Stochastic Energy Optimization for Mobile GPS Applications

Anthony Canino
SUNY Binghamton
Binghamton, New York
acanino1@binghamton.edu

Yu David Liu
SUNY Binghamton
Binghamton, New York
davidl@binghamton.edu

Hidehiko Masuhara
Tokyo Institue of Technology
Tokyo, Japan
masuhara@is.titech.ac.jp

ABSTRACT

Mobile applications regularly interact with their noisy and ever-changing physical environment. The fundamentally *uncertain* nature of such interactions leads to significant challenges in *energy optimization*, a crucial goal of software engineering on mobile devices. This paper presents Aeneas, a novel energy optimization framework for Android in the presence of uncertainty. Aeneas provides a minimalistic programming model where acceptable program behavioral settings are abstracted as *knobs* and application-specific optimization goals — such as meeting an energy budget — are crystallized as *rewards*, both of which are directly programmable. At its heart, Aeneas is endowed with a stochastic optimizer to adaptively and intelligently select the reward-optimal knob setting through a form of reinforcement learning. We evaluate Aeneas on mobile GPS applications built over Google LocationService API. Through an in-field case study that covers approximately 6500 miles and 150 hours of driving as well as 20 hours of biking and hiking, we find that Aeneas can effectively and resiliently meet programmer-specified energy budgets in uncertain physical environments where individual GPS readings undergo significant fluctuation. Compared with non-stochastic approaches such as profile-guided optimization, Aeneas produces significantly more stable results across runs.

CCS CONCEPTS

• **Software and its engineering → Power management**; • **Human-centered computing → Mobile computing**;

KEYWORDS

Energy Management, Stochastic Optimization, Uncertainty

ACM Reference Format:
Anthony Canino, Yu David Liu, and Hidehiko Masuhara. 2018. Stochastic Energy Optimization for Mobile GPS Applications. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages.
https://doi.org/10.1145/3236024.3236076

1 INTRODUCTION

Energy efficiency is a critical goal of mobile software ecosystem design. Effective energy optimization requires a precise understanding of the *in-situ* energy behavior of the application and its underlying systems. The very nature of *mobile* systems, however, poses a fundamental challenge against meeting this requirement: a mobile application may frequently interact with the noisy and ever-changing physical environment, leading to significant *uncertainty* in energy behavior.

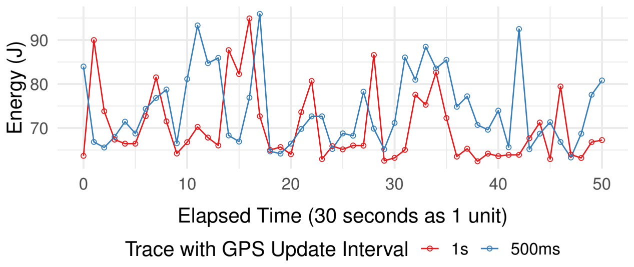


Figure 1: The Energy Fluctuation of a GPS Application. The two traces are collected when a mapping app, `MAPS.ME`, navigates in a moving vehicle. Each data point represents the energy consumption in the preceding time unit.

Take the energy behavior of a GPS-based mapping application for example. Figure 1 illustrates two traces of its energy consumption, with GPS update intervals set at 1 second and 500 milliseconds respectively through Google LocationServices [2]. Some observations may surprise an energy optimization designer who expects more "predictable" results. First, within each trace, there is significant variation in energy consumption — *e.g.*, ranging from 65 joules to 95 joules in the 1s trace — despite that the GPS update interval does not change within each trace. Second, the variation is often time-dependent — *e.g.*, for the 1s trace, energy consumption ranges 65-90 joules during elapsed time units 15-20 while later it remains within 65-70 joules during elapsed time units 35-45 — possibly due to physical environment change. Third, when we consider multiple traces, the seemingly "obvious" task of deciding between 1s and 500ms as the desirable GPS interval to minimize energy consumption — a common decision in application-level energy optimization —may not be obvious at all.

Aeneas Design. In this paper, we describe Aeneas[1], a novel energy optimization framework for Android applications where *uncertainty is treated as the rule — not the exception — in mobile application energy optimization.*

Aeneas provides a simple programming interface where *knobs*, *i.e.*, application-specific semantic elements that may have different

[1]Aeneas is the hero in Virgil's epic *Aeneid*. Despite many uncertainties on land and at sea, he finally reached Latium.

energy impacts, can be defined by programmers. In addition, programmers may provide their application-specific goal for optimization, such as Quality-of-Service (QoS) or energy budget, through programmable *rewards*. The maintenance tasks of the knobs and rewards are left to the *stochastic* run-time optimizer, such as deciding which knob setting should be selected, when it should be selected, and how to meet the reward. Unlike traditional approaches based on testing or profiling, our optimization is fundamentally *in vivo*: the process of observing the uncertain vicissitudes of energy behavior and that of performing stochastic optimization are unified as one. Within the landscape of energy optimizations where uncertainty is either "smoothed out" through heuristics, or mitigated through fault tolerance, or worse, treated as an experimental afterthought, Aeneas represents a distinct point in the design space where uncertainty is captured *by design*.

Philosophically, Aeneas can be viewed as an interesting instance of human-machine co-optimization. The two components of Aeneas — a simple programming interface and a sophisticated runtime design — bring the programmer and the automated algorithm together for cooperative energy optimization. This two-pronged design leads to three noteworthy features representing different work distributions between the programmer and the automated algorithm. First, we allow programmers to fully provide alternative settings of a knob — such as a GPS update interval can be either 1s or 500ms — a feature we call the *declared discrete knob*. The goal of the stochastic optimizer for this feature is to adaptively select the reward-optimal one among the alternatives. Second, when the programmer does not fully specify what the alternatives for a numeric knob should be, we allow programmers to simply provide a range, a feature we call the *inferred knob*. To support this feature, the stochastic optimizer is endowed with an *iterative refinement* algorithm. Third, our stochastic algorithm itself comes with inherent parameters. Instead of asking programmers or deployers to provide hard-code values, Aeneas can function as a *self-optimizer*, where the optimal parameter setting is stochastically selected.

