. However, focusing on deadlines would misplace the focus on trying to predict how long operations take such that deadlines are satisfied. Instead, we would like the developer to focus on specifying the additional delay that the processing of objects may tolerate to save power. This design philosophy is reflected in the decision to use budgets that are consumed only when a thread is blocked by a `@Wait` annotation.

Example: Consider an application that processes sensor data to detect health related events, presents results to a user, and periodically uploads reports to a remote server (see Figure 4.3). The `generateReport` method is responsible for generating upload-ready `HealthReport` objects. The `uploadHealthReport` method, which handles the upload of `HealthReport` objects, contains a `@Wait` annotation that delays transmissions up to twenty minutes while waiting for a Wi-Fi connection to be established

or for another application to wake the cellular radio. However, the developer would like to ensure that the upload of (or any other use of) `HealthReport` objects related to critical health events are *never* subject to delay by Tempus. To do so, a reference to the critical `HealthReport` is annotated with `@DelayBudget(0)` at line 7. As the `uploadHealthReport` method contains a possible reference to the health report at line 33, the `@Wait` will check to see if a delay budget was assigned to `report`. If `report` has a delay budget of zero (because it was a critical event), then the `@Wait` has no effect and transmission continues without delay.

4.2 System Design and Implementation

An Android application containing Tempus annotations is processed by a source-to-source translator, built on the Java processor package. Each Tempus annotation is translated into a Java call to the Tempus runtime service, which performs device-state monitoring, power-management policy evaluation, and object-budget tracking on behalf of client applications.

With respect to ensuring timely execution of delay-sensitive operations, the runtime ensures that the maximum delay introduced by a `@Wait` is bounded by the smallest budget of all objects that are potentially referenced after the `@Wait` annotation. However, it is not possible, at runtime, to determine what objects will be potentially referenced in the future. Thus, a static program analysis is performed by the translator to conservatively determine these objects. It walks through the call graph of the application and identifies potential references to budgeted objects. Each `@Wait` annotation is then compiled to include an argument, the "scope", about which object budgets to consider.

The runtime tracking and the static analysis required to achieve the semantics of Tempus are non-trivial. For one, the delay budgets of all objects that are "downstream" in the control flow from a `@Wait` need to be monitored and updated. Two, the determination

```
1   HealthReport generateReport(Data data) {
2        ...
3        if(healthStatus == CRITICAL_STATUS) {
4         // Notify user of critical condition
5         ...
6         // Generate time-sensitive report for upload
7         @DelayBudget(0)
8         HealthReport criticalReport =
9          new HealthReport(System.currentTimeMillis(), data);
10        ...
11        return criticalReport;
12       } else {
13        // Non-critical event
14        ...
15        HealthReport processedReport =
16         new HealthReport(System.currentTimeMillis(), data);
17        ...
18        return processedReport;
19       }
20       ...
21  }
22
23  void uploadHealthReport(HealthReport report) {
24   URL url = new URL(SERVER_ADDR);
25   @Wait(UpTo="20min", For="WiFi.Connected
26    OR Network.Active and (Cell.4G OR Cell.3G)")
27   HttpURLConnection conn =
28    (HttpURLConnection) url.openConnection();
29     try {
30      conn.setDoOutput(true);
31      ObjectOutputStream out =
32       new ObjectOutputStream(conn.getOutputStream());
33      out.writeObject(report);
34     ...
35  }
```

Figure 4.3. The `@DelayBudget` annotation (line 7) ensures that the upload of `HealthReport` objects related to critical health events are not subject to any Tempus-introduced delays (line 25) when uploaded (line 33).

of these downstream objects should be as precise as possible while still being safe. That is, if it is impossible for an object to be accessed downstream from a particular `@Wait`, then its budget should not be monitored or updated when that `@Wait` is invoked. Third, object budgets need to be tracked correctly when multiple `@Waits` occur at the same time, due to multithreading. Finally, object tracking needs to be sufficiently efficient so that the application neither is noticeably slowed or its energy wasted. In addition, a

challenge addressed by our previous work is how an energy policy specified in a `@Wait` is efficiently evaluated and executed. We briefly touch on this subject after first addressing the issues of budget tracking.

4.2.1 Budget Tracking

We first describe the translation of Tempus annotations to Java code, along with the semantics of the operations. We then describe the static analysis required to make the budget tracking as precise as possible.

Annotation Translation. A `@DelayBudget` annotation is replaced with a call to the Tempus service that registers the relevant object and its budget with Tempus: `@DelayBudget(X)` on a reference `R` is replaced with

```
Tempus.RegisterBudget(ID, R, X)
```

where ID is a unique label assigned to each annotation site. An example of this translation is shown in Figure 4.4: the object referred to by `criticalHealthReport` is registered with the Tempus runtime under the label `0` with a budget of 600 seconds.

A formalization of `RegisterBudget`'s behavior is provided at the top of Figure 4.6. Tempus maintains two mappings for budget tracking, one from a label to its objects (λ), and another from an object to its budget (β). The label is used by the runtime as a retrieval key for all the objects ever returned from a `@DelayBudget` annotation site that are still accessible by the application. `RegisterBudget` is essentially an initializer for these maps. It associates an object with the annotation site's label and assigns the object a budget. To handle the case where an object is assigned a budget more than once, the assigned budget is the minimum of the old and new budgets.

```
@DelayBudget(600)
HealthReport criticalHealthReport = ...;
```

```
HealthReport criticalHealthReport = ...;
Tempus.RegisterBudget(0, criticalHealthReport, 600);
```

Figure 4.4. `@DelayBudget` annotations are translated into runtime calls to the Tempus service that that assign a budget to a particular object and begin tracking it. The first parameter is an automatically-generated label.

While `@DelayBudget` initializes objects for budget tracking, `@Wait` consumes and updates budget-tracking information (in addition to providing the all-important conditional delay capability). Figure 4.5 shows how the example from Figure 4.3 would be translated. In the figure, the number 5 is an index that refers to a pre-compiled version of the expression `"WiFi.Connected..."`; the 7200 is 20 minutes in seconds; and the static integer array is a list of labels provided to the call, typically the labels of object references that the static analysis found to be downstream from the `@Wait`. In this example, the list includes just the label 0. The labels provided to a `@Wait` are used to lookup the objects associated with the labels, determine the minimum budget among those objects, and then set a maximum wait time based on the minimum of 7200 and the minimum object budget. When the `@Wait` stops waiting, it will decrement the budgets of the objects by the actual wait time.

```
Tempus.Wait(5,72000,int[]{0});
```

Figure 4.5. Translation of the `@Wait` annotation. See text for explanation.

The budget-tracking behavior of `@Wait` is formalized in the second and third operations listed in Figure 4.6. These are invoked inside the `Tempus.Wait` call. At the beginning of that call, `GetSmallestBudget` takes a set of labels and returns

the smallest budget of all the objects referred to by the labels. It helps determine the maximum allowable delay. At the end of the call, after the delay has completed, `ConsumeBudget` takes a set of labels and decrements the budgets of all the objects referred to by all the labels.

```
1  RegisterBudget(l, r, b):
2    λ[l] = λ[l] ∪ {r}
3    β[r] = min(β[r], b)
4
5  GetSmallestBudget(L):
6    return b_ij | b_ij ≤ b_kl, b_ij = β[r_j], b_kl = β[r_l],
7                  r_j ∈ λ[l_i], r_l ∈ λ[l_k], l_i, l_k ∈ L
8
9  ConsumeBudget(L, t):
10   β[r_j] = β[r_j] − t | r_j ∈ λ[l_i], l_i ∈ L
11
12 AssignNamespace(l, r):
13   λ[l] = λ[l] ∪ {r}
14
15 InitNamespace(l):
16   λ[l] = ϕ
17
18 InitBudget(r):
19   β[r] = ∞
```

Figure 4.6. Formalization of the single-threaded behavior related to object budget tracking in the Tempus runtime.

An annotation `@Namespace(l)` on a reference `r` is replaced with a runtime call to the Tempus service just after `r` that assigns the specified label `l` to the object pointed to by `r`: `Tempus.AssignNamespace(r, l)`. Its formalization is shown three from the bottom in Figure 4.6. At this level, it can be seen that `AssignNamespace` is basically `RegisterAnnotation` without assigning a budget.

Two related annotations are `ClearNamespace(l)` and `ClearBudget(r)`. `ClearNamespace(l)` is formalized as reinitializing the label to the empty set. The `ClearBudget(r)` annotation is formalized as resetting the budget of the object to

infinity. See Figure 4.6, operations `InitNamespace` and `InitBudget`. We discuss latter's actual implementation shortly.

