Managing the Energy-Delay Tradeoff in Mobile Applications with Tempus

Nima Nikzad[†], Marjan Radi[‡], Octav Chipara[‡], and William G. Griswold[†]
[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]Department of Computer Science, University of Iowa
nnikzad@cs.ucsd.edu, marjan-radi@uiowa.edu, octav-chipara@uiowa.edu, wgg@cs.ucsd.edu

ABSTRACT

Energy-efficiency is a critical concern in continuously-running mobile applications, such as those for health and context monitoring. An attractive approach to saving energy in such applications is to defer the execution of delay-tolerant operations until a time when they would consume less energy. However, introducing delays to save power may have a detrimental impact on the user experience. To address this problem, we present Tempus, a new approach to managing the trade-off between energy savings and delay. Tempus saves power by enabling programmers to annotate power-hungry operations with states that specify when the operation can be executed to save energy. The impact of power management on timeliness is managed by associating delay budgets with objects that contain time-sensitive data. A static analysis and the run-time service ensure that power management policies will not delay an object more than its assigned budget. We demonstrate the expressive power of Tempus through a case study of optimizing two real-world applications. Furthermore, laboratory experiments show that Tempus may effectively manage the energy-delay trade-off on realistic workloads. For example, in a news application, five Tempus annotations may be used to create a policy that reduces the latency of downloading images 10 times compared to the original implementation without affecting energy consumption. Our experiments also indicate that the overhead of tracking budgets in Tempus is small.

Categories and Subject Descriptors

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

General Terms

Design, Languages, Performance

Keywords

Mobile applications, energy-efficiency, annotations, programming

1. INTRODUCTION

In recent years, a growing class of continuously running mobile (CRM) applications has helped turn smartphones into a critical part of our everyday lives. Examples include cloud storage applications, such as Dropbox [3] and BitTorrent Sync [2], and various health and context-monitoring applications, such as Fitbit [4], CitiSense [12], BeWell+ [10], and Ohmage [7]. Even though these applications operate at low duty cycles, cumulatively they have a large impact on the battery life of a device due to their periodic use of power-hungry system resources, such as the cellular radio for networking or the GPS for localization. Programmers must, therefore, employ sophisticated power management schemes to reduce their energy consumption.

CRM applications may periodically collect and process sensor data, download application updates, and synchronize data with a remote server. Since most of these operations are *delay-tolerant* [11, 14], an attractive approach to saving power is to defer the execution of power hungry operations until the device enters a power state that minimizes the cost of that operation. This approach can generate significant power savings as the energy cost of an operation varies significantly depending on the device's power state. For example, briefly accessing the network can incur significant power costs if the network device is currently off, but can incur almost no additional cost if another application has already caused the device to be turned on. However, any given operation can vary in its tolerance to delay based on the situation, such as processing normal versus abnormal readings from a sensor. Thus, the introduction of delays to conserve energy can negatively affect the user's experience. Therefore, it is essential for the developer to be able to situationally control the delay that power management introduces to *manage the trade-off between energy savings and delay effectively.*

The central contribution of this paper is the development of a novel programming model that provides precise control of the energy-delay trade-off. We overcome two key challenges to meet this goal: (1) specifying how delays should not impact the user experience in an existing program, without having to rewrite or restructure it; and (2) enabling both developers (and static analysis tools) to determine the net impact of multiple programmer-introduced power-related delay policies in a multi-threaded application (i.e., providing a compositional semantics).

In this paper, we introduce *Tempus* – a new paradigm for writing power-management policies that manage the energy-delay tradeoff. Power management policies are added to the program using annotations that specify a desired hardware state when a power-hungry operation should be executed. The time that the system may wait for a power-hungry state is controlled by annotating objects that carry time-sensitive data with a *delay budget*. The budget is "consumed" when an operation that uses the object is delayed to save power. Once an object's delay budget has been exhausted, any power-management policies that would further delay any processing of the object are ignored so that the object will be used in a timely fashion. Static analysis is used to determine the objects that are impacted by the delay of an operation. The runtime service enforces the global invariant that *an operation may never be delayed more than the budget of any object it impacts*. We make the following contributions:

- We present a novel paradigm and annotation language for specifying power management policies. Tempus may specify differentiated delay-energy trade-offs on execution paths by associating delay budgets with objects carrying time-sensitive data. This approach keeps the burden of annotating applications to a minimum.
- We present a new *may-delay* static analysis that determines the impact of power annotations on objects carrying time-sensitive data. The results of the static analysis are used to generate code that integrates efficient object budget tracking into the application. The main abstraction of the run-time is the `TempusLock`, which integrates concurrency and power management. The run-time *guarantees* that power management will not delay an object more than its assigned budget.
- We demonstrate the effectiveness and flexibility of Tempus by introducing power-management in two real applications. The case studies highlight how a developer can compose *safe and effective* power-management policies while avoiding the burden of manually reasoning about paths of execution. The case studies are complemented by experimental results characterizing the energy savings and timeliness properties of the applications. We show that a few annotations significantly change the energy-delay profile of an application. For example, the NPR News app, the addition of five annotations implement a policy that reduces the latency of downloading images by 10 times over the original version without affecting energy consumption. Additionally, Tempus effectively coordinates the execution of power management policies with minimal overhead.

2. PROBLEM FORMULATION

CRM 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 balance energy savings against user experience. Through an example, we highlight the challenges of writing such policies and introduce the basic concepts of Tempus informally. We will formalize these concepts in Section 3.

Consider an application that processes sensor data to detect health-related events, generates health reports, and uploads them to a server (see Figure 1). The `generateReport` method is responsible for generating upload-ready `Report` objects on one thread, while `uploadReport` uploads them to the server on another thread. For the time-being, ignore the bolded annotations.

Suppose that we would like to reduce the energy consumption of the application by delaying the network uploads. However, the application must provide *differentiated* energy-delay trade-offs depending on whether the report includes a critical health event or not.

Power Management: A common technique for achieving significant energy savings is to coordinate the access to power-hungry resource in order to minimize their usage time and offset startup and shutdown costs. In our example, we may defer the `uploadReport` *operation* since the uploading of most health reports is delay tolerant. A reasonable policy might be to wait for up to twenty minutes for a Wi-Fi connection to be established or for another application to wake the cellular radio. Implementing this policy without the help of a system like Tempus requires event-based programming that tracks hardware states along with multi-threaded code that blocks the network thread until the device enters the desired state. In previous work, called APE [11], we showed that a large class of power management policies can be specified as (simplified) timed automata that specify the power-saving state the device should enter prior to executing the power-hungry operation. We showed such a restricted automaton can be declaratively represented as Java annotation that simplifies the mechanics of writing power management policies. Tempus builds on this mechanism (see lines 18 – 19), adding the capability for specifying how delays should be limited according the application's state. Thus, the novelty of Tempus is its ability to effectively manage the energy-delay trade-off.