Our concrete stochastic algorithm design is based on Multi-Armed Bandits (MAB), a family of reinforcement learning algorithms with solid theoretical underpinnings on optimality [16, 18, 28]. Our framework bridges program-level elements of knobs and rewards to two fundamental concepts in MAB: slots and rewards. In this view, Aeneas is a unique systematic study in streamlining the programming interface between the application and the runtime for reinforcement learning. In Aeneas, stochastic optimization is no longer a pure system-level feature invisible to programmers, but a "service" that applications may choose to resort to for cooperative energy optimization.

Aeneas for GPS Applications. Our experience with Aeneas represents a significant effort of evaluating the idea of stochastic energy optimization in the real world. Aeneas is built on top of Android, and we use it to guide an open-source GPS mapping app, `MAPS.ME`. The evaluation was conducted through driving for 150 hours covering approximately 6500 miles, as well as 20 hours of biking and hiking. To the best of our knowledge, our in-field evaluation is the most extensive study of applying MAB algorithms to mobile/transportation systems. These use scenarios are impactful in the real world for a number of reasons. First, GPS apps are prevalently used. Second, they continuously interact with the surrounding environment regardless of being in the foreground or background; the need for turn-by-turn directions implies they may be used for a long duration. Third, for use scenarios where driving is involved, plugging mobile devices to car chargers is known to be environmentally unfriendly—it may cost 33 times more energy than charging from wall sockets [1]. Fourth, for more environmentally friendly use scenarios such as biking, hiking, and riding in public transportation, the energy optimization of GPS applications critically links to mobile device usability.

Our evaluation shows that Aeneas is highly effective in energy optimization in the presence of uncertainty inherent in GPS data. Through a comprehensive design space exploration — with declared discrete knobs *vs.* inferred knobs, varying energy budgets as rewards, varying numbers and settings of knobs, and varying modes of transportation — we find Aeneas consistently converges toward the knob settings to maximize the programmer-specified reward, such as a particular battery drain rate. When compared with non-stochastic approaches, we also find Aeneas remarkably *resilient*: occasional noise or physical environment change has little impact on the effectiveness of optimization. In contrast, profiling-based approaches are significantly more sensitive to how representative the profile data are.

Contributions. This paper makes the following contributions:

- A simple and intuitive programming model and API to streamline application-level energy management with run-time level stochastic optimization
- A novel stochastic optimizer based on MAB with 3 variants: an optimizer for declared discrete knobs, an iterative refinement optimizer for inferred knobs, and a self optimizer
- A significant in-field evaluation with a comprehensive design space exploration and extensive driving/biking/hiking experiments

Aeneas is an open-source project. Details can be found at our website with URL `https://github.com/pl-aeneas/aeneas`.

2 BACKGROUND

MAB is a classic problem in stochastic optimization. The real-world inspiration comes from slot machines. A gambler wishes to optimize the accumulated reward by pulling a set of slot machines, each with a distinct but unknown distribution of rewards. The gambler may choose to *interact* with the slot machine by either pulling a lever to "try it out," a step known as *exploration* in MAB, or pulling one known to produce a good reward, a step known as *exploitation* in MAB. Most MAB solutions focus on balancing the trade-off between exploration and exploitation, a common theme within reinforcement learning.

Mathematically, the MAB problem can be viewed as follows. Given a set of configurations C, let us assume each configuration $c \in C$ when interacted at its t^{th} time has reward $\rho_t(c)$. The MAB algorithm attempts to construct an *interaction sequence* $I = [i_1,\ i_2\ \ldots\ i_q]$ where $i_j \in C$ for any j, with the goal of maximizing $\sum_{k=1}^{q} \rho_{\mathcal{L}([i_1,\ldots i_k], i_k)}(i_k)$ where $\mathcal{L}(I, c)$ is a *local counting* function defined as the number of times c appears in I. The algorithm is online in nature, in that the selection of i_j may be influenced by the rewards associated with configuration choices $i_1 \ \ldots \ i_{j-1}$.

Many solutions exist for solving the MAB problem [17]. For example, the one algorithm that will be extensively used by Aeneas is Value-Difference Based Exploration (VDBE) [32]. In the interaction sequence, the *average* reward for each configuration c is maintained separately through the simple Q function below.

$$Q_t(c) = \frac{1}{t}(\rho_1(c) + \rho_2(c) + \cdots + \rho_t(c))$$

As the algorithm is online in nature, the same definition can be written a more friendly recurrence form:

$$Q_{t+1}(c) = Q_t(c) + \frac{1}{t+1}(\rho_{t+1}(c) - Q_t(c))$$

The essence of VDBE is to choose between exploration and exploitation at each interaction. For an interaction sequence so far as $I = [i_1, \ldots, i_{t-1}]$, the next interaction should chose i_t as:

- **exploration**: random $c \in C$.
- **exploitation**: c_i where $Q_{\mathcal{L}(I,c_i)}(c_i)$ is the maximum among $Q_{\mathcal{L}(I,c_1)}(c_1) \ldots Q_{\mathcal{L}(I,c_p)}(c_p)$ where $C = \{c_1, \ c_2, \ \ldots \ c_p\}$.

The choice is decided by a simple comparison between a random number ξ at interaction t and a computed value

$$\epsilon_t = \delta \times \frac{1 - e^{\mathcal{D}_{t-1}}}{1 + e^{\mathcal{D}_{t-1}}} + (1 - \delta) \times \epsilon_{t-1}$$

where $\mathcal{D}$ is defined as:

$$\mathcal{D}_t = \frac{-|Q_{\mathcal{L}(I,c)}(c) - Q_{\mathcal{L}(I,c)-1}(c)|}{\sigma}$$

with $c = i_t$, and δ and σ are constants inherent to VDBE design. Despite the sophisticated formula here, the $\mathcal{D}$ function reveals the simple philosophy behind VDBE. It utilizes the difference of Q of the last two rounds of results to guide the MAB to make decisions.

Classic MAB solutions are known to enjoy strong guarantees on optimality known as asymptotic effectiveness [16, 18, 28]. VDBE is a widely used practical variant of MAB for balancing exploration and exploitation.

3 AENEAS DESIGN

Aeneas is an application-runtime co-design which consists of a programming model (Section 3.1) and runtime stochastic support (Sections 3.2, 3.3, 3.4).