Runtime Implementation Details. To ensure that tracking does not interfere with the garbage collection of objects that are no longer referenced in the application, the objects in both mappings are maintained with what Java calls *weak references*. This means that when a tracked object is garbage collected, its entry is dropped from the mapping. Specifically, the budget mapping β is implemented with a `WeakHashMap`, which has weak references to keys, and λ is implemented with a `SparseArray` of lists of `WeakReference`.

A related issue is that infrequent garbage collection may result in objects staying in our mappings after the objects are no longer reachable by the application. These objects will still be included in budget tracking, perhaps causing `GetSmallestBudget` to calculate an artificially low value. For applications that this is an issue, there is an extra optional parameter to the `@Wait` annotation, `AggressiveGC=True`, that forces the `@Wait` to call the garbage collector, thus mitigating this issue at the cost of an additional GC. We discuss the impact of this option in Section 4.4.2.

For the `ClearBudget(r)` annotation, formalized in Figure 4.6 as `InitBudget` setting the object's budget to infinity, what really happens is that the object `r` is deleted as a key from the β mapping. Note that this leaves `r` in any namespaces that it might reside. In practice, when such a "dangling" reference is discovered by Tempus (i.e., a labelled object is not in the budget table), it is removed from the mapping. Such cleanups also occur for garbage-collected objects, which leave empty `WeakReferences` behind.

The formalization provided in Figure 4.6 and discussed above describes the single-threaded sequential semantics of Tempus, and the actual implementations are

generally analogous to what is in the figure, even in the concurrent case.[2] The exceptions are `GetSmallestBudget` and `ConsumeBudget`, whose implementations are more complicated than suggested in order to get the desired semantics under concurrency. The critical case is handling when two or more `@Wait` invocations overlap, as can occur when more than one thread is waiting for a resource to wake. To discuss this, a little terminology helps. An object that is deemed impacted by a `@Wait`, whether by the static analysis or directly by the programmer passing in a user-defined namespace to the `@Wait`, is referred to as *waiting* for the `@Wait`. Now consider the case of two or more `@Wait` invocations overlapping in time, and sharing one or more waiting objects. Once the first `@Wait` starts, the budgets of all its waiting objects are continually dropping. When the next `@Wait` begins, it will need to use the budget of objects waiting on the first `@Wait` (to compute `GetSmallestBudget`), but their budgets have not been updated yet, since budgets are updated at the end of a `@Wait` by `ConsumeBudget`.

Thus, to handle the overlapping case correctly, the initialization stage of the first `@Wait` timestamps all of its waiting objects with the `@Wait`'s start time (as reported by `System.nanoTime()`). Also, for each object, the identity of the `@Wait` is inserted into a set of in-progress `@Wait` invocations. Now, when another `@Wait` begins (before the first `@Wait` completes, i.e., its in-progress set is non-empty), the budget of any object waiting on the first `@Wait` current budget is calculated from its stored budget minus the time elapsed since the recorded timestamp. This ensures that the right budget is provided for use in `GetSmallestBudget`. This `@Wait`'s identity is also inserted into the object's set of in-progress `@Waits`, but the timestamps are not set because in-progress `@Wait` set was not empty. When a `@Wait` completes (say, the first one), it removes itself from its waiting objects' `@Wait` sets. If an object's `@Wait` set becomes empty,

[2]In the following, the discussion focuses on the maintenance of object time budgets. It is assumed that concurrent access to objects is correctly managed (e.g., via locks).

it updates the budget of the object using the timestamp stored by the initial `@Wait` (and clears the timestamp. Otherwise, it leaves the budget to updated by the remaining `@Wait`(s). The result is that the net budget time charged to an object is the system clock's elapsed time, and is not double-charged when two `@Wait`s are active at the same time. This is the system-delay property described in Section 4.1.

Another exception to the sequential semantics occurs when a thread sets an object's budget to zero while another thread is at a `@Wait` with that object waiting for it. In this case, the `@Wait` is immediately terminated to satisfy its intended delay semantics. This is detected by the RegisterBudget method (which is used to set the budget), which checks if the object's timestamp is set. If so, it signals all the `@Wait`s in the object's in-progress list.

Static Analysis. It would be safe for Tempus to treat every object as though it were potentially referenced downstream from every `@Wait`. That is, it would be safe to pass every label to every `@Wait`. However, this simple approach would be detrimental. CRM applications often have many threads of execution and some objects may never be referred to on certain paths of execution. It would be overly conservative and undesirable to consider and consume the budgets of *every* object with a delay budget assigned when a `@Wait` annotation is reached. As soon as any object's budget reached zero, no `@Wait` would be able to wait again until the object becomes unreachable and is reclaimed by the garbage collector. Perhaps worse, every object has to have its budgets tracked and updated at every `@Wait`, potentially hurting performance. Thus, it is beneficial for the Tempus translator to use a static analysis of the application's source code to identify a minimal, safe approximation of the objects that can be referenced downstream from each individual `@Wait`.

The static program analysis employed by Tempus is built on the Soot Java Optimization Framework [38].[3] Specifically, the framework is used to generate a call graph for the target application and to perform context-sensitive flow-insensitive points-to analysis. A points-to analysis conservatively determines the set of possible objects referenced by each Java expression. For any two given expressions, it can be determined if they possibly reference the same object by intersecting their points-to sets. The analysis handles multithreading correctly, so that objects shared between threads are appropriately accounted for in points-to sets.

A particular object is said to be *potentially impacted*, or simply *impacted*, by a `@Wait` if a reference to that object appears on a path of execution leaving from that `@Wait`. In aggregate, an object is said to be *impacted* by Tempus as a whole if it is impacted by at least one `@Wait`. To ensure that delay-sensitive data is being processed in a timely fashion, it is necessary that the delays introduced by the use of a `@Wait` be bounded by the smallest budget across all impacted objects, but ideally not by objects that aren't impacted by the `@Wait`.

Algorithmically, this means that the static analysis needs to find the set of labels that cover all `@DelayBudget`-annotated objects that are potentially referenced after each `@Wait` annotation in the program. It is essentially a live variable analysis. Pseudocode for this analysis is provided in Figure 4.7. For each `@Wait`, the call graph of the application is searched to enumerate all paths that begin at the annotation. A points-to analysis is used to generate a points-to set for each reference that appears on each such

[3]Proper use of Soot requires that the developer specify the entry point into their application from which to begin analysis, such as the `main` method. Android applications often do not have a single entry point, but rather include many entry points that may be called by the Android framework. To allow Soot to properly analyze the application, a dummy `main` method is first constructed that 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 [18] is an area of future work.

```
1  let G be the control flow graph of the application
2  let R be the set of references annotated with @DelayBudget
3  let W be the set of references annotated with @Wait
4  let V be the set of all references to objects in the program
5  let A = {}
6
7  for w_i ∈ W:
8    let A_i = {}
9    use G to find P_i, the set of all paths starting at w_i
10   for p_j ∈ P_i:
11     let V_j be the set of all references made along p_j
12         for v_k ∈ V_j:
13           let p_k be the points-to set for v_k
14           for r_l ∈ R:
15             let l_l be the label assigned to r_l
16             let p_l be the points-to set for r_l
17             if p_l ∩ p_k ≠ ∅:
18                   add l_l to A_i
19   add (w_i,A_i) to A
20
21 return A
```

Figure 4.7. Algorithm for static program analysis that returns the set of object labels to be considered at each `@Wait` annotation in the program.

path. If any such reference has a non-empty intersection with the points-to set of any of the `@DelayBudget`-annotated references in the program, then the two references may potentially refer to the same object at runtime. In such a case, the unique label of the `@DelayBudget`-annotated reference is added to a set of labels determined to be relevant to the `@Wait`. The `@Wait` annotation is then translated so that it includes the set of relevant labels as an argument, as previous shown in Figure 4.5.

4.2.2 Policy Evaluation

When a `@Wait` annotation is reached, and after its maximum wait time has been calculated, Tempus begins monitoring the subset of components required to evaluate the annotation's boolean expression, using the methods employed with our previous APE system [30]. Device state monitoring is accomplished using a variety of APIs exposed by the Android framework. Depending on the resource being monitored, state information

is either periodically polled or delivered via call backs from the Android framework. The monitoring is invoked via a synchronous request made to the Tempus runtime. This call does not return until it either times out or the requested expression is evaluated to be true, thus inserting a delay at the site of the `@Wait` annotation.

4.3 Case Study

In this section we present a case study of introducing a power-management into the NPR News mobile application [13] and CitiSense [31]. We will demonstrate the flexibility of Tempus by constructing increasingly complex power management policies. The chosen examples illustrate how a developer can provide differentiated timeliness in power management using statically and dynamically assigned budgets. The CitiSense example illustrates the composition properties of Tempus that allow developers to reason about the timelines of applications that may include concurrently running power management policies.