Data-centric Approach: Since the power-saving delays are introduced along execution paths, a natural way to consider managing the energy-timeliness trade-off would be to associate timing constraints with different execution paths in the application. These constraints would control the maximum delay that `uploadReport` may be deferred to save energy. However, it is our experience that sensitivity to a delay often does not map naturally to execution paths. As a result, many paths must be considered and annotated to achieve the desired result. Not only is the result verbose, but also reasoning about paths in large, object-oriented, multi-threaded applications is difficult for the developer.

A better alternative, used by Tempus, is to associate budgets with delay-sensitive data (objects) and guarantee that their use cannot be delayed beyond their budgets. In our example, the programmer has annotated `generateReport` to associate a delay budget to some of the `Report` objects it produces. The `uploadReport` method will consume these `Report` objects, and use their budgets to modify the behavior of the `@Wait`, thus balancing power management and timeliness. This approach effectively allows the programmer to concisely control the timing behavior of the application over all possible execution paths in a predictable manner.

Timing Semantics: Another important question is related to how time is modeled in our system. An obvious choice would be to adopt real-time semantics according to which budgets are relative deadlines specifying by when op-

```
1  Report generateReport(Data data) {
2   if (healthStatus == CRITICAL_STATUS) {
3     // Generate time-sensitive report for upload
4     @DelayBudget(0,criticalReport)
5     Report criticalReport = new Report(data);
6     ...
7     return criticalReport;
8   } else {
9     // Generate delay-tolerant report for upload
10    Report normalReport = new Report(data);
11    ...
12    return normalReport;
13  }
14 }
15
16 void uploadReport(Report report) {
17    URL url = new URL(SERVER_ADDR);
18    @Wait(UpTo="20min", For="WiFi.Connected
19    OR Network.Active AND (Cell.4G OR Cell.3G)")
20    HttpURLConnection conn = (HttpURLConnection) url.
          openConnection();
21    ObjectOutputStream out = new ObjectOutputStream(conn.
          getOutputStream());
22    out.writeObject(report);
23 }
```

Figure 1: The `@DelayBudget` annotation (line 4) ensures that the upload of `Report` objects related to critical health events are not subject to any delays introduced by power annotations (line 18) when uploaded.

eration should be completed and time passes according to the wall-clock. Unfortunately, real-time semantics would focus our attention on trying to predict how long operations such as `uploadReport` would take. The total delay of `uploadReport` is subject to significant variability as it depends on many factors including availability of a connection, link quality, and congestion level. An alternative is to observe that the total latency of the operation includes the time spent saving energy and the operation's time. Tempus shifts the attention of the developers from predicting the highly variable execution times to bounding the additional time an operation may be delayed to save energy. This approach is further supported by the empirical observation that in delay-tolerant applications budgets are significantly larger than execution times.

3. DESIGN

Tempus provides an intuitive programming model for managing the trade-off between energy and timeliness. An application is modeled as a sequence of operations. The *scope* of an operation is the set of objects that may be referenced on any path starting with the considered operation. The programmer annotates an object that contains time sensitive data with a *delay budget* and a power-hungry operation with a *power-saving state*. When a wait annotation (operation) is reached, it pauses the thread of execution and begins consuming the budgets of the objects in its scope. The thread resumes when the device enters the desired power-saving state or the budget of any object in its scope is depleted. Therefore, the invariant enforced by Tempus is: *an operation may never be delayed more than the budget of any object in its scope.*

Tempus has three components: language annotations, a translator, and a run-time service. The programmer uses the Tempus annotations to specify the delay budgets of objects and the power management policies of an Android application. The translator employs a static analysis to determine the impact of power annotations on time-sensitive objects. The translator produces an Android application where annotations are replaced with code that implements power management policies and initializes object budgets. The run-time service tracks hardware states and object budgets to orchestrate the execution of power management policies. The remainder of the section discusses the annotations, static analysis, and run-time service, respectively.

3.1 Annotation Semantics

Next, we formalize the semantics of Tempus annotations.

`@DelayBudget/@ClearBudget`: Tempus's core abstraction is that of a *delay budget* that is associated with an object. Consider a variable v in the application code. The annotation `@DelayBudget`(B_v, v) assigns a budget of B_v to the object referenced by v.[1] If a budget was already assigned to v, its value is updated to equal B_v. The tracking of an object's budget is stopped by using annotation `@ClearBudget`(v).

Java is a garbage-collected language that provides only indirect control over when the memory of unreachable objects is reclaimed. Accordingly, Tempus may conservatively decide to stop waiting for a power-efficient state due to the budget of an unreachable object.[2] `@ClearBudget` provides the programmer explicit control for removing such an object from consideration in determining delays. In our experience, `@ClearBudget` is seldom necessary, as our static analysis and garbage collection together usually determine the relevance of objects with sufficient accuracy.

`@Namespace/@ClearNamespace`: A namespace is a named group of objects. This is useful for providing an alternate scope for a `@Wait` annotation (discussed next), overriding the one calculated by the Tempus static analysis. The annotation `@Namespace`(v, l) adds the object referenced by v to namespace l. An object may belong to multiple namespaces. The annotation `@ClearNamespace`(v, l) removes v from namespace l.

`@Wait`: The `@Wait` annotation defers the execution of an operation until either the device enters a power-saving state or the timing constraints are violated. 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 the mechanism used for specifying the power state E from APE [11]. In its simplest form, E is a boolean expression composed of one or multiple built-in terms. A term refers to the state 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 cellular network becomes active or the WiFi is connected.

Thread τ is blocked until either (a) the device enters a state in which expression E holds, (b) the annotation's timeout D_P is reached, or (c) the budget of one of the objects in the scope of the `@Wait` is exhausted. We call the time

[1]Here and throughout this section, we will variably refer to *variables* and the *objects* they reference. Variables are especially pertinent at compile-time, when no objects yet exist. Thus, variables are used conservatively as proxies for the objects they will reference at runtime. Objects are pertinent during runtime, as it is the time budgets of actual objects – values – that are tracked.

[2]The Tempus runtime maintains a weak reference to each budgeted object, which does not influence the garbage collector. We note that conservatively budgeting for an unreachable object does not violate timeliness guarantees. However, the power savings are potentially reduced.