3.1 Aeneas Programming Model

Aeneas provides a simple and flexible programming model to help programmers customize their applications and the underlying stochastic energy optimizer. To streamline our support for Android, the Aeneas programming model is crystallized as a form of Java APIs. As seen in Listing 1, a simple example using the APIs shows a `LocationProvider` dynamically adjusts its accuracy priority (`gpsPrio`) and update interval (`gpsUpdate`) in the presence of uncertainty of GPS to meet a Service Level Agreement (SLA) of 20% per hour battery drain rate.

Three programming abstractions central to Aeneas are *knobs*, *rewards*, and *machines*. The three concepts are supported by 3 library classes `Knob`, `Reward`, and `AeneasMachine` respectively.

Knobs represent program elements where alternative values are allowed, a recurring theme in application-level energy-aware and

```
1  class LocationProvider implements Interactor {
2    AeneasMachine aeneas;
3    Knob gpsPrio = new DiscreteKnob(
4      new Integer[]{
5        ANDROID.PRIORITY_LOW_POWER, ANDROID.PRIORITY_HIGH_ACCURACY});
6    Knob gpsUpdate =
7      new InferredKnob(10000, 500, new Integer[]{1000, 5000});
8    LocationProvider() {
9      aeneas = new AeneasMachine(
10       new VBDE(), new Knob[]{gpsUpdate, gpsPrio},
11       new BRateReward(20), this);
12   }
13   void onInteract() {
14     LocationRequest req = new LocationRequest();
15     req.setUpdate(gpsUpdate.read());
16     req.setPriority(gpsPrio.read());
17     requestLocationUpdates(req);
18   }
19   void start() {aeneas.start(); }
20   void stop() {aeneas.stop(); }
21   void resume() {aeneas.resume(); }
22 }
23 class BRateReward extends Reward {
24    float brate;
25    EnergyReward() {this.brate = brate; }
26    float valuate() {
27        float e = this.perInteractionEnergy();
28        return this.batteryRate(e);
29    }
30    float SLA() { return brate; } }
```

Listing 1: An Aeneas Programming Example

approximate computing (*e.g.*, [5, 12]). In Aeneas, knobs may either be a *declared discrete knob* or an *inferred knob*. The former indicates a small set of choices given by the programmer. For example, at Line 7, two Android built-in values for GPS accuracy, `PRIORITY_LOW_POWER` and `PRIORITY_HIGH_ACCURACY`, are considered by the programmer as acceptable during the online optimization. The `DiscreteKnob` class, a subclass of `Knob`, captures this notion. The latter, an inferred knob, is used in scenarios when the programmer does not have a clear notion on the optimal setting, leaving it to be decided by the runtime optimizer — a case of inference. Inferred knobs are supported by another subclass of `Knob`, `InferredKnob`. For example, at Line 5, the programmer indicates the `gpsUpdate` field may be set at any value between 10000 (ms) and 500 (ms), and be optionally provided with “seed” values — such as 1000 ms and 5000 ms — for the underlying stochastic optimizer to start the search. An `InferredKnob` object consists of values of numerical type. Knob objects have a `read` method which returns a value determined by the stochastic optimizer, as seen in Lines 15-16.

Rewards can be viewed as programmer-definable Quality of Service (QoS) specifications, which is in turn used by the underlying runtime to determine the desirable knob setting in stochastic optimization. Among its methods, a subclass of `Reward` class can override a method `valuate` to define how reward should be computed, and a `SLA` method on what the goal of the optimal reward should be. When both methods are provided, the underlying optimizer attempts to operate the application so that the reward is as close to the defined `SLA` as possible. When the `SLA` is not overridden, Aeneas simply maximizes the reward. The `Reward` base class provides a default implementation that is equivalent as minimizing energy consumption. As energy is a commonly used metric to define QoS, the `Reward` also includes a utility method called `perInteractionEnergy` to compute the energy consumption between two latest `valuate` method invocations, and a utility method

Algorithm 1 AENEAS Discrete

```
struct CONFIG {
  $\rho$ : FLOAT[] // reward
  $\nu$ : INT // local counter
}
$\mathcal{R}$ : () $\rightarrow$ FLOAT // reward function
$\mathcal{T}$ : FLOAT // SLA target
MIN : INT // min interactions
MAX : INT // max interactions
CT : FLOAT // converge threshold
RATE : FLOAT // interaction rate
function MAINDISCRETE($\mathcal{K}$ : T[])
  $C \leftarrow$ INIT($\mathcal{K}$)
  $\mathcal{N} \leftarrow |C|$
  while $\mathcal{N} \leq$ MAX do
    $i \leftarrow$ MAB($C$)
    INTERACT($C, i, \mathcal{K}$)
    $\mathcal{N} \leftarrow \mathcal{N} + 1$
    if ($\mathcal{N} \geq$ MIN) && CONVG($C, \mathcal{N}$) break
  return $C$
function INIT($\mathcal{K}$ : T[])
  for $i$ in $0..|\mathcal{K}| - 1$ do
    $C[i] \leftarrow$ CONFIG([], 0)
    INTERACT($C, i, \mathcal{K}$)
  return $C$
function INTERACT($c$ : CONFIG, $i$ : INT, $\mathcal{K}$ : T[])
  SET($\mathcal{K}, i$)
  ONINTERACT()
  TIMEELAPSED(RATE){UPDATE($c$)}
function UPDATE($c$ : CONFIG)
  $c.\rho[c.\nu] \leftarrow \mathcal{T} - \mathcal{R}()$
  $c.\nu \leftarrow c.\nu + 1$
function CONVG($C$ : CONFIG[], $\mathcal{N}$ : INT)
  for all $c$ in $C$ do
    return true if $\frac{c.\nu}{\mathcal{N}} \geq$ CT
  return false
```

Algorithm 2 VDBE

```
$\delta$ : FLOAT $\leftarrow$ 1.0/KS //recommended by VDBE
$\sigma$ : FLOAT // inverse sensitivity
$\epsilon$ : FLOAT $\leftarrow$ 1.0 // latest epsilon
function MAB($C$ : CONFIG)
  if RANDOMFLOAT(0, 1) $< \epsilon$ then
    $i \leftarrow$ RANDOMINT(0, $|C| - 1$)
  else
    $i, q \leftarrow$ BEST($C$)
  $\mathcal{D} \leftarrow \frac{-|q - Ql(C[i].\rho)|}{\sigma}$
  $\epsilon \leftarrow \delta * \frac{1 - e^{\mathcal{D}}}{1 + e^{\mathcal{D}}} + (1 - \delta) * \epsilon$
  return $i$
function BEST($C$ : CONFIG)
  $i \leftarrow \mathbf{argmin}_{j \in 0..|C|-1} |Q(C[j].\rho)|$
  return $i, Q(C[i].\rho)$
function $Q$($r$ : FLOAT[])
  return $\frac{r[0] + \ldots r[|r| - 1]}{|r|}$
function $Ql$($r$ : FLOAT[])
  return $\frac{r[0] + \ldots r[|r| - 2]}{|r| - 1}$
```