4.3.1 NPR News

When started, the NPR News application downloads an XML document containing a list of story titles and URLs to corresponding thumbnail images, audio stories, and textual stories. A scrolling list of titles and matching thumbnails is then presented to the user (see Figure 4.8). Selecting any story in the list will then download the full contents of the story and any additional images. As a means of reducing network utilization and power-consumption, the application has been modified such that all image downloads are delayed until either a Wi-Fi connection is available or the cellular radio is woken by another request. However, to preserve the user experience, we would like to ensure that for the first few stories that a user would see, the thumbnail images would be downloaded and presented immediately.

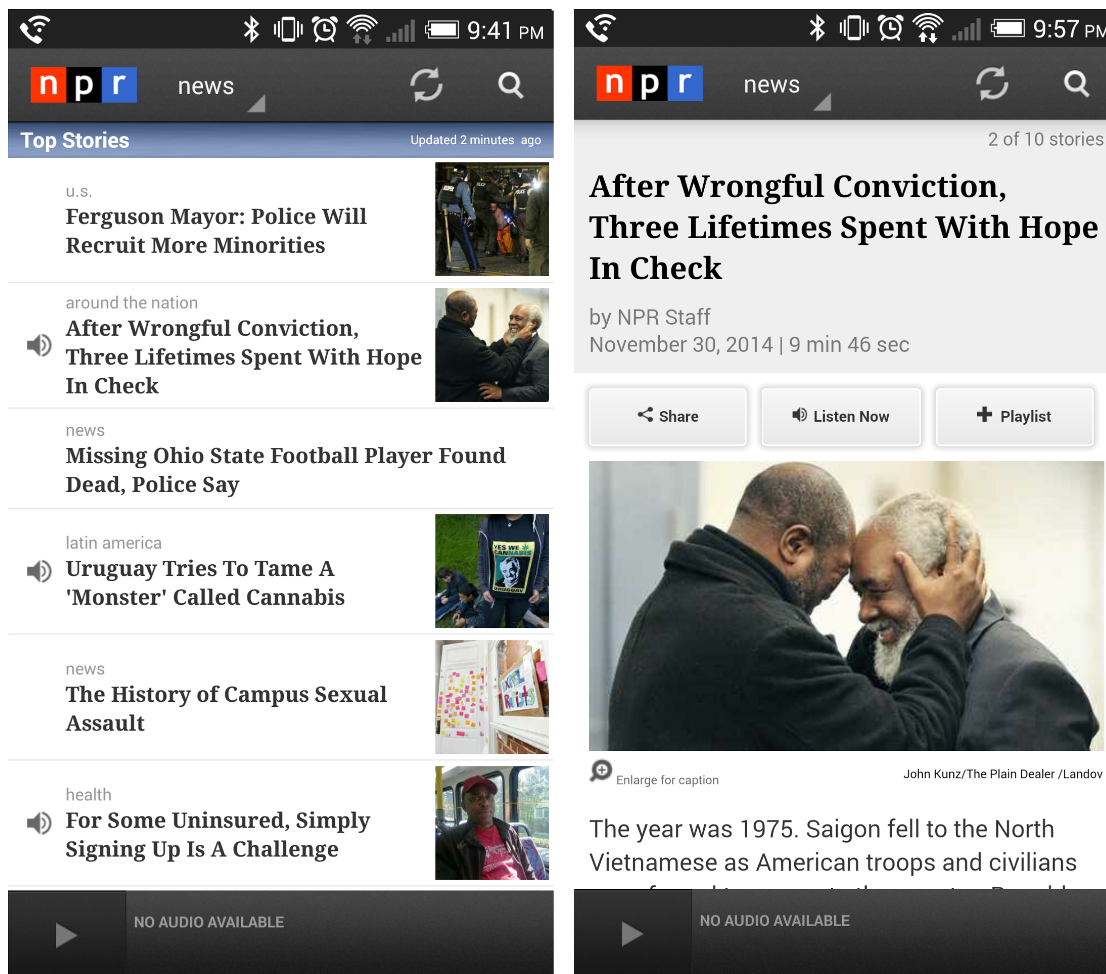

Figure 4.8. Screenshots from the NPR News application: looking at the list of stories (left) and reading a story (right).

All requests to download images, regardless of where they originate in the application, eventually reach the `readBitmapFromNetwork` method, which is responsible for the actual fetching of the image from the server (see Figure 4.9). This method contains a `@Wait` annotation that manages the trade-off between the energy saved and timeliness of fetching images. The `@Wait` annotation uses its default scoping and, the static analysis determines that objects `url` and `conn` are within its scope. The URLs to thumbnails for the first eight stories are assigned a `@DelayBudget` of zero, as those are the first stories presented to a user (line 6). A typical user takes approximately five

```
1 public View getView(int position, View view, ViewGroup parent) {
2   ...
3   String imageUrl = entry.getValue().getSrc();
4   if (imageUrl != null) {
5     if (position < 8}) {
6       @DelayBudget(0, imageUrl)
7     } eIse if (position < 16) {
8       @DelayBudget("5min", imageUrl)
9     }
10    Drawable cachedImage = imageLoader.loadImage(imageUrl,
11                new ImageLoadListener(position, (ListView) parent));
12  }
13 }
14
15 private static Bitmap readBitmapFromNetwork(String urlString) {
16   @Wait(UpTo="10min", For="Network.Active or WiFi.Connected")
17   URL url = new URL(urlString);
18   URLConnection conn = url.openConnection();
19   conn.connect();
20   @ClearBudget(url)
21   ...
22 }
```

Figure 4.9. The download of images is deferred until the radio or WiFi are on to save energy. Additionally, we ensure that download of the first 8 images is not delayed, the download of the next 8 stories may be deferred by up to 5 minutes, and the download of all other stores may be deferred by up to 10 minutes.

minutes to read or skim through these initial stories, so the next eight stories are assigned a `DelayBudget` of five minutes (line 8). All other images will be fetched ten minutes later, the maximum delay of the `@Wait`. A `@ClearBudget` is used to remove `url` the scope of `@Wait` annotation in line 16. The annotation is necessary to demarcate the time-critical part of the `url` life-cycle that occurs between lines 16 – 20. We note that the `url` object is never garbage collected since it includes a reference to `urlString`, which is referenced as part of the XML list of news stories that is always live. Without `@ClearBudget`, once an object's budget becomes zero the `@Wait` annotation would never defer the execution of network operations to save power.

The example illustrates how Tempus may be used to provide differentiated timeliness based on the budgets of `url` objects. Moreover, it shows the scope of `@Wait` annotations may be determined automatically using static analysis. The example also illustrates the need for providing fine-grained control over identifying the time-critical part of an object's life cycle.

The policy above assigns static budgets based on assumptions about the rate at which a user reads through content. An alternative approach would be to dynamically adjust the `@DelayBudget` of image URLs based on the current position of the user in the list of stories (see Figure 4.10). As soon as a `View` UI element is determined to be in the viewable area of the screen, the method `isInView` is called with a reference to the relevant object as an argument. This method assigns a `DelayBudget` of zero to the relevant URL String, forcing the download to begin immediately. The assignment of `DelayBudgets` in the `getView` method has also been modified. While the initial eight stories are downloaded immediately, all stories after that are assigned an initial budget relative to their position in the list, so that one new image is fetched once every minute that the user is interacting with the application. These changes introduce two benefits: (1) the download of news story thumbnails are more evenly spread out and

```
1  Map<View, String> viewToUrlMap = new HashMap<View, String>();
2
3  public void isInView(View view) {
4    String imageUrl = viewToUrlMap.get(view);
5    if(imageUrl != null) {
6      @DelayBudget(0, imageUrl)
7    }
8  }
9
10 public View getView(int position, View view, ViewGroup parent) {
11   String imageUrl = entry.getValue().getSrc();
12   if (imageUrl != null) {
13     if(position < 8) {
14       @DelayBudget(0, imageUrl)
15     } else {
16       @DelayBudget("1min" * pos, imageUrl)
17     }
18     Drawable cachedImage = imageLoader.loadImage(imageUrl,
19         new ImageLoadListener(position, (ListView) parent));
20   ...
21 }
```

Figure 4.10. Budgets for the `@Wait` annotation in Figure 4.10 are refined to ensure that the first 8 stores are not delayed and the budget of the remaining stores is computed dynamically based on their position in the news story list.

dependent on the number of stories presented and (2) the image for a story that is in view of the user is immediately fetched. This example demonstrates the capability of Tempus to manage dynamically assigned budgets.