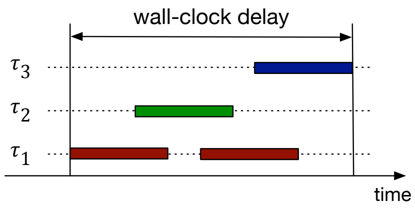

Figure 2: The execution of three concurrent `@Wait` annotations on threads τ_1, τ_2, and τ_3 that have the same object o in their scope. The filled boxes indicate when a thread is running. The delay observed by v is the *wall-clock delay* i.e., the duration of time while at least one thread is running.

the thread τ is blocked while operation P is deferred as the *operation's delay* ($\Delta(P, \tau)$). Since `@Wait` blocks τ, the processing of instructions following a `@Wait` is delayed. By default, the scope of a `@Wait` is conservatively determined by the static analysis. Accordingly, we define the *default scope* to include any object that has a budget *and* may be referenced on any path starting at the `@Wait` annotation. Note that a `@Wait` may delay the execution of other threads that are waiting for a notify signal from τ whose delivery is delayed by `@Wait`. The static analysis described in the next section addresses the difficulties of determining the scopes of `@Wait` in concurrent applications. We allow the programmer to override the scope of a `@Wait` by adding an additional `Scope` argument. The `Scope` argument may include both object references and namespaces separated by the "|" operator, forming their union.

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 its scope and the annotation timeout D_P.

$$\Delta(P, \tau) \leq \min(D_P, \min_{v \in \text{scope}} B_v) \tag{1}$$

where, B_v is the budget of the object referenced by v.

After the execution of a `@Wait`, the budgets of all objects in scope are 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 referenced by v is in the scope of multiple `@Wait` annotations that are executed concurrently requires more careful handling. In this case, the operations that involve v are delayed according to the *wall-clock delay* that measures the time at least one of the threads is executing. As an example, consider Figure 2 in which three annotations that involve v are executed on three threads τ_1, τ_2, and τ_3. In this case, the time that operations on v are delayed is the time while at least one of the three threads is executing. This time is labeled as the wall-clock delay in the figure. These semantics are consistent with the intuition that time flows in parallel on independent threads; that is, the waiting of two threads on v at once does not result in double-counting the time they are waiting.

Example: Let us return to the application considered in the previous section (see Figure 1). Two annotations are sufficient to implement a power management policy that provides differentiated service depending on the criticality of the health report. The `uploadReport` may be annotated with a `@Wait` (line 4) to delay the transmission up to twenty minutes while waiting for a WiFi connection to be established or for another application to wake the cellular radio. The time-

```
Result: scope S[w] of each @Wait annotation w
Procedure computeScopes()
    methodWorklist = { methods including @Wait annotations }
    while methodWorklist ≠ ∅ do
        remove method m from methodList
        toAnalyze = solve (m, summary (m))
        add methods in toAnalyze to methodWorklist
    foreach w ∈ @Wait do
        let P = set of variables labeled with getLabel(w)
        let V = set of variables annotated with @DelayBudget
        S[w] = ∅
        foreach (p, v) ∈ P × V do
            if pointsTo(p) ∩ pointsTo(v) ≠ ∅ then
                S[w] = S[w] ∪ {v}

Procedure solve(Method $m$, Labels initLabels)
    Let e be the entry node of method m
    if IN[m, e] = initLabels then
        return summary(m)
    IN[m, e] = IN[m, e] ∪ initLabels; worklist = { e }
    toProcess = ∅
    while worklist ≠ ∅ do
        remove n from worklist
        switch n do
            case @Wait: $OUT[m,n] = IN[m,n] \cup \{labelWait()\}$
            case basic-node: $OUT[m,n] = IN[m,n]$
            case merge: $IN[m,n] = \bigcup_{p \in pred(n)} OUT[p]$
            case branch:
                $OUT^T[m,n] = IN[m,n]$ $OUT^F[m,n] = IN[m,n]$
            case call(m'):
                OUT[m,n] = IN[m, n] ∪ summary(m')
                add m' to toProcess
            case return:
                summary (m) = summary (m) ∪ IN[m,n]
                if summary(m) changed then add all methods calling
                m to toProcess
            case Thread.Notify:
                foreach waitSite ∈ WNG.wait(n)) do
                    Let $m'$ be the method including $waitSite$
                    $IN[m', waitSite] = IN[m', waitSite] \cup IN[m,n]$
                    add m' to toProcess
        add all the succ(n) that have updated labels to worklist
    return toProcess
```

Algorithm 1: Computes the default scopes of `@Wait`.

sensitive objects are annotated with `@DelayBudget(0)` at line 7. As the `uploadReport` method contains a possible reference to the health report at line 22, 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.

3.2 Static Analysis of `@Wait` Scopes

Manually determining the scope of a `@Wait` annotation is prone to error since an annotation can affect multiple components and threads. These challenges are further compounded on Android by the asynchronous interactions between the Android framework and the application. To avoid potentially hard-to-debug errors, we developed a novel static analysis that determines the objects that *may be delayed* by a `@Wait`. We use the analysis to automatically determine the scope of all annotations using the default namespace.

The Tempus analysis is implemented on top of the Soot Java Optimization Framework [18] to take advantage of its analysis framework, as well as its built-in algorithms for call-graph generation and points-to analysis. The latter determines the sets of variables that may refer to the same object(s). In contrast to traditional Java applications that have a single entry point, an Android application has multiple entry points that may be invoked by the Android framework at run-time. Tempus uses FlowDroid [8] to build a *dummy main* that models the interactions between the application and the Android framework (e.g., calls to `onCreate`, `onResume`, etc.). In order to ensure accurate results, Tempus jointly analyzes the code of the Android SDK and that of the application.

The pseudocode of the analysis is included as Algorithm 1. At a high level, the analysis works in three steps. First, it identifies the set of variables that are annotated with `@DelayBudget`, which indicates that they are time-sensitive. Next, it determines the potential impact of each `@Wait` by labeling all program points reachable from each `@Wait` with a unique label. The analysis propagates the labels across all reachable procedures and threads. Finally, it determines the scope of each `@Wait` based on the information derived in the first two steps: for each `@Wait`, it extracts all references to annotated variables that were reached by the `@Wait`'s label. Essentially, these variables are live within the reach of the `@Wait`.

The propagation of `@Wait` annotations has an intra-procedural and an inter-procedural part. Both parts are achieved using the standard worklist algorithm that propagates data-flow facts until no new facts are derived.