Algorithm 3 AENEAS Inferred

```
struct FRAME {
  $\kappa$ : T[] // knobs
  $\theta$ : FLOAT // best Q value
  $\psi$ : T // best knob setting
}
IT : INT $\leftarrow$ 4 // interpolation density
$\mathcal{K}_0$ : T[KS] // initial inferred knob
$\mathcal{S}$ : STACK[FRAME] // Stack
function MAININFERRED
  $\mathcal{K} \leftarrow \mathcal{K}_0$
  while true do
    $C \leftarrow$ MAINDISCRETE($\mathcal{K}$)
    $i, q \leftarrow$ BEST($C$)
    if $q \neq_{\mathcal{T}} 0$ then
      $\mathcal{K} \leftarrow$ NEXT(FRAME($\mathcal{K}, q, \mathcal{K}[i]$))
function NEXT($f$ : FRAME)
  $f' \leftarrow$ TOP($\mathcal{S}$)
  case $f.\theta \times f'.\theta < 0$ // boundary cross
      PUSH($\mathcal{S}, f$)
      $\mathcal{K} \leftarrow$ INTERP($f.\psi, f'.\psi$))
  case $(|f'.\theta| - |f.\theta|) >_{\mathcal{T}} 0$ // tighten
      PUSH($\mathcal{S}, f$)
      $i \leftarrow \frac{\text{LAST}(f.\kappa) - \text{FIRST}(f.\kappa)}{2}$
      $\mathcal{K} \leftarrow$ INTERP($f.\psi - \frac{i}{2}, f.\psi + \frac{i}{2}$)
  otherwise // widen
      $f \leftarrow$ POP($\mathcal{S}$)
      $\mathcal{K} \leftarrow f.\kappa$
  return $\mathcal{K}$
function INTERP($i, j$ : T)
  $i \leftarrow$ MAX(MIN($i, j$), FIRST($\mathcal{K}_0$))
  $j \leftarrow$ MIN(MAX($i, j$), LAST($\mathcal{K}_0$))
  $k \leftarrow \frac{j - i}{\text{IT}}$
  return $[i, i + k, i + 2 \times k, \ldots, j]$
```

Figure 2: AENEAS Algorithm Specification

`batteryRate` to convert energy to battery life in the unit of percentage/hour. These base implementations rely on querying low-level `Android` battery management APIs (more details in Section 4). The `Reward` design reflects the flexible extensibility and portability of AENEAS API: not only an end-programmer may define customized QoS/energy metrics, but also a framework-level programmer may customize how battery should be accounted for — when low-level battery management solutions evolve from one smartphone to another — by overriding methods such as `perInteractionEnergy`. Either way, the underlying stochastic optimizer remains unchanged.

Finally, an `AeneasMachine` may be created, the programming abstraction for the stochastic optimizer itself. Each `AeneasMachine` may be customized by the MAB strategy, the available knobs, and the reward. One such machine can be seen at Line 9. The last argument of the `AeneasMachine` constructor is an object with `Interactor` interface. This interface consists of one method called `onInteract`, the callback method for specifying application-specific behavior each time an MAB interaction is initiated, as seen in Line 13. In addition, our framework allows flexible lifecycle management of the `AeneasMachine`, such as starting, stopping, restarting, and resetting the optimizer (`start`, `stop`, `resume`, `reset`).

3.2 Declared Discrete Knob Runtime Support

The algorithm used by AENEAS for runtime optimization is specified by Algorithm 1. The key data structure maintained at run time is CONFIG, a *configuration*. Intuitively, each configuration indicates a particular knob combination, such as setting `gpsPrio` to `ANDROID.PRIORITY_HIGH_ACCURACY` and setting `gpsUpdate` to 5000. This combination is indicated by the ϕ field of the configuration struct. At run time, each configuration is used for a number of interactions, kept by counter ν. For each interaction on the configuration, a reward is stored in the array ρ. For simplicity, the algorithm shows only one knob $\mathcal{K}$ with `KS` number of settings. In the presence of multiple knobs, the number of the configurations is the cartesian product of all knob settings. Finally, the $\mathcal{R}$ is the abstract representation of the `valuate` method defined by the user, and $\mathcal{T}$ is the SLA provided by the programmer.

The optimizer operates in its own thread, and its algorithm is conceptually simple. After a "warm-up" procedure to make sure each configuration is at least attempted once (INIT), the algorithm henceforth alternates between inquiring the MAB to select a configuration (MAB), and performing an interaction based on the selected configuration (INTERACT). Within each interaction, the runtime first sets the knob settings according to the selected configuration (SET), calls the programmer-defined callback method (ONINTERACT), and updates the reward for the chosen configuration (UPDATE). The reward is computed as the difference between the latest reward returned by $\mathcal{R}$ and the specified by SLA $\mathcal{T}$. When SLA is not provided, $\mathcal{T}$ is defaulted at 0, so the reward used for the update is the value computed by the $\mathcal{R}$. We delay the discussion of TIMEELAPSED until Section 3.4.

Since MAB algorithms are diverse, Algorithm 1 is abstract in its MAB policy choice. We provide a concrete VDBE-based implementation of the MAB function in Algorithm 2. For clarity, we choose the same metavariables for this algorithm specification as those in Section 2 when applicable, so the algorithm should be self-explanatory. Note that Q values can be computed incrementally — the choice of Aeneas — but we do not define it here for simplicity.

We will delay the explanation of `MIN`, `MAX`, `CT`, and the function of ConvG until next section. As a standalone algorithm for declared discrete knobs, `MIN` is set to be 0, `MAX` is set to be a largest constant, `CT` is set to be 100%.