4.3.2 CitiSense

CitiSense is an environmental air pollution monitoring application that utilizes a body-worn sensing device to accurately measure a user's exposure to a variety of pollutants (see Figure 4.1). The general structure of the CitiSense code is shown in Figure 4.11. Sensor readings are streamed to the user's phone over Bluetooth connection between the phone and sensor device. The sensor data is read in `ReadSensors`, written to disk in `Storage`, and uploaded to the remote in `Uploader`. Consistent with best programming practices for Android, each class operates on an independent thread to

minimize the impact of long operations on the user interface. `DisplayReadings` renders the sensor readings in real-time.

CitiSense must effectively manage the trade-off between energy and timeliness. The application includes two power management policies that operate in isolation. The first policy focuses on optimizing the energy consumed by networking. Specifically, energy savings may be achieved by deferring the upload of sensor data until another application turns on the radio. This power management behavior is specified by the `@Wait` annotation introduced in line 8. However, not all sensor readings are of equal importance in CitiSense: it is more valuable to upload unusually high pollution levels. Providing differentiated timeliness is achieved by defining the `NormalReading` and `UrgentReading` namespace (lines 47 and 50). Delay budgets of 10 and 0 minutes are assigned with the objects belonging to the `NormalReading` and `UrgentReading` namespaces, respectively (lines 48 and 51). We ensure that the delay introduced by `@Wait` considers the readings in the `NormalReading` and `UrgentReading` namespaces by specifying its scope to be:

```
NormalReading | UrgentReading
```

The implementation of the policy spans three threads that read, store, and upload the sensors that exchange data using shared queues. Pipeline processing as the one in this example are common in Android applications. Tempus can effective manage the energy-timeliness trade-offs across multi-thread processing pipelines.

The second policy takes advantage of the fact that pollution values do not change fast at the same location (see lines 42-43). Leveraging this insight, the energy consumed for collecting sensor data may be reduced by delaying the acquisition of new readings until the user moves from the current location. Obviously, we must ensure a minimal

```
1 class Uploader implements Runnable {
2   protected final BlockingQueue<Message> toUpload;
3   public void run() {
4     while(true) {
5       final Message m = toUpload.take();
6       @Wait(UpTo="10min", For="Network.Active",
7                    Scope="UrgentReading|NormalReading")
8       UploadHelper.upload(m);
9     }
10  }
11  public void addMessage(Message m) {
12    toUpload.put(m);
13  }
14 }
15
16 class ReadSensors implements Runnable {
17  protected final Handler handler;
18  protected final Uploader uploader;
19  public void run() {
20    while (true) {
21      @Wait(UpTo="10min", For="Location.Change",
22            Scope="DisplayReading")
23      final SensorReading reading = Sensor.read();
24      if (reading.getValue() > EXPOSURE_THRESHOLD) {
25             @Namespace("UrgentReading", reading)
26             @DelayBudget("0Min", reading)
27      } else {
28             @Namespace("NormalReading", reading)
29             @DelayBudget("10min", reading)
30      }
31      uploader.addMessage(reading);
32      handler.sendMessage(reading);
33    }
34  }
35 }
36
37 class DisplayReadings extends Activity {
38  @Namespace("DisplayReading", displayActive)
39  protected Object displayActive = new Object();
40  protected final Handler handler = new Handler(...) {
41    public void handleMessage(Message m) {
42       ... update UI ...
43    }
44  }
45  public void onResume() {
46   @DelayBudget("0Min", displayActive)
47   ...
48  }
49  public void onPause() {
50   @ClearBudget(displayActive)
51   ...
52  }
53 }
```

Figure 4.11. The application has two power management policies: (1) data acquisition is deferred until the user changes his location and (2) sensor readings are classified as either `NormalReading` or `UrgentReading`.

sampling rate that, in this example, is set to a reading every 10 minutes. CitiSense provides users the ability to view the pollution readings in real-time. In this case, data should be collected at the highest possible frequency to provide the user a comprehensive view of the pollution within her surroundings. This is achieved by controlling the delay budget associated with the `DisplayReading` namespace. When the user interface is brought in view (line 68) the budget of the `displayActive` object is set to zero. Similarly, when the user interface is no longer in view (line 74), the budget of the `displayActive` object is cleared. As a result, the `displayActive` object is no longer in the scope of the `@Wait` in line 42.

This example demonstrates the positive compositional properties of Tempus. The two power management policies operate in isolation. The developer can verify that this is the case by determining if there is any overlap between the scopes involve in the two policies. Moreover, the first policy includes multiple threads. Even in this case, the developer can reason about the aggregate behavior of the threads because of the timing semantics of Tempus. This is owed to two key design decisions on how timing is handled (1) the only operation that consumes time is a `@wait` operation (as opposed to the alternative of using real-time) and (2) the concurrent execution of `@Wait` annotations has predictable compositional semantics due to the *system delay* concept.

4.4 Experiments

In this section we evaluate Tempus and demonstrate it's efficacy in managing the trade-off between power consumption and timeliness in real mobile applications. Multiple versions of the NPR News and Citisense applications were developed, each including a different Tempus-driven power-management policy. Each application was then run on a Pantech Burst smartphone device with Android 4.0.4 and studied to determine average power consumption and the impact of the policy on the timing of certain operations.

Power consumption was measured using a Power Monitor from Monsoon Solutions [12] by modifying the device's battery to allow a direct bypass such that power was drawn from the monitoring device rather than the battery. All cellular communication was performed on the T-Mobile network in the San Diego metropolitan area.

As the behavior and power-consumption of these applications is highly dependent on input, it is important to evaluate them in a consistent manner. To do so, the NPR and Citisense applications were modified to record all user interactions and collected sensor readings. Each author was then tasked with using each application during a one week period. In the case of NPR, the application logged how a user scrolled through the list of stories in the application and which stories were selected to be read. In the Citisense application, traces were collected of each mobility state (stationary or mobile) as well as the values of any air quality measurements. These applications were then modified to allow 'playing back' such traces and to drive the operation of the applications automatically during experiments.

We also provide an evaluation of the overhead associated with the usage of Tempus. Specifically, we measure the time required to track and use budgets at runtime and the overhead associated with periodically forcing garbage collection to ensure a precise set of budgets when evaluating policies.

4.4.1 Power-Timeliness Trade-off

This subsection characterizes the power consumption of our case study applications under a variety of different budget constraints and usage patterns. Power consumption is dependent on both the usage patterns of an application as well as the quality-of-service requirements of the application. Tempus is not a power-management policy in itself, but rather a tool for composing and introducing power-management policies in an effective and safe manner. As such, these experiments are not intended to be a thorough

study of all possible power-management policies or usage patterns for mobile applications. Rather, these experiments demonstrate the correctness of Tempus's implementation while visualizing the potential impact of assigned budgets on power-consumption.

NPR News

The NPR News application provides the user with a list of recent news stories that can be selected and read. On start up, the application downloads an XML file that includes the names of stories, some content, and references to images that are used as thumbnails in the story list. The original implementation of the NPR News application downloads images in an *ondemand* fashion, in that it downloads images only when the user is actively trying to view them. A benefit of this policy is that it minimizes the amount of data downloaded by the application. However, this policy has an observable and negative impact on the user-experience: since images are not downloaded until after they are in view of the user, many UI elements may be empty when first viewed and eventually have an image 'pop-in' once the download is complete. An alternative to this approach would be to *prefetch* some or all of the images, so that they are immediately available for viewing by the user and thus reducing the frequency of, or totally avoiding, the jarring pop-in effect found in the ondemand version of the application. Given opportunities to piggy-back network transmissions with requests from other applications on the device, it may be possible to apply prefetching in a way that reduces the average time that a user spends waiting for images to populate UI elements while having negligible impact on power consumption when compared to a purely ondemand approach.

To demonstrate the versatility of Tempus for managing timeliness and efficiency in such an application, four different versions of the NPR News application were developed and evaluated, each using Tempus to manage the timing of image download requests. The implemented policies are as follows:

- **Ondemand**: images are only downloaded when needed to populate an in-view UI element;
- **Light Prefetch**: a batch of three images is prefetched for every two minutes that a user using the application, while in-view images are downloaded immediately;
- **Medium Prefetch**: a batch of six images is prefetched for every two minutes that a user using the application, while in-view images are downloaded immediately; and
- **Full Prefetch**: all images are downloaded immediately at the start of the application.