The inter-procedural analysis propagates labels between procedures in a context-insensitive manner. That is, for each method we create a single *summary* that includes the labels of all `@Wait` annotations that impact that method. When the algorithm determines that the summary includes new labels, all the callers of that method are reevaluated to propagate these labels (see line 2).

The intra-procedural analysis for a method m works on its control-flow graph (CFG). The CFG has a distinguishable entry point, merge nodes with two or more predecessors, branch nodes with a *true* successor and an optional $false$ successor, and basic nodes with a single successor and predecessor. The call and return nodes are used to indicate method calls and associated return sites. Successor and predecessor functions (edges) are appropriately defined on the CFG. During the construction process we ensure that at most one Tempus annotation is included in any basic block. The notation $IN[m, n]$ and $OUT[m, n]$ refers to labels available immediately before and after a node n in method m. The superscripts T and F are added in the case of a branch node to indicate the *true* and $false$ branches, respectively.

The core of the intra-procedural analysis is the case statement found in the `solve` procedure. A label for a `@Wait` is generated when the algorithm encounters the annotation (line 23). This label is propagated through basic nodes (line 24), on both branches of branch nodes (line 26), and combined at merge nodes (line 25) using the union operator. An invocation to method m' (line 28) is handled by adding all the labels that impact and are stored in the method's summary. The summary of the method under consideration is updated include all the labels derived to impact the method when a return node is considered (line 31).

Note that `solve` also does much of the heavy lifting of the inter-procedural analysis, both updating the summaries of methods and building up a local worklist of methods (`toProcess`) that need to be added to the inter-procedural worklist maintained by `computeScope`. (Procedure `computeScope`'s contribution to the inter-procedural analysis is to pull the pending methods off the worklist and invoke `solve` on them.)

One special case is that a `@Wait` annotation can also impact other threads of execution when it delays the delivery of a notify event i.e., when a call to `Thread.notify` is delayed. Accordingly, when a `Thread.notify` instruction is encountered during analysis at a program point n, the labels in the $IN[n, m]$ are propagated to all `Thread.wait` sites that may receive the notify event. These sites may be identified using a points-to analysis since a notify event generated by `Thread.notify` is received at a `Thread.wait` only if the two methods operate on objects that may alias. The methods where the `Thread.wait` calls originated are added to the set of methods to be analyzed in order to continue the propagation of labels.

To implement this analysis efficiently, prior to running the analysis, we construct a *Wait-Notify Graph* (WNG), effectively extending the call graph. The graph has the `Thread.wait` and `Thread.notify` call sites as vertices. A directed edge is added from a `Thread.notify` to a `Thread.wait` in the case the two methods are invoked on the same object, which may be determined using the point-to analysis.

Although our analysis aims to be sound, its implementation does have some limitations. These could be addressed through extensions to the approach described here, but were not because they are special cases that were not encountered in our case studies. For one, our analysis is not capable of analyzing either dynamically loaded classes or native code invoked through the JNI. A common way of handling these is to provide Java stubs that conservatively approximate their effects. Also, our analysis's understanding of concurrency is currently limited to the `Thread.wait`/`Thread.notify` constructs. None of these issues hindered the analysis of any of the applications we considered.

3.3 Tempus Service

The Tempus service is responsible for tracking budgets and executing power management policies. The key abstraction provided by the Tempus runtime is `TempusLock` (which implements the `@Wait` annotation), integrating concurrency control and power management. A thread calling the `wait` method of a `TempusLock` will block until the device enters the power-saving state of the lock or the budget of one of the variables in the lock's scope reaches zero. In contrast to standard locks that provide mutual exclusion, a `TempusLock` allows multiple threads to enter the lock and guarantees that a thread will block only if all the objects in its scope have budgets greater than zero.

During code generation a `TempusLock` replaces each `@Wait`. The scope of a `TempusLock` is the same as the scope of the `@Wait` annotation that was determined during static analysis or explicitly declared in the `@Wait`. Tempus maintains a mapping (λ) from each object in the lock's scope to a `Tracker`. Each `@DelayBudget`(b, v) annotation is translated to create a new `Tracker` with budget b for the object referenced by v when there is no tracker for it in λ. Otherwise, the budget of the existing `Tracker` is updated to b. A `TempusLock` has three states `UNUSED`, `WAITING`, and `POWER`. The `UNUSED` state indicates that no thread is in the lock. The `WAITING` state indicates that at least one thread is using the lock, waiting for either the delay budget

```
Data: Map<Variable, Tracker> $\lambda$
Class Tracker
    Data: WeakRef object
          long budget = $+\infty$
          long countLocks = 0, useStart
          Set<TempusLock> sleeping
    public entry(TempusLock lock)
        if countLocks = 0 then useStart = now()
        countLocks = countLocks + 1
        sleeping.add(lock)
    public exit(TempusLock lock)
        countLocks = countLocks - 1
        inUseBudget = now() - useStart
        if countLocks = 0 then  budget = budget - inUseBudget
        sleeping.remove(lock)
    public budget()
        if countLocks = 0 then return budget
        inUseBudget = now() - useStart
        return budget - inUseBudget
    public setBudget(long newBudget)
        currBudget = budget()
        budget = min(newBudget, maxBudget)
        if newBudget < currBudget and countLocks > 0 then
            foreach lock in sleeping do  lock.notify()

Class TempusLock
    Data: PowerExpression expr, long maxLockBudget
          List<Variable> scope
          state : {UNUSED, WAITING, POWER }
          Lock lock
          int countThreads = 0
    public wait()
        state = WAITING
        countThreads = countThreads + 1
        foreach tracker in $\lambda$ do  tracker.entry (this, entryTime)
        minBudget = min(budget(), maxLockBudget)
        if minBudget > 0 then
            hwMonitor.register(this, expr)
            while state = WAITING do
                lock.wait(minBudget)
                if state = WAITING then
                    // Recompute budget to reflect a smaller
                       budget
                    minBudget = budget()
            hwMonitor.unregister(this, expr)
        countThreads = countThreads - 1
        if countThreads = 0 then state = UNUSED
        foreach tracker in $\lambda$ do  tracker.exit (this)
    public budget()
        return $\min_{tracker \in \lambda}(tracker.\text{budget}())$
```

Algorithm 2: Formalization of `TempusLock` and `Tracker` maintained by the run-time service.

of a tracked object to become zero, for the device to enter the desired power state, or for the annotation to reach its timeout. The `POWER` state indicates the device has entered the desired power state.