3.3 Inferred Knob Runtime Support

The algorithm used by Aeneas for inferred knob optimization is specified by Algorithm 3. Conceptually, an inferred knob represents a *range* of possible knob combinations, such as 10 seconds to 1 second for `gpsUpdate`. Concretely, an inferred knob exists as a set of iteratively-updating discrete values within the user supplied range. We represent an inferred knob's current set of discrete values with a Frame that keeps a collection of values κ, a reward SLA associated with the frame θ and the best knob setting within the values ψ. We maintain a stack $\mathcal{S}$ of inferred knob values. When working with real values, we define a closeness relation $\neq_{\mathcal{T}}$ where $\mathcal{T}$ is a user-defined threshold for an acceptable precision, analogous to `equate` in the programmer API.

The process of stochastic optimization for inferred knobs consists of a sequence of *iterations*. Each iteration is indeed a stochastic optimization for declared discrete knobs, as indicated as the use of MainDiscrete function in the algorithm. Unlike a *bona fide* declared discrete knob optimization where the optimizer may continuously (and potentially infinitely) perform interactions, each iteration in an inferred knob optimization must terminate so that the next iteration may start. This explains why the MainDiscrete function definition consists of the `MIN` and `MAX` constants to bound the minimum and maximum number of interactions to try in each iteration. In addition, the ConvG function computes a *convergence ratio*, *i.e.*, the percentage of interactions of a particular configuration being used within the current iteration. If it exceeds a threshold `CT`, the iteration can also terminate, intuitively indicating the stochastic optimizer has converged toward a particular configuration.

On the high level, the algorithm connects iterations through *iterative refinement*. After an iteration, the inference algorithm selects the configuration closest to the SLA (Best). If it is not "close enough" — algorithmically represented as the q value approximates 0 defined by the $\neq_{\mathcal{T}}$ relation — the optimizer starts a new iteration (Next). It decides upon a new set of values by comparing the current frame f with the previous frame f'. Conceptually, there are three cases:

- *Boundary Crossing*: f and f' carry "opposite" rewards – one above the SLA and the other below the SLA—indicating the optimal values exist somewhere between the best values of the two frames. We perform interpolation (INTERP) between the best knob settings represented by f and f'.
- *Tightening*: the reward produced by f is closer to the SLA than that of f', indicating f is the more desired choice. We half the range of the inferred knob around the current setting represented by f, interpolate a new set of settings within the range, and lead to a new iteration exploring a "tighter" range.
- *Widening*: the reward produced by f is further away to SLA than f'. In this case, we need to reverse the decision made by the previous iteration, leading to a pop on the stack.

3.4 Parameter Selection and Self Optimization

An important design issue in Aeneas stochastic optimization is the *interaction rate*, *i.e.*, how frequent a new interaction should be triggered. This is shown in Algorithm 1 as `RATE`, which determines the delay associated with firing Update with TimeElapsed as seen in the same algorithm. Intuitively, a higher interaction rate may improve the adaptiveness of the optimization, whereas the lower interaction rate may reduce overhead. In the case of GPS applications on mobile devices, two additional thorny issues exist. First, there is a delay effect in updating the GPS interval, *i.e.*, setting a new GPS interval on the Google API level does not imply GPS hardware will immediately take a new sample based on the new interval. Second, fuel gauge on Android devices (we will discuss in Section 4) has its own rate for updates. Setting an interaction rate higher than the fuel gauge update rate implies reading the same value as the previous interaction. With these hardware/system-level limitations, an excessively high interaction rate would lead to erroneous results that the reward may not reflect the effect of the interaction it is intended to. Manually setting the interaction rate is a challenge.

A key insight of Aeneas is that the interaction rate for our GPS application can be determined by *stochastic optimization itself*. The self-optimizer performs inferred stochastic optimization with the interaction rate as the knob. The algorithm is identical to the inferred knob algorithm we described in Section 3.3, with three exceptions. (i) for each interaction, the self optimizer adjusts `RATE` based on the inferred knob. (ii) the knob for the GPS update interval is alternately set between the fastest interval and the slowest interval. (iii) the reward is set as the power difference of the last two interactions (in watts). The intuition here is that, if the delay effect we described earlier does not occur, the design of (ii) above *should* lead to significant power difference of the last two interactions.

Throughout our experiments we found the interaction rate is the most sensitive parameter of algorithm design, which justifies the automated approach of self optimization. For other parameters, such as the VDBE's δ, σ, and ϵ as in Algorithm 2, we rely on settings recommended in existing literature. Note that our framework itself is fully extensible: a programmer with a different parameter setting of VDBE can provide an alternative `VDBE` (sub-)class.

4 AENEAS EVALUATION

In this section, we report our findings while experimenting Aeneas "in field," *i.e.*, while being used in the real-world use scenarios of driving, biking, and hiking. This is a significant effort. For driving alone, we have covered roughly 2000 miles for early-stage testing and exploratory data collection, and 4500 miles to collect data reported in this section. In total, we have collected approximately 150 hours of driving data over the duration of the project, as well as 20 hours of biking and hiking.

4.1 System Setup

Experiments were performed on Google Nexus 5X Android phones with Qualcomm Snapdragon 808 processor at 1.8 GHz, 2GB RAM, and Android Runtime (ART) 2.1.0 with Android 6.0 Marshmellow OS. `MAPS.ME` (Release Build 7.1.3) utilizes Android's GPS through the LocationServices Google API. Accelerometer sensors and bluetooth are switched off, and the phones are set at the 'Do Not Disturb" mode. Since display system is known to be a signifiant contributor to energy consumption in smartphones [7, 20], we choose to leave them *on* with bright set to 100% and auto-adaptive brightness set to off. This is a challenging base setting to start with. In the following sections, we will show that even when the display system is set to be fully in operation, GPS energy optimization remains relevant.

Energy samples were collected using the phones' internal fuel-gauge chips, whose battery voltage and current can be queried programmably through `BatteryManager` API. Some experiments uses drain rate — such as 20% per hour — as a form of reward specified by programmers. To correlate drain rate with energy consumption, we first start the phone at 100% battery and run `MAPS.ME` until Android sends out a critical power warning—this is done via the `android.intent.action.BATTERY_LOW` intent—triggered at 15% battery life. We observed that between this range (100% - 15%), battery discharge is linear. Hence, the energy consumption of a user-provided drain rate — such as 20% — can be computed proportionally.