The reasoning behind the Light and Medium prefetching policies is that a user is more likely to view stories and images further down in the story list the longer that they are interacting with the application. These policies provide a middle ground between the Ondemand policy, which minimizes data usage, and the Full policy, which minimizes user observed delay. In practice, the ideal number of images to download in each batch and the frequency of prefetch attempts is highly dependent on application usage patterns. The policies and parameters selected here are intended to demonstrate a range of possible policies and their impact on the trade-off between power consumption and timeliness.

To implement these policies, the implementation of the NPR News application was updated using Tempus in three ways:

1. A `@Wait` annotation was introduced to the method responsible for downloading images so that requests are delayed until the cellular radio is first powered up by another application or request;
2. A `@DelayBudget` annotation was used to ensure that in-view images are downloaded immediately, if they have not been previously downloaded, by assigning

a budget of zero to the `String` object containing the path to the remote image resource, which is used by the aforementioned method to download the image; and

3. An additional `@DelayBudget` annotation was used to limit how long any image prefetch request is delayed by assigning a budget of 100 seconds to the `String` objects containing the paths to the remote image resources being prefetched.

In addition to these three annotations, a `Handler` object (provided by the Android framework) was was used to schedule prefetch requests in the Light and Heavy policies.

At compile time, the static program analysis was able to infer that the `String` object referred to in the `@Wait` annotated download method may have been assigned a budget when it was scheduled to be prefetched, came into view of the user, or both. This implied that the `@Wait` annotation must be considerate of the budgets of any objects that were annotated with `@DelayBudget` as a result of either of those two events.

As previously mentioned, usage traces of the original application were compiled for each of the authors. These traces were combined to create a trace representing 70 minutes of usage of the application, including starting the application, reading through the story list, selecting stories to be read, and closing the application. The application was then modified to allow playback of this trace file to automatically drive the operation of the application during experiments. The total amount of data used and time required for an image to appear in the UI for each 'use' of the application in the trace file was logged to a file at the end of each experiment. Each experiment was run five times for each policy, resulting in a total of 350 minutes of measurements for each policy.

Figure 4.12 presents the average system power consumption, user observed delay to view an image, and data usage for each policy when piggyback opportunities are presented once every three minutes. The Ondemand policy consumed the least amount of power and used the least amount of data during each run of the application, but it

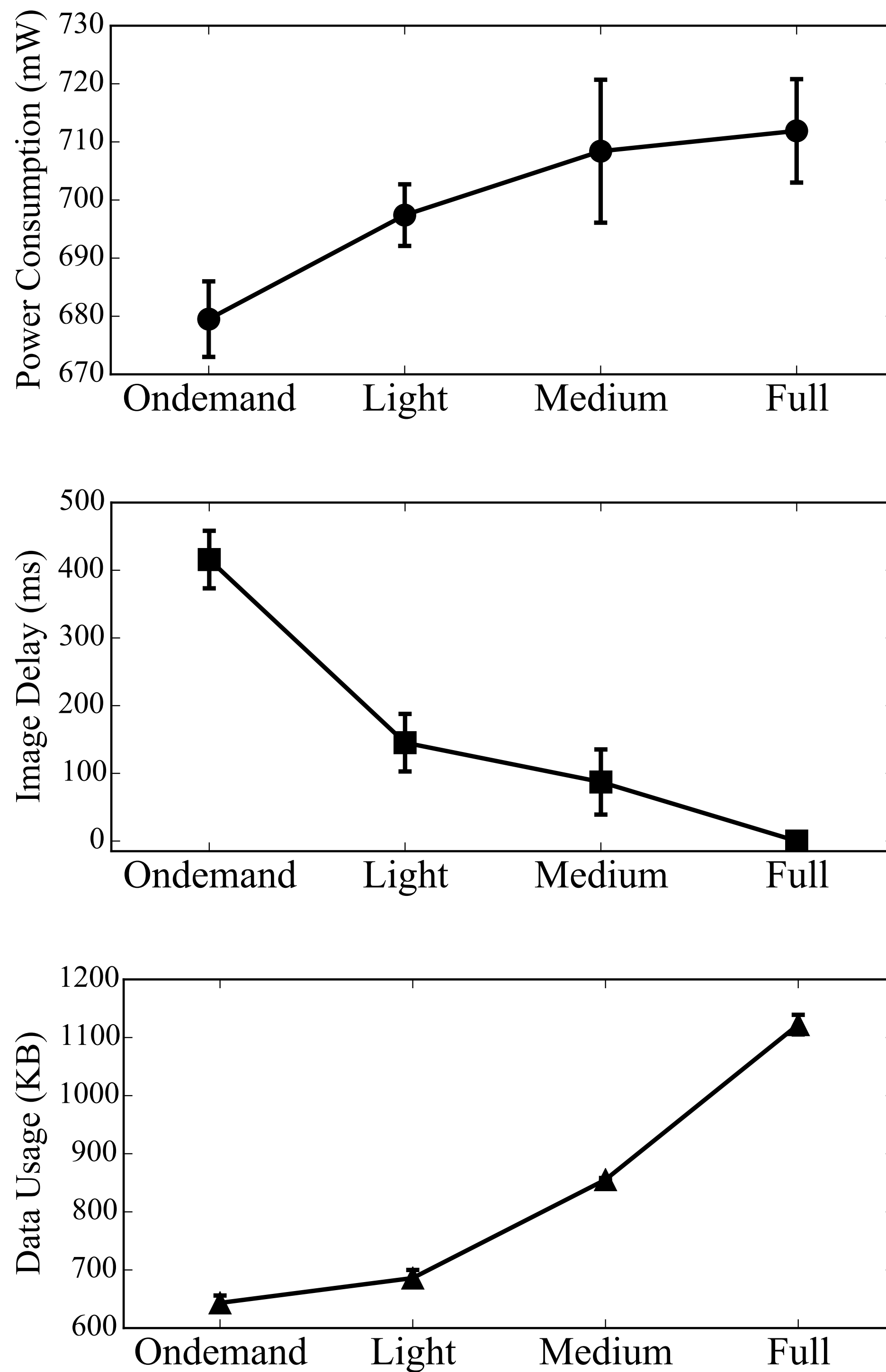

Figure 4.12. Impact of various policies on the NPR News application when piggybacking opportunities are available once every three minutes.

suffered the worst delay: in average, a user had to wait 416 ms for an image to appear in the application. In contrast, the Full Prefetch policy consumed the most power and used 74% more data than the Ondemand policy, but completely eliminated the observed pop-in of images by downloading all images at application start. The Light policy arguably provided the best results: data usage increased by only 7%, but the observed delay was only 145 ms, a reduction of 65%. The Medium policy further reduced the observed delay, down to 87 ms, but at the expense of additional data and power consumption. While each of the prefetch based policies significantly reduced the user observed delay, they came at the cost of additional data and power consumption.

Figure 4.13 presents results for each of the four policies when piggyback opportunities are instead presented once every minute. The increased frequency at which the radio is woken by other workloads impacts the performance of the policies in primarily two ways. First, the average time required to download and display an image is reduced in the Ondemand, Light, and Medium policies, as the radio will already be in a connected state and requires less time to begin transmission when compared to using a previously idle radio. Secondly, the power-consumption of the Light and Medium policies is more in line with that of the Ondemand policy, as the application is no longer waking a previously idle radio to prefetch images. In this set of experiments, the Medium policy is arguably the best policy, as it reduces image delay by 91% while negligible impact on power-consumption. In fact, even though it uses more data, the Medium policy consumes less power than the Light policy, as its larger cache of prefetched images reduces the likelihood of a ‘cache miss’ and can help avoid the radio having to be suddenly powered to download an image for a UI element.