Tracking budgets in the case when a single thread may enter the `TempusLock` works along the following lines. Upon entering the lock, the state of the `TempusLock` is changed to `WAITING`. The time the thread may spend within the lock is determined by computing the minimum budget among all `Trackers`. If the minimum budget *minBudget* is greater than zero, then the thread will block for at most the *minBudget* and the maximum delay specified as the `UpTo` argument in the `@Wait`. The run-time concurrently evaluates the power state expression based on asynchronous events regarding the changing hardware states of the device. If the power expression of the lock evaluates to true, the state will be changed to `POWER` and notify signal will be generated to wakeup the thread. Upon exiting the lock, the time spent while blocking in the lock is subtracted from all the budgets of the references in the scope of the lock. Before exiting the lock, the state is changed to `UNUSED`.

Handling concurrency requires careful accounting of the budgets in the implementation of `TempusLock` (see Algorithm 2). The budgets of the `Trackers` associated with a `TempusLock` must (in effect) be continuously decremented while there is at least a `TempusLock` using the `Tracker`. The `entry`, `exit`, and `budget` methods of a `Tracker` are used to manage its budget. Threads entering and leaving a `TempusLock` will invoke the `entry` and `exit` of all `Trackers` associated with that lock. The `budget` returns the remaining budget of the `Tracker` and is updated as follows. We say that a `Tracker` is *not-used*, when all locks that have the `Tracker` in its scope are in the `UNUSED` state (i.e., $countLocks = 0$). Conversely, the `Tracker` is *used*, when at least a lock is in the `WAITING` state (i.e., $countLocks > 0$). When the tracker is *not-used*, the value of its remaining budget is available in the *budget* variable. When the tracker is *used*, we first compute the budget spent for the current *used* period as the difference between the current time and start of the *used* period (identified by the first call to `entry`). Then, the remaining budget is simply the difference between the budget consumed before the current *in-use* period (stored in *budget*) and the budget consumed during the current *in-use* period (stored in $inUseBudget$). The *budget* variable is updated at the end of the *in-use* period, when all locks that have the variable in the scope are in the `UNUSED` state (i.e., $countLocks = 0$).

Another interesting case occurs when the budget of a `Tracker` is set to a value smaller than the current budget while a `TempusLock` is in the `WAITING` state and has the `Tracker` in its scope. This case is handled in the `setBudget` method of a `Tracker` that wakes all threads currently using the `Tracker` to recompute the time it is safe to sleep without violating the Tempus guarantees.

Power Management: The runtime service monitors the hardware resources of the phone in a separate thread. Each `TempusLock` registers itself with its power state expression, which is managed by the Tempus runtime. When changes in device state cause the expression to evaluate to true, the Tempus runtime notifies the `TempusLock` by triggering a state transition to `POWER`. Tempus runtime monitors the state of hardware 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 callbacks (events) from the Android framework. The monitoring of a hardware resource is suspended when no `TempusLocks` require it.

Garbage Collection: To ensure that object 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 *weak references*. The garbage collector does not count these references as true references. When a tracked object is garbage collected, its entry is automatically dropped from the mapping.

A related issue is that infrequent garbage collection may result in a `Tracker` staying in our mappings after its associated object is no longer reachable by the application. Such an object will still be included in budget tracking, perhaps causing Tempus to calculate an artificially low available budget for a `TempusLock`. For an application which this is an issue, there is an extra optional parameter to the `@Wait` an-

```
1  class NewView extends View {
2   public void prepareNewsList(List<NewsItems> NewsList) {
3   ...
4     for(int i = 0; i < NewsList.size(); i++) {
5      NewsItems newsitem = NewsList.get(i);
6      String imageUrl = newsitem.getUrl();
7      prefetchHandler.postDelayed(new DelayedDownload(imageUrl),
            i * PREFETCH_PERIOD);
8      }
9   ...
10  }
11
12  public View getView(int position, View convertView, ViewGroup
         parent) {
13    ...
14    NewsItem newsitem = NewsList.get(position);
15    String imageUrl = newsitem.getUrl();
16    @DelayBudget(0, imageUrl)
17    ...
18    DownloadUtils.downloadBitmap(imageUrl);
19    ...
20  }
21 }
22
23 class DownloadUtils {
24  private static Bitmap downloadBitmap(String imageUrl) {
25    @Wait(UpTo="10min", For="Network.Active or WiFi.Connected")
26    URL url = new URL(imageUrl);
27    URLConnection conn = url.openConnection();
28    conn.connect();
29    @ClearBudget(imageUrl)
30    ...
31  }
32 }
33
34 class DelayedDownload implements Runnable {
35    String imageUrl;
36    public DelayedDownload(String url) {
37      imageUrl=url;
38    }
39    public void run() {
40      @DelayBudget(100, imageUrl)
41      DownloadUtils.downloadBitmap(imageUrl);
42    }
43 }
```

Figure 3: The download of images in the NPR News is deferred until the radio or WiFi are on to save energy. Additionally, the download of the in-view images is not delayed, while an image prefetch request is delayed up to 100 seconds.

notation, `AggressiveGC=True`, that forces the `@Wait` to call the garbage collector, thus mitigating this issue at the cost of an additional garbage collection.

4. CASE STUDY

In this section we present a case study of introducing power management policies into two delay tolerant applications: NPR News [6] and CitiSense [12]. 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.

4.1 NPR News

NPR News provides the user with a scrolling 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 news stories, some content, and references to images that are used as thumbnails in the news list. The original implementation of the application downloads images in an *on demand* 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 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 pop-in effect found in the on demand version of the application. We will show in Section 5 that given opportunities to piggyback network transmissions with requests from other applications on the device, it is possible to prefetch 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.

Figure 3 shows the relevant code snippets from the NPR app. To improve its energy efficiency, we annotated the `downloadBimap` method that is responsible for downloading images with a `@Wait` (line 25). The `@Wait` can defer downloads until either the cellular radio or the WiFi is turned on. The timeliness of the application is controlled by annotating the `String` objects containing the image URLs with two `@DelayBudget` annotations. We ensure that images that are already in view are downloaded immediately by setting their budget to zero (line 16). Additionally, we limit the time any download may be delayed to be 100 seconds (line 40). At compile time, the static program analysis infers that the `imageUrl` 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. A `@ClearBudget` is used to stop tracking the budget of `imageUrl` after line 29. The annotation is necessary to demarcate the time-critical part of the `imageUrl` life-cycle that occurs in lines 25 – 29. We note that the `imageUrl` is never garbage collected since it is referenced as part of the XML list of news stories that is always live.

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

4.2 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 [12].