4.2 Experiment Design

We structure the evaluation of Aeneas with 4 types of experiments. First and second, we evaluate declared discrete knobs and inferred interval knobs, an in-field validation of Algorithm 1 and Algorithm 3 respectively, detailed in the immediately following subsections. Third, we compare our stochastic optimizer with non-stochastic profile-based optimization, with experiments detailed in Section 4.5. Fourth, we report our results on self optimization in Section 4.6. Finally, we present overhead analysis in Section 4.7.

We use a combination of three GPS parameters as knobs. *GPS Update Interval* represents the *frequency* that updates are provided to the application from the GPS hardware. *GPS Priority* represents the power mode of the GPS hardware. *GPS Maximum Wait Time* represents the maximum additional wait time the application can tolerate once an update has been received. This parameter allows for a form of "batching" where location updates are delayed beyond the specified update interval and delivered in a batch, resulting in "slower" updates.

Value-Difference Based Exploration is the default stochastic policy chosen for Aeneas. All experiments use 32.5 seconds as the interaction rate. We justify this choice in Section 4.6. We set VDBE's inverse sensitivity σ to 5.0 for experiments with discrete knobs. This causes less exploration and quicker convergence, ideal properties when primarily using discrete knobs. When running experiments with inferred knobs, we set σ to 1.0 which causes more exploration throughout the run. For the inferred knob optimization, we set `MIN`, `MAX`, and `CT` in the algorithm specification as 15, 30, and 60%.

For driving experiments, routes were selected that followed a 60MPH speed limit as much as possible, in a combination of urban and rural environment.[2] In all experiments of driving, biking, and hiking, the `MAPS.ME` application actively displays updated location information at all time.

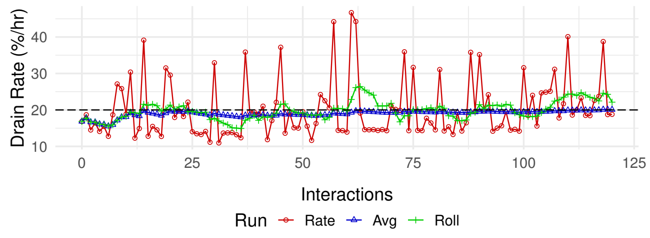
[2]The details of the route information is available at our project site.

4.3 Energy Optimization with Declared Discrete Knobs

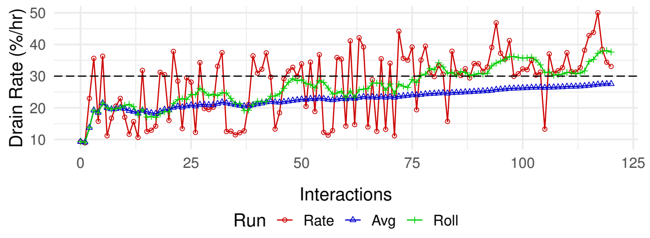
We first evaluate how Aeneas targets a specified SLA in the presence of alternative settings for multiple knobs, based on the algorithm specified in Algorithm 1. We define reward to be the instantaneous battery drain rate in percentage drain per hour. This was done by overriding Aeneas default `Reward` class as shown in Section 3. We perform the experiment targeting two SLAs, 20 and 30 percent drain rate per hour (%/hr). We have 3 knobs—GPS Update Interval with 10s,7.5s,5.5s,3.5s, and 1s, GPS Priority with `LOW_POWER` and `HIGH_ACCURACY`, and GPS Wait Time with `5x` and `1x`—each with 5, 2, 2, settings respectively. In total, there are $5 \times 2 \times 2 = 20$ configurations for the stochastic optimizer to choose from. This creates a challenging and realistic case in that it may not be obvious to a programmer which of the 20 configurations is the best. Indeed, there may be multiple reasonable configurations that exhibit similar energy behavior.

Results from representative runs are presented in Figure 3. In all experiments — with different drain rate SLAs and with different modes of transportation — Aeneas demonstrates remarkable effectiveness and stability in converging on configurations that make best efforts to meet the specified SLA. In all figures, the blue line — accumulated average drain rate— approaches the target drain rate SLA as time proceeds. Observe that the "rolling average," *i.e.*, the most recent drain rate of the last 10 interactions — indicated by the green line — may often lie far from the target SLA in the beginning stage of each run; however, as more and more interactions are attempted, Aeneas intelligently selects configurations that "get better and better", resulting in the rolling average tending toward the target SLA. Furthermore, despite the stable trend indicated by the blue line and the green line, the red line continues to fluctuate. This line indicates the per-sample drain rate. The fluctuation results from two reasons. First, different configurations may be attempted at different interactions, resulting in different instantaneous drain rate. This is consistent with the stochastic design itself: even after the initial warm-up stage (interactions 0-25), the system continues to explore potentially sub-optimal configurations (but with less likelihood). Second, uncertainty in instantaneous energy behavior is inherent, so the drain rate may still fluctuate even in the presence of the same configuration. This can be seen toward the end of each trace (such as interactions 75-120) where sub-optimal explorations are significantly reduced, yet the red line continues to fluctuate significantly. As the blue/green lines indicate, Aeneas operates successfully in this uncertain environment.

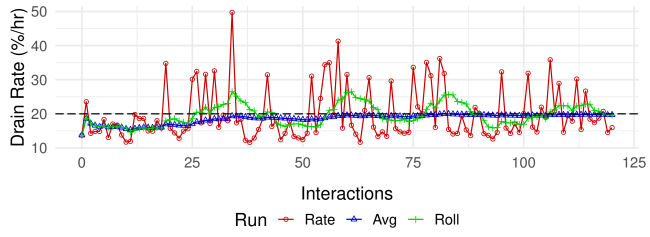
Comparatively, Figure 3a and Figure 3b show that 20%/hr experiments converge faster than the 30%/hr experiments. This trend is similar across the different transportation modes. This highlights the limitation of stochastic optimization based on declared discrete knobs fully provided by programmers. After inspecting the traces, we found that few among the 20 configurations generated from the supplied knobs have energy profiles near 30%/hr SLA, but the optimizer must explore these configurations enough to find an ideal configuration. This can be seen from observing the relatively low battery drain rate during the initial warm-up phase of the 30%/hr run (interactions 0-25), and also time for additional exploration (interactions 25-50). Approaching the end of each trace, there remains a 3%-5% gap between the accumulated average and the target