Figure 4.14 presents the state of the budgets associated with nine images during a run of the NPR News application with the Light Prefetch policy. The URLs to the first three images to be prefetched are initially assigned a budget of 100 and placed into the

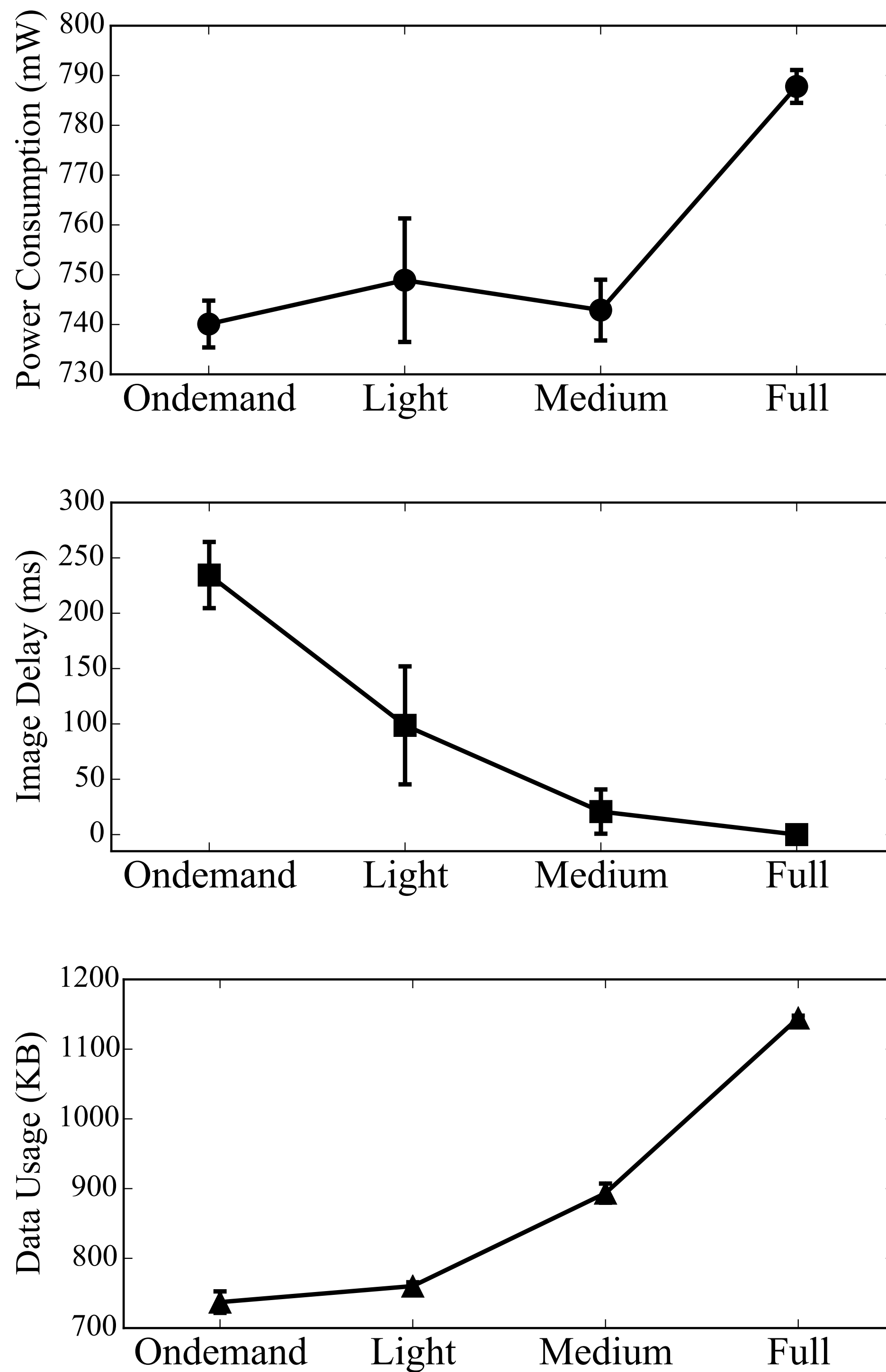

Figure 4.13. Impact of various policies on the NPR News application when piggybacking opportunities are available once every one minute.

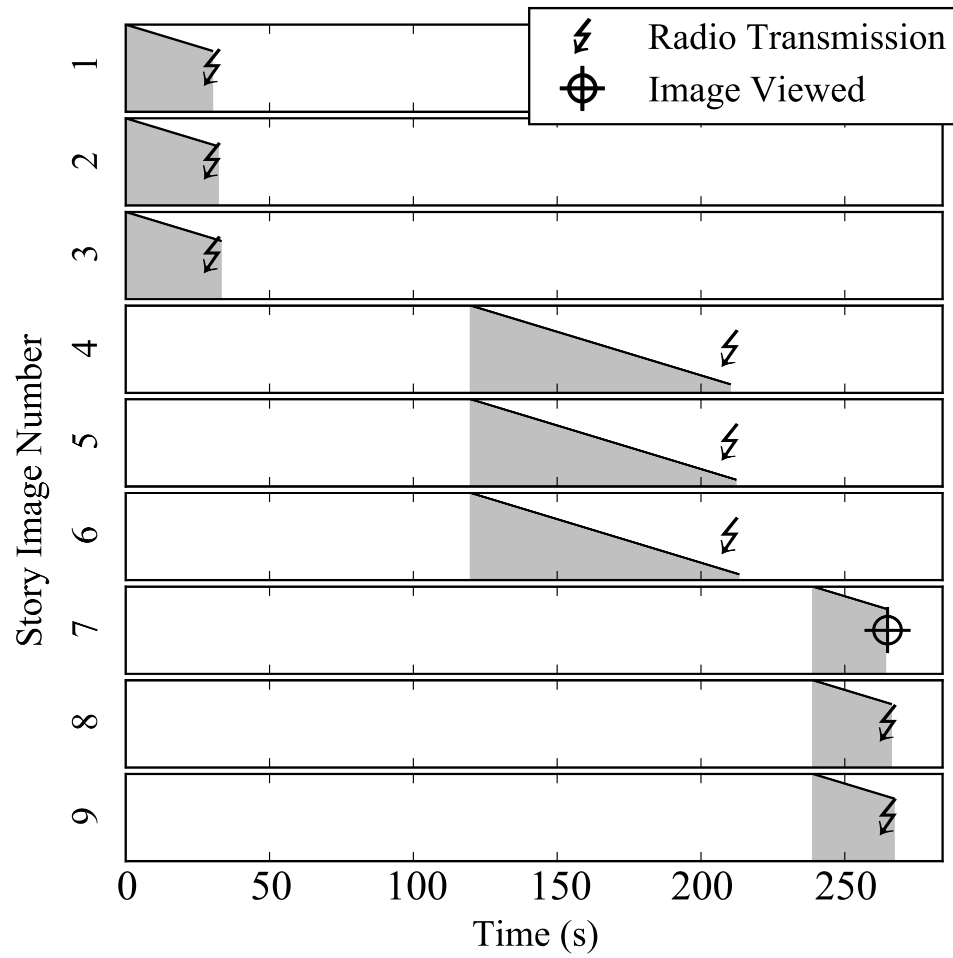

Figure 4.14. The state of budgets assigned to various images during a run of the NPR News application with the Light Prefetch policy.

queue of pending image download requests. A worker thread then reads the first request out of the queue, makes a call to the `@Wait` annotated download method, and is blocked until either the cellular radio is powered on or until the budget of any of the requests reaches zero. Some time later, a network request does in fact arrive, wakes the radio, and allows the blocked worker thread to continue with downloading the first image. The thread then proceeds to pull the next request out of the queue, and as Tempus sees that the radio is still powered from handling the previous request, allows the thread to continue immediately. Unlike the image requests that precede it, the blocking of the seventh image request is interrupted when the UI element containing that image is viewed by the user and assigned a new budget of zero, triggering its download. The eight and ninth images, though not in view, are downloaded as well, as the radio was powered to fetch the seventh image just moments before.

Citisense

The Citisense application collects air pollution measurements from a user-carried, Bluetooth-enabled sensing device and periodically uploads the location-tagged readings to a remote server for further analysis and to generate pollution warnings for other nearby users. While a user is determined to be moving (using a combination of accelerometer data and wireless network based localization), the application makes use of the GPS to generate precise location information. Otherwise, the application relies on less precise, but more efficient, wireless network based location information. Originally, the application would generate a new sensor reading once every six seconds and upload batches of readings once every ten minutes. This behavior implies that sensor readings would, on average, reach the server approximately five minutes after they are generated. Tempus was used to improve upon this basic application behavior in three ways:

1. The primary objective of our changes is to to reduce the average delay encountered in the upload of high measurements, as they are important for accurate pollution modeling and the timely warning of other nearby users. To do so, `@DelayBudget` was used to assign a budget of zero to any measurements that fall above the threshold for 'Good' air quality, defined by the EPA as an Air Quality Index above 50 [1];

2. Rather than simply waiting for exactly ten minutes between upload attempts, the upload logic was updated to `@Wait` up to ten minutes for an opportunity to piggyback on the waking of the radio by another application running on the device; and

3. As readings from the same location typically show very little variance, the application was updated to `@Wait` up to one minute for the user's location to change before sampling the sensor, reducing the cost of Bluetooth communication during stationary periods.

These three changes were implemented using only three Tempus annotations and a few trivial changes to the code, such as removing the call to `Thread.sleep()` in the sensor reading upload thread, which became redundant after introducing the `@Wait` annotation. This new version of the Citisense application was then evaluated, using traces of real sensor data from users, to determine the average power consumption and time between upload attempts when the user is stationary and when mobile. Unlike the previously presented NPR News experiments which included frequent opportunities to piggyback transmissions, the set of CitiSense experiments were run without any other workloads generating piggybacking opportunities. This represents a 'worst case scenario', where each upload by Citisense is responsible for waking the radio and the average time between batched uploads is maximized.