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. This policy saves considerable energy at the expense of timeliness. In order to enable CitiSense to effectively manage the trade-off between energy and timeliness, we updated this application using Tempus. The general structure of the CitiSense code including Tempus is shown in Figure 4.

The primary objective of our changes is 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

```
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 Storage implements Runnable {
17   protected final Uploader uploader;
18   protected final BlockingQueue<Message> toSave;
19
20   public void run() {
21     while(true) {
22        final Message m = toSave.take();
23        ... save message to flash ...
24        uploader.addMessage(m);
25      }
26   }
27   public void addMessage(Message m) {
28     toSave.put(m);
29   }
30 }
31
32 class ReadSensors implements Runnable {
33   protected final Handler handler;
34   protected final Storage storage;
35
36   public void run() {
37     while (true) {
38       @Wait(UpTo="10min", For="Location.Change",
39             Scope="DisplayReading")
40       final SensorReading reading = Sensor.read();
41
42       if (reading.getValue() > EXPOSURE_THRESHOLD) {
43         @Namespace(reading,"UrgentReading")
44         @DelayBudget("0Min", reading)
45       } else {
46         @Namespace(reading,"NormalReading")
47         @DelayBudget("10min", reading)
48       }
49       storage.addMessage(reading);
50       handler.sendMessage(reading);
51     }
52   }
53 }
```

Figure 4: We improve the original CitiSense in three ways: (1) The energy consumption is reduce by deferring the upload of sensor data until another application turns on the radio. (2) We reduce the average delay encountered in the upload of high pollution levels. (3) We save additional energy by deferring data acquisition until the user moves.

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]. Providing differentiated timeliness is also achieved by defining the `NormalReading` and `UrgentReading` namespace (lines 46 and 43). Delay budgets of 10 and 0 minutes are assigned with the objects belonging to the `NormalReading` and `UrgentReading` namespaces, respectively (lines 47 and 44). 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 objective of our changes is to optimize the energy consumed by networking. Specifically, energy saving may be achieved by deferring the upload of sensor data until another application turns on the radio. Therefore, 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.

The last objective of our changes is also to save energy through taking advantage of the fact that pollution values do not change fast at the same location (see line 38). 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. This can be achieved by enabling the application to make use of the GPS to generate precise location information, while a user is determined to be moving (using a combination of accelerometer data and wireless network based localization). In this regard, the application was updated to `@Wait` up to one minute for the user's location to change before sampling the sensor, in order to reduce the cost of Bluetooth communication during stationary periods.

This example demonstrates the compositional properties of Tempus. The two power management policies operate in isolation. The developer can verify that this is the case by checking 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 global invariant provided by Tempus: an operation may never be delayed more than the budget of any object in its scope.

5. EXPERIMENTS

In this section, we demonstrate the efficacy of Tempus in the two real-world applications discussed in the previous section. Since Tempus is a tool for introducing power management policies, it is not our goal to study all possible power management policies. Instead, we will focus on showing Tempus's ability to specify and implement different energy-delay trade-offs, measure the overhead of the runtime, and evaluate the correctness of the implementation.

To this end, we implement different Tempus power management policies for the NPR News and CitiSense. Both applications were modified to play back real-world measurements obtained from users of NPR News and CitiSense. This is essential for a consistent evaluation since the timing and energy consumption of the applications is highly depended on the input. NPR News was modified to log how five different users made use of the application over the course of a one week period. These logs were then 'played back' during experiments to control the rate at which stories were scrolled through in the application and which stories were selected for further reading. In the case of CitiSense, five days worth of pollution and mobility data (five different users, one day each) from previous studies of the CitiSense project were replayed through the application. During the study, users were asked to charge their devices frequently to ensure 24-hour operation of the application and sensor device. This data set includes both mobile and stationary measurements, as well as a wide range of Air Quality Index scores, all of which can have a significant impact on the behavior of the power management policy found in CitiSense. A detailed description of the traces may be found in [12].

The experiments were performed on a Pantech Burst smartphone with Android 4.0.4. Power consumption was measured using a Power Monitor from Monsoon Solutions [5]. To integrate with the Power Monitor, we 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.

5.1 NPR News

We implemented four different power management polices for the NPR application that provide different energy-delay trade-offs: `Ondemand`, `Light-`, `Medium-`, and `Full-`prefetch. The `Ondemand` policy downloads images when needed to populate the in-view UI elements. The `Light` and `Medium` prefetch policies download batches of three and six images every two minutes since a user is using the application. The `Full` prefetch policy fetches all images when the application is started.

The `Light` and `Medium` policies are based on the observation that a user is more likely to view stories and images further down the list the longer they interact 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 impacts on the trade-off between power consumption and timeliness.

The application usage data was collected from five users during the course of a week. The trace logged the start of the application, reading through the story list, selecting the stories to be read, and closing the application. The traces ware played back through the application. During an experiment, we recorded the total amount of data used and time required for an image to appear in the UI for each "use" of the application. Each experiment was run five times for each policy. Error bars in each figure represent one standard deviation. In the following we will discuss the trade-off between energy saving and timeliness of each policy when piggybacking opportunities are available every three and one minute.

Figure 5 presents the average system power consumption, user-observed delay to view an image, and data usage for each policy when piggyback opportunities are available 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 suffered the worst delay: on average, a user had to wait 416 ms for an image to appear in the application. In contrast, the `Full` 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 also increase power consumption.

Figure 6 presents results for each of the four policies when piggyback opportunities are 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 an idle radio. Second, 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 it has a 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.

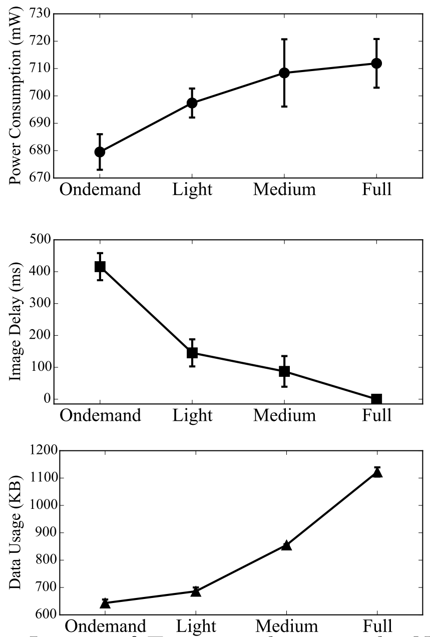

Figure 5: Impact of Tempus policies on the NPR News when piggybacking opportunities are available every three minutes.

These experiments indicate that Tempus is effective in managing the trade-off between energy savings and timeliness. The four policies provide a wide range of trade-offs between delay, energy, and even data usage.