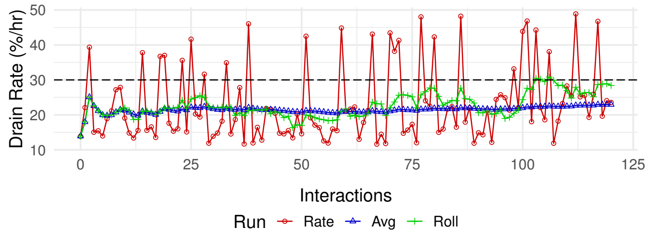
(a) `Drive 20 %/hr SLA`



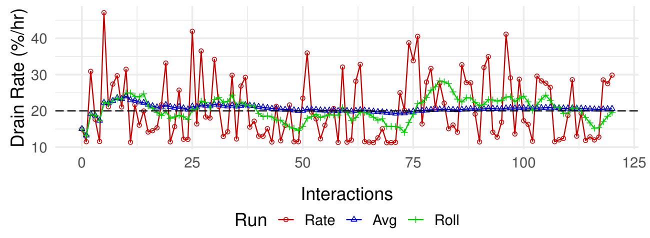
(b) `Drive 30 %/hr SLA`



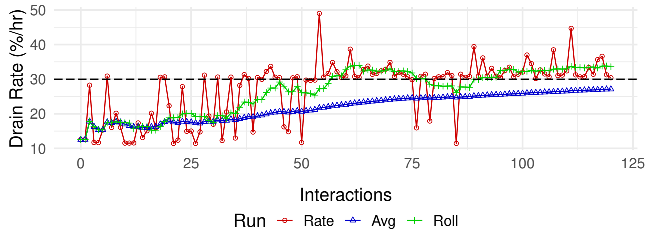
(c) `Bike 20 %/hr SLA`



(d) `Bike 30 %/hr SLA`



(e) `Hike 20 %/hr SLA`



(f) `Hike 30 %/hr SLA`

Figure 3: Traces with Declared Discrete Knobs. Each graph is a representative trace of a run lasting approximately 1 hour. The X-Axis represents the elapsed interactions. The red line shows the instantaneous battery drain rate at each interaction; the blue line shows the accumulated average battery drain rate of the run; the green line shows the 10-interaction rolling average battery drain rate. The dotted horizontal line represents the target battery drain rate SLA.

SLA, as a consequence of early sub-optimal interactions. Here, the rolling average line indicates in all 30%/hr experiments, the rolling average of the drain rate is indeed converging toward the SLA from around interaction 50 onward.

Taking these figures as a whole, Aeneas demonstrates consistent effectiveness in different transportation modes and different target SLAs. In addition to these representative runs, the left half of the columns of Figure 4 details statistics on all runs. Observe that the percentage of occurrences that the rolling average drain rate falls within 3%/hr of the target SLA generally increases from the first quarter of the experiment to the final quarter, reflecting the intuition that the Aeneas learning algorithm incrementally finds more optimal configurations as time goes on. We report traces from other runs at the project website.

One challenging case is perhaps biking with SLA target of 30%/hr. In our experience, we found it difficult to construct a one-hour bike ride without frequent stops and interruptions, as opposed to driving. In addition, biking does cover a relatively large area with significant GPS signal variations, as opposed to hiking. In essence, biking compounds the uncertainty from both driving and hiking. This is evidenced by the relatively large variance between energy readings, as seen in Figure 3d. In addition, the relatively low number of *discrete* configurations that have energy profiles friendly for the target 30%/hr SLA poses a challenge for our experiment. As we shall see in Section 4.4, this last challenge is overcome when Aeneas works with inferred knobs, leading to improved results for 30%/hr biking.

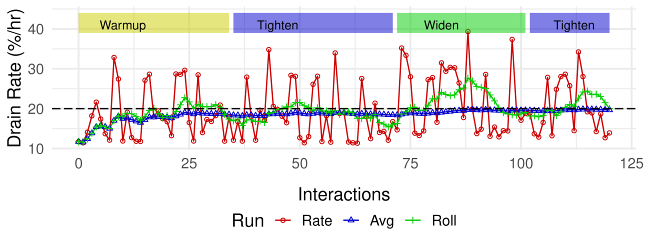
4.4 Energy Optimization with Inferred Interval

Next we evaluate how Aeneas infers unknown knob settings that target a programmer-defined SLA in the presence of uncertainty, following the algorithm specified by Algorithm 3. We define an inferred knob for GPS update interval with initial range of 10s to 1s with 5 interpolation points, resulting in 10s, 7.5s, 5s, 3.5s, and 1s for the initial set of intervals and fix GPS priority and GPS maximum wait time to `HIGH_ACCURACY` and 1x respectively. Reward and SLA are as defined in Section 4.3.

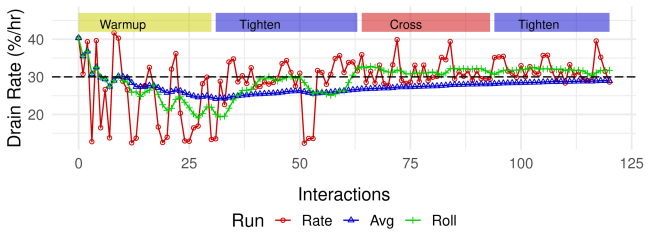
Results from representative runs are presented in Figure 5. For all experiments, the accumulated average drain rate—the blue line—tends toward the target SLA as Aeneas performs iterations of inference. Each iteration ends with a set of *more refined intervals*. Tightening and boundary crossing — which refine an inferred knob

	Declared Discrete								Inferred							
	Accumulated	σ	Roll	σ	Q1	Q2	Q3	Q4	Accumulated	σ	Roll	σ	Q1	Q2	Q3	Q4
Drive-20	18.79	4.13%	18.78	17.19%	86.67%	90.83%	88.33%	96.77%	20.34	2.83%	19.92	16.75%	70.83%	95.0%	89.17%	83.06%
Bike-20	19.23	2.49%	18.47	13.01%	86.67%	90.83%	87.5%	94.35%	24.05	2.31%	21.37	13.45%	29.17%	65.0%	58.33%	78.23%
Hike-20	19.46	3.54%	17.88	10.08%	92.5%	92.5%	81.67%	85.48%	22.74	5.21%	22.64	12.05%	79.17%	74.17%	72.5%	72.58%
Drive-30	25.3	6.28%	31.14	13.07%	0.83%	31.67%	69.17%	83.87%	28.56	2.25%	30.75	2.62%	46.67%	86.67%	97.5%	100.0%
Bike-30	22.42	2.21%	22.67	14.84%	13.33%	15.83%	35.0%	20.16%	26.89	12.49%	24.31	15.57%	36.67%	43.33%	53.33%	44.35%
Hike-30	25.6	7.98%	31.23	10.33%	4.17%	45.0%	73.33%	70.16%	27.46	3.53%	29.13	7.07%	59.17%	59.17%	84.17%	86.29%