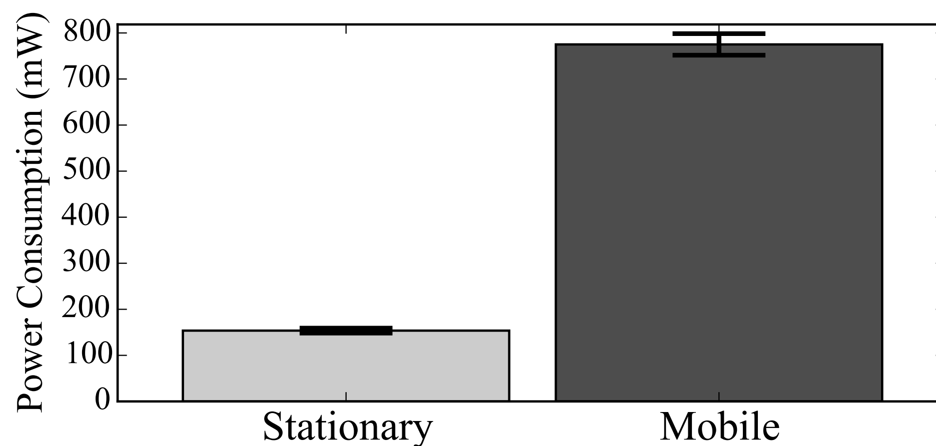

Figure 4.15. Average power consumption in the Citisense application when processing traces from stationary and mobile users.

As can be seen in Figure 4.15, the power consumption of the application while the user is mobile far exceeds that of stationary operation. The requirement that mobile data be tagged with precise GPS data contributed approximately 318 mW of power consumption in our experiments. In contrast, the use of network based localization techniques added only 14 mW of power consumption over the baseline, as such techniques make use of information about access points and cell towers that are passively collected by the device during normal operation. The reduced sampling rate of the Bluetooth sensor while stationary contributed approximately 8 mW of savings. The remaining difference in power consumption is due to the policy used to upload sensor readings and the observation that stationary readings typically come from clean, indoor environments while mobile readings typically come from outdoor environments, especially on or near roadways as users commuted between their home and office. Nearly all sensor readings that are collected while a user is stationary fall into the category of 'Good' quality air and can be batched to be uploaded once every ten minutes. However, mobile users often encountered stretches of poor air quality while traveling, receiving many readings that

fall above the threshold for ‘Good’ air. Each of these high readings would initiate an upload. As these poor readings often followed one another, meant that uploads would often include a single sensor reading and were spaced only seconds apart. During such stretches of constant uploading, the power consumption of the application averaged 914 mW. This implies that, due to the requirement that the server be notified immediately of any high readings, power consumption in the Citisense application is highly dependent on the quality of air that a user is exposed to.

While the savings from Bluetooth duty cycling do not come near making up for the high cost of frequent network transmission, the primary objective of our changes was to reduce the time required for important readings to reach the server. Where the previous version of Citisense would upload readings, regardless of their content, on average five minutes after they were sampled, the new version ensures that high readings are immediately flushed. In our experiments, high readings were transmitted and stored on the server approximately five seconds after they were sampled. Since the majority of our users’ time was spent in clean air environments, the average time to upload for ‘Good’ readings remained approximately five minutes. However, if presented with opportunities to piggyback transmissions, the average time to upload such readings would drops significantly.

4.4.2 Budget Tracking Overhead

As previously mentioned, Tempus builds upon and provides a new interface for accessing the information provided by the standard device hardware monitoring interfaces provided by the Android framework. As static program analysis is utilized at compile time to reason about which object references are impacted by `@Wait` annotations, the only significant overhead introduced by Tempus at runtime is that of tracking budgeted objects.

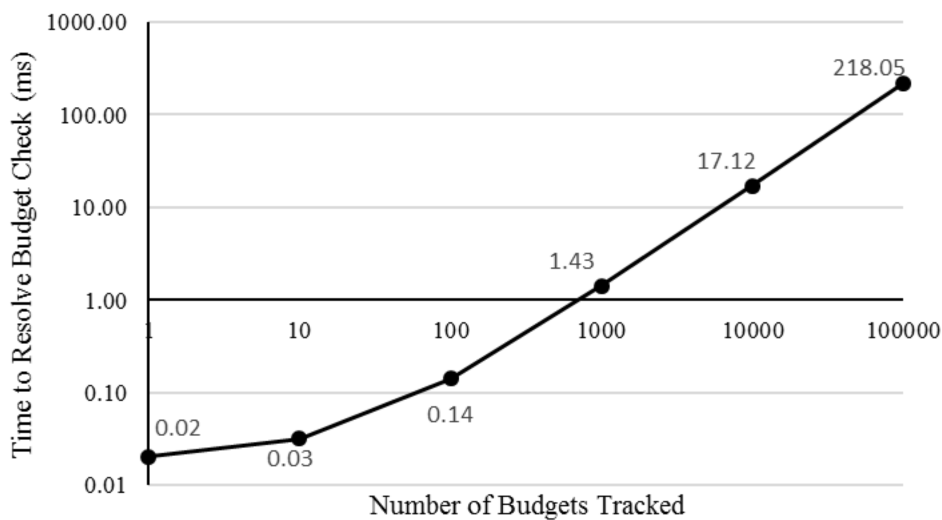

Figure 4.16. Overhead associated with the tracking of budgets for various numbers of objects. Note the logarithmic scale.

Figure 4.16 presents the overhead associated with "using" the budgets of all objects that have been assigned a particular namespace. As expected, this overhead is highly dependent on the number of objects associated with the targeted namespace. When tracking the budget of a single object, overhead was measured to be approximately 0.02 ms. Even when tracking and adjusting the budgets of 1,000 items, overhead was measured to be only 1.43 ms. When tracking 10,000 and 100,000 objects, the overhead begins to become more noticeable: 17.12 ms and 218.05 ms, respectively. However, we expect that the most applications will track far fewer than 1,000 budgets at a time. In the study of the Citisense application, only one object was tracked at a time: the high measurement responsible for flushing all batched readings. In the NPR News application, budgets were tracked for each of the approximately twenty news stories presented to the user at a time.

Garbage Collection. The `@ClearBudget` annotation was earlier presented as a mechanism for notifying Tempus to stop tracking the budget of a particular object as

soon as the developer knows that timely references to the object are no longer required for correct application behavior. However, the use of this annotation is not required to ensure correct application behavior at runtime: budgets can only provide tighter, and thus more conservative, bounds on when operations must be executed. While an obsolete budget will not cause any other budget deadline to be missed, it may however limit how long operations may be delayed by and thus limit potential power-savings from energy management policies.

As previously mentioned, garbage collected objects and their budgets are automatically removed from tracking at runtime. This update helps maintain a more precise set of tracked budgets, even when it is too difficult for a developer to reason about when an object is "done" and can be forgotten. The garbage collector is periodically invoked by the Android runtime framework, but the frequency of this collection is dependent on a variety of factors, including the application's memory footprint. As such, budgeted objects that are no longer referenced may have to wait long periods of time before they are finally garbage collected. This motivated the inclusion of the optional `AggressiveGC` argument to the `@Wait` annotation, as it forces garbage collection before checking budgets that may impact a Tempus introduced delay.