Figure 7 presents the state of the budgets associated with nine images during a run of the NPR application with the `Light` policy. The URLs to the first three images to be prefetched are initially assigned a budget of 100 and placed into the 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

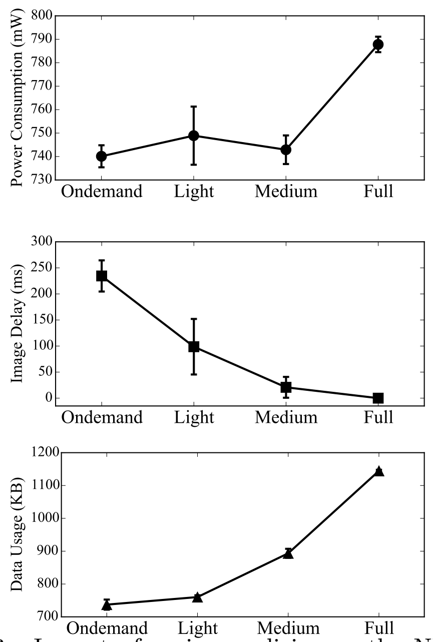

Figure 6: Impact of various policies on the NPR News when piggybacking opportunities are available every minute.

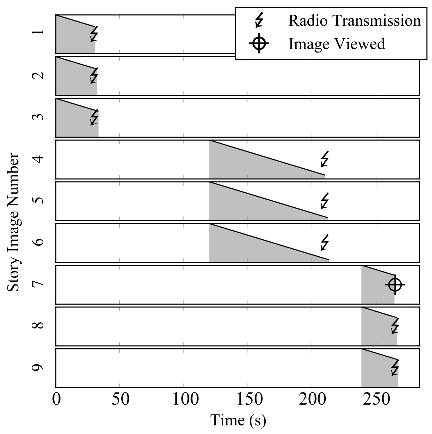

Figure 7: The budgets assigned to various images during a run of the NPR News application with the `Light` policy.

element containing that image is viewed by the user and assigned a new budget of zero, triggering its download. The eighth and ninth images, though not in view, are downloaded as well, as the radio was powered to fetch the seventh image just moments before. We found that such visualizations are effective in understanding the impact of the power annotations and can be easily generated using our tools.

5.2 CitiSense

CitiSense originally implemented a power management policy (denoted as `Max Savings`) that emphasized energy savings at the cost of timeliness. `Max Savings` batches air quality readings for 10 minutes and uploads the complete batch to a remote server, regardless of its content. We have modified CitiSense to implement the power management policies described in Section 4.2 (denoted as `Min Delay`). `Min Delay` ensures that poor air quality readings are uploaded to the server as soon as possible and that the Bluetooth sensor is sampled less frequently when the user is stationary. 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. In the following, we will analyze the performance of the two policies when the users are mobile and stationary.

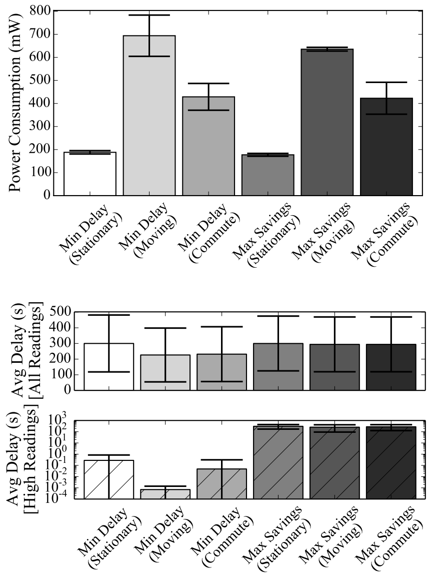

Figure 8: Impact of policy and mobility on average system power consumption (top) and delay experienced by sensor readings before reaching the server (bottom) in CitiSense.

Figure 8 shows the impact of the two policies and the presence of mobility on the power consumption and upload latency in CitiSense. The performance of the application is highly dependent on whether the sensor readings were taken while the user was stationary or mobile. Additionally, we provide statistics regarding the performance of the application when the user commutes to their workplace (labeled as `Commute`). The commute is approximately 43 minutes each day and includes both measurements taken while moving (e.g., walking, biking, and driving) and while stationary (e.g., stopped at intersection and stuck in traffic).

The lowest power-consumption occurred while users were stationary, as network-based localization is relatively energy-efficient. While the `Min Delay` policy samples the Bluetooth sensor less frequently than the `Max Savings` policy, it does occasionally wake the radio to transmit high readings and, therefore, had a slightly higher average power consumption (188 mW vs. 177 mW) during our experiments. The highest power-consumption occurred when users were mobile, as GPS based localization is power-hungry. Most stationary readings were in indoor environments, where air quality is typically good, while most mobile measurements were taken outdoors, where proximity to major roadways can have a significant impact on air quality. These high readings were more prevalent in the mobile measurements, further increasing the gap in power consumption between `Min Delay` (694 mW) and `Max Savings` (635 mw). Power consumption during commutes was slightly higher for the `Min Delay` policy (435 mW vs. 423 mW).

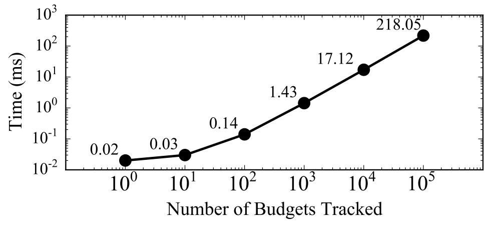

Figure 9: Overhead associated with the tracking of various numbers of budgets at runtime. Note the logarithmic scale.

The choice of policy had a significant impact on time to upload the readings to the server for further processing. The `Max Savings` and `Min Delay` offer the developers different trade-offs between energy and timeliness. As expected, `Max Savings` savings an additional 9% and 3% when users are mobile and commuting. Comparatively, `Min Delay` reduces average delay of delivering all the data by 23% and 21% when users are mobile and commuting, respectively. More importantly, `Min Delay` reduces the average latency of delivering high readings from 275 s to 0.05 s during commute. We note that both policies may be specified using only three annotations showing the range of trade-offs that could be achieved using Tempus.

5.3 Budget Tracking Overhead

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 objects are affected by `@Wait` annotations, the only significant overhead introduced by Tempus at runtime is that of tracking budgeted objects.

Figure 9 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 app, budgets were tracked for each of the approximately twenty news stories presented to the user at a time.