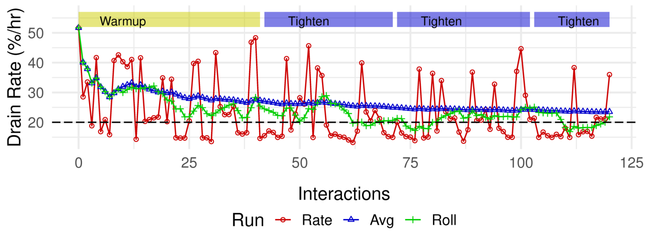
Figure 4: Declared Discrete and Inferred Interval Experiment Statistics: We show the mean and deviation of `Accumulated` and `Roll` for all runs. In addition, we show the percentage of times that `Roll` was within 3%/hr of the target SLA for four quarters of the experiment runtime (Q1-Q4).



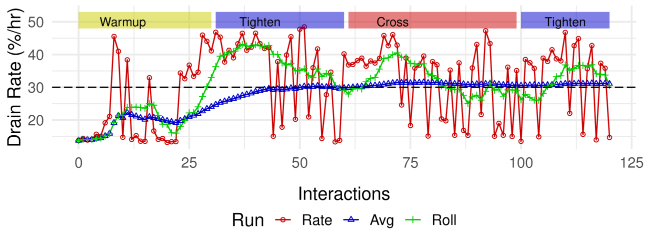
(a) `Drive 20 %/hr SLA`



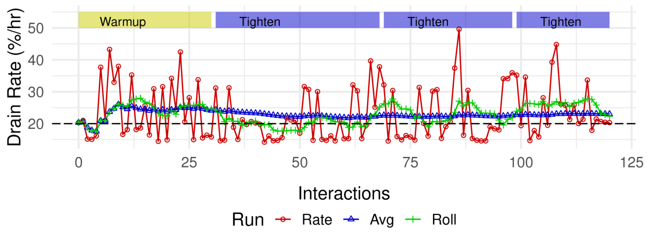
(b) `Drive 30 %/hr SLA`



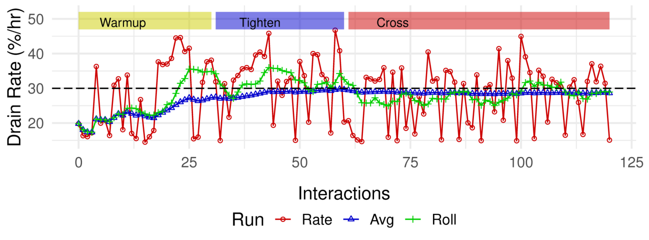
(c) `Bike 20 %/hr SLA`



(d) `Bike 30 %/hr SLA`



(e) `Hike 20 %/hr SLA`



(f) `Hike 30 %/hr SLA`

Figure 5: Traces with Inferred Knobs. The blue, green, red, and yellow labeled boxes indicate iterations where tightening, widening, boundary crossing, and warmup were performed, respectively. All other legends are identical to those in Figure 3.

towards an often smaller and more specific range of settings — occur more frequently than widening. We can see the rolling average—the green line—stabilizes throughout the iterations of inference. This is good news for programmers who are unsure about the behavior of their application; they can simply provide suggestions—in the form of inferred knobs—and let the Aeneas inference algorithm discover ideal settings.

Of particular interest is how the Aeneas inference algorithm adapts to uncertainty. For example, when driving with 20%/hr SLA (Figure 5a), instantaneous drain rate experiences significant fluctuation, leading to sub-optimal results after the initial iteration of tightening. Consequently, the optimizer widened in the next iteration to reach a more stable state. The final iteration of tightening leads to a more refined set of intervals, finishing with a running average of 19.87%/hr. Additionally, we can see a similar pattern when biking targeting 30%/hr SLA (Figure 5d). Observe the initial shift in rolling average after a boundary cross was performed. As the new set of intervals stabilize, a final iteration of tightening brings the average rate to 31.02%/hr. In both cases, Aeneas adaptively adjusts to the environment and stabilizes around the desirable SLA.

The right half of the columns of Figure 4 details statistics on all runs. We report traces from other runs at the project website.

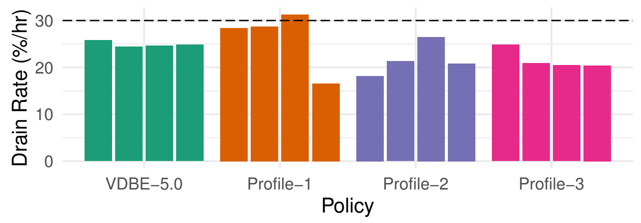
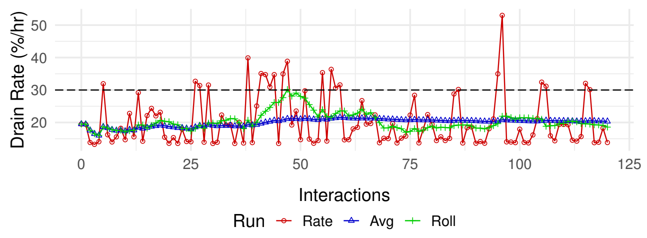
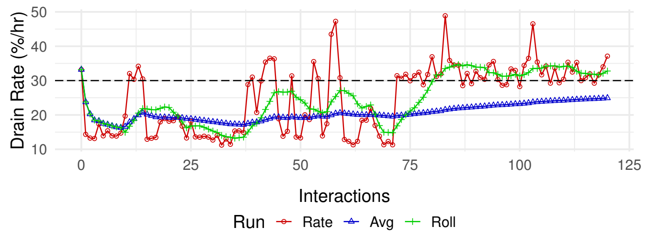


Figure 6: Stochastic vs Non-Stochastic Approaches: Each bar represents a single drive that lasts for approximately 1 hour (120 interactions). The Y-Axis shows the percent battery drain rate per hour of