The authors experimented with making `AggressiveGC` the default behavior of the Tempus runtime, as it helped maintain as small and accurate a set of tracked budgets as possible. However, the overhead associated with garbage collection, and its potential impact on user experience, led to the decision to dropping this as this default behavior. Figure 4.17 presents the results of experiments that were conducted to measure the impact of garbage collection on the execution time of a long-running, CPU-bound operations. While the exact numbers will be highly dependent on the particular device being tested, as well as it's version and implementation of Android, it was determined that frequent requests for garbage collection were prohibitively expensive. If a developer would

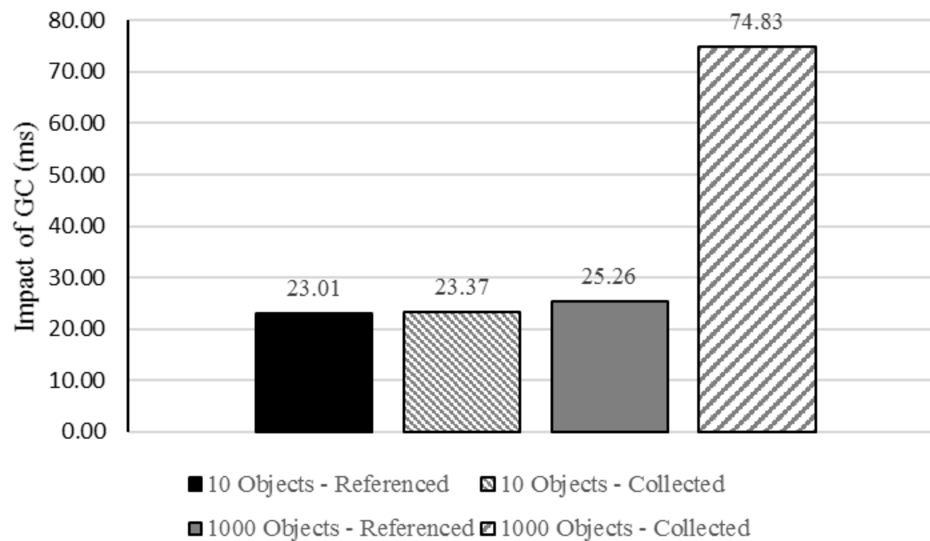

Figure 4.17. Garbage collection overhead when Tempus is tracking objects that are either all referenced elsewhere in the program or are all garbage collected.

like to improve the precision of budget tracking at runtime, they may either introduce `@ClearBudget` annotations to their program or, if performance is not as critical as efficiency, by making use of `AggressiveGC`.

4.5 Conclusion

This chapter presented Tempus – a novel power management aimed at applications that must manage the trade-off between energy savings and timeliness. Tempus saves power by deferring the execution of power hungry operations until the device enters a state that minimizes the cost of that operation. The impact of power management on the timeliness of operations is managed by associating delay budgets with objects. The static analysis and run-time environment ensure that the processing of any object is not delayed by more than the user-specified budget.

We showed that our approach is both expressive and flexible by annotating introducing power management into two realistic applications. Tempus is able to provide

differentiated delays for different operations in mobile applications. Additionally, owning to its simple compositional semantics, the developer can reason about the properties of applications that include multiple power annotations that operate concurrently. Our experiments indicate that Tempus introduces a relatively small overhead to tracking budgets on objects. In practical applications the number of objects is relatively small. Detailed experiments from the studied applications show that Tempus may save energy while meeting timeliness constraints.

4.6 Acknowledgements

This chapter, in part is currently being prepared for submission for publication of the material. Nikzad, Nima; Chipara, Octav; Griswold, William G. The dissertation author was the primary investigator and author of this material.

Chapter 5

Conclusion

Even though continuously-running mobile applications typically 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. While 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, there are also significant software engineering challenges: (1) the code for power management tends to be complex, and (2) power management optimizations should be postponed until the application's requirements are set.

To address these challenges, this dissertation presented Annotated Programming for Energy-efficiency (APE), 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. Language constructs and accompanying static program analyses were presented that allowed a developer to specify constraints for delay-sensitive and -intolerant methods and objects. The APE compiler and runtime generate and insert effective power-management policies within the target application while ensuring that all delay related constraints are satisfied at runtime.

APE's approach was shown 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. The efficacy of APE's approach was further demonstrated in a study of introducing power-management policies to six CRM applications. Measurements have shown that the policies generated by the Policy Generation Engine 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.

Although APE can simplify the mechanics of writing power management code, APE does not help a developer reason about the impact of power annotations on timeliness. Mentally reasoning about the many possible execution paths in a large, object-oriented, multi-threaded application is taxing, at best. An additional limitation of a path-centric approach to reasoning about power management is that it is difficult to write power management policies that crosscut many modules of an application. To address this challenge, this dissertation presented *Tempus* — a new paradigm for writing power-

management policies that allows developers to reason about their impact in terms of objects being delayed rather than execution paths. Specifically, an object that contains delay sensitive data can be annotated with a *delay budget*, which bounds the total delay that the object experiences due to power-management-related delays. Power-management policies are constructed using annotations that specify a desired hardware state, delaying execution of costly operations while "spending" the delay budgets of impacted objects. Once an object's delay budget has been exhausted, power-management policies that would impact the object are ignored so that the object may be used in a timely fashion. Static program analysis and runtime support ensures that, along all paths to instructions that reference an object with a delay budget, the total delay introduced by a Tempus power-management policy does not exceed the object's remaining budget.

In this dissertation, we have demonstrated that a high-level annotation language can be used to effectively describe and implement the diverse set of energy-management policies typically found in mobile applications. Additionally, these annotation-based policies can be efficiently evaluated at runtime to reduce the power-consumption of an application while improving the maintainability of a code base through an improved separation of concerns.

Bibliography

[1] Air Quality Index (AQI) Basics. http://airnow.gov/index.cfm?action=aqibasics.aqi.

[2] Android Developers: Best Practices. http://developer.android.com/guide/practices/index.html.

[3] Android Developers: Processes and threads. http://developer.android.com/guide/components/processes-and-threads.html.

[4] AndroidAnnotations. http://androidannotations.org/.

[5] Andstatus project. http://andstatus.org/.

[6] Bittorrentsync. http://www.bittorrent.com/sync.

[7] Build Efficient Apps: AT&T Developer Program. http://developer.att.com/developer/forward.jsp?passedItemId=7200042.

[8] Dropbox. https://www.dropbox.com/.

[9] Fitbit. http://www.fitbit.com/android.

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

[11] K-9 mail. http://k9mail.org/.

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

[13] Npr news android application. http://www.npr.org/services/mobile/android.php.

[14] ohmage. http://ohmage.org/.

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

[16] Weatherlib. http://survivingwithandroid.github.io/WeatherLib/.

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

[18] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 29. ACM, 2014.

[19] Martin Azizyan, Ionut Constandache, and Romit 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.

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

[21] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. In *ACM SIGPLAN Notices*, volume 47, pages 831–850. ACM, 2012.

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

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

[24] Syed Shabih Hasan, Farley Lai, Octav Chipara, and Yu-Hsiang 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.

[25] Azeem J Khan, Kasthuri Jayarajah, Dongsu Han, Archan Misra, Rajesh Balan, and Srinivasan 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.

[26] Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An analyzable annotation language. In *ACM SIGPLAN Notices*, volume 37, pages 231–245. ACM, 2002.

[27] Mu Lin, Nicholas D Lane, Mashfiqui Mohammod, Xiaochao Yang, Hong Lu, Giuseppe Cardone, Shahid Ali, Afsaneh Doryab, Ethan Berke, Andrew T Campbell, et al. Bewell+: multi-dimensional wellbeing monitoring with community-guided user feedback and energy optimization. In *Proceedings of the conference on Wireless Health*, page 10. ACM, 2012.

[28] Emiliano Miluzzo, Nicholas D Lane, Kristóf Fodor, Ronald Peterson, Hong Lu, Mirco Musolesi, Shane B Eisenman, Xiao Zheng, and Andrew 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.

[29] Prashanth Mohan, Venkata N Padmanabhan, and Ramachandran 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.

[30] Nima Nikzad, Octav Chipara, and William G Griswold. Ape: an annotation language and middleware for energy-efficient mobile application development. In *Proceedings of the 36th International Conference on Software Engineering*, pages 515–526. ACM, 2014.

[31] Nima Nikzad, Nakul Verma, Celal Ziftci, Elizabeth Bales, Nichole Quick, Piero Zappi, Kevin Patrick, Sanjoy Dasgupta, Ingolf Krueger, Tajana Šimunić Rosing, and William 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.

[32] Abhinav Pathak, Abhilash Jindal, Y Charlie Hu, and Samuel 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.

[33] Joseph Polastre, Jason Hill, and David 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.

[34] Dan Quinlan, Markus Schordan, Richard Vuduc, and Qing Yi. Annotating user-defined abstractions for optimization. In *20th International Parallel and Distributed Processing Symposium, 2006*, pages 8–pp. IEEE, 2006.

[35] Lenin Ravindranath, Sharad Agarwal, Jitendra Padhye, and Chris Riederer. Procrastinator: Pacing mobile apps' usage of the network. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 232–244, New York, NY, USA, 2014. ACM.

[36] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[37] Vaidyanathan Srinivasan, Gautham R Shenoy, Srivatsa Vaddagiri, Dipankar Sarma, and Venkatesh Pallipadi. Energy-aware task and interrupt management in linux. In *Ottawa Linux Symposium*, 2008.

[38] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

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

[40] Fengyuan Xu, Yunxin Liu, Thomas Moscibroda, Ranveer Chandra, Long Jin, Yongguang Zhang, and Qun 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.

[41] Wei Ye, Fabio Silva, and John 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.