6. RELATED WORK

Research in energy-efficient software typically falls into one of two categories. Low-level optimizations are typically implemented at the kernel or device-driver level and manage the power state of hardware components. Examples of such techniques include dynamic voltage and frequency scaling (see [19] for a review), tickless kernel implementations [17], low-power listening [13], and batching of I/O operations for devices such as flash [20]. Such optimizations are typically the responsibility of device vendors. System-level optimizations, on the other hand, are implemented at the application- or middleware-level. These optimizations interact with hardware components at longer time-scales and include techniques such as workload shaping, sensor fusion, and filtering. A workload shaping policy like delaying large network operations until an available WiFi connection is found in applications such as Google Play Market, Facebook, and Dropbox.

The difficulties of implementing such policies motivated our work, as well as that of others, in developing higher-level abstractions for power-management. Energy Types introduces a type system to the specify phased behavior and energy-dependent operating modes of an applications. The types information to dynamically adjust CPU frequency and application fidelity at runtime to save energy [9]. EnerJ employs a type system to specify which data values in their application may be approximated to save energy and guarantees the isolation of precise and approximate components [16]. However, neither Energy Types nor EnerJ aim at managing the tradeoff between energy and timeliness. Moreover, these techniques are best-suited for CPU-intensive applications and they have limited applicability to applications that make heavy use of other power-hungry resources.

Procrastinator is a system that automatically delays the prefetching of network resources to reduce network data usage and energy consumption [15]. Procrastinator identifies common code pattern in applications and automatically modifies the relevant network calls to delay fetching the content of a user interface element until it becomes visible. This work is related to our own in that it eases the implementation of a delay-based technique within an application, but it is narrower in scope and less flexible. Procrastinator focuses only on networking operations, and only those related to the user interface. It also provides no developer control of user experience: all identified prefetching calls are delayed. On the other hand, these limitations allow the approach to be completely automated, with no developer input. In this respect, Procrastinator and Tempus are at opposing ends of the spectrum of tool-assisted delay-based power management techniques.

In previous work, we introduced APE [11], 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. However, similar to Procrastinator, APE does not address the trade-off between energy saving and timeliness. APE makes it easy to achieve the `Max Savings` policy for CitiSense described in the previous two sections, but provides no mechanism for achieving the `Min Delay` policy. The only way to achieve such a policy with APE is to deftly place *multiple* APE wait annotations upstream on the various paths to a power hungry operation (including across threads), each getting a different timeout. This can require restructuring the application with new paths, and necessarily places the many annotations distant from the power-hungry operation, hurting understandability of the code and complicating maintenance. These complications are a manifestation of what the examples in the paper demonstrate, that reasoning about the impact of power policies on user experience in a path-centric manner is extremely demanding, at best. In contrast, Tempus's fundamentally different data-centric approach dramatically simplifies reasoning about the impact of power policies on timeliness. Additionally, Tempus includes a novel static analysis and run-time abstraction to support this new paradigm of reasoning about power management.

The trade-off between energy and delay in smartphone applications was also studied in [14]. The authors propose an online algorithm for dynamically selecting the best wireless interface to be used for packet transmissions. Currently, Tempus includes a simple but effective mechanisms of waiting for a low-power state before performing a power transmission. In the future we plan to explore other venues of determining the most energy-efficient timing for performing power-hungry operations.

7. CONCLUSION

CRM 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. Tempus is a novel approach for writing power management policies that effectively controls the trade-off between energy saving and delays. 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 timeliness is managed by associating delay budgets with objects that contain time-sensitive data. We use static analysis to help the programmer determine the impact of introducing power-related delays in an application, which may lead to difficult-to-debug errors due to concurrency. The main abstraction provided by the run-time service is that of a `TempusLock` that implements power management and concurrency control. In combination, the static analysis and the run-time service ensure that power management policies will not delay an object more than its assigned budget.

We showed that our approach is both expressive and flexible by introducing power management into two realistic applications. Tempus is able to provide differentiated delays for different operations in mobile applications. Detailed experiments from the studied applications show that Tempus may effectively control the energy-delay trade-off. A few annotations can profoundly impact the performance of an application. In the case of NPR News, five Tempus annotations may implement a policy that reduces the latency of downloading images by 10 times compared to the original without affecting energy consumption. Similarly, in CitiSense, three Tempus annotations may implement a policy that reduces the latency of uploading the low quality air samples from 275 s to 0.05 s compared to the original implementation, albeit at the cost of a 3% – 9% increase in energy consumption. Our experiments also indicate that Tempus introduces a relatively small overhead to track the budgets of time-sensitive objects. In practice, the number of time-sensitive objects is relatively small.

Acknowledgements

The authors would like to thank the anonymous reviewers for their feedback and Sorin Learner for his guidance on static analysis. This work was supported by NSF grants CNS-0932403, CNS-1144757, CNS-1446912, and CNS-1144664. Additional funding was provided by Roy J. Carver Charitable Trust (Grant No. 14-4355).

8. REFERENCES

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

[2] Bittorrentsync. `http://www.bittorrent.com/sync`.

[3] Dropbox. `https://www.dropbox.com/`.

[4] Fitbit. `http://www.fitbit.com/android`.

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

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

[7] ohmage. `http://ohmage.org/`.

[8] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of PLDI*, page 29. ACM, 2014.

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

[10] M. Lin, N. D. Lane, M. Mohammod, X. Yang, H. Lu, G. Cardone, S. Ali, A. Doryab, E. Berke, A. T. Campbell, et al. Bewell+: multi-dimensional wellbeing monitoring with community-guided user feedback and energy optimization. In *Proceedings of WH*, 2012.

[11] N. Nikzad, O. Chipara, and W. G. Griswold. APE: an annotation language and middleware for energy-efficient mobile application development. In *Proceedings of ICSE*, 2014.

[12] N. Nikzad, N. Verma, C. Ziftci, E. Bales, N. Quick, P. Zappi, K. Patrick, S. Dasgupta, I. Krueger, T. v. Rosing, and W. G. Griswold. Citisense: Improving geospatial environmental assessment of air quality using a wireless personal exposure monitoring system. In *Proceedings of WH*, 2012.

[13] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of SenSys*, 2004.

[14] M.-R. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely. Energy-delay tradeoffs in smartphone applications. In *Proceedings of MobiSys*.

[15] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. Procrastinator: Pacing mobile apps' usage of the network. In *Proceedings of MobiSys*, 2014.

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

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

[18] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a Java bytecode optimization framework. In *Proceedings of CASCON*, 1999.

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

[20] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li. Optimizing background email sync on smartphones. In *Proceeding of MobiSys*, 2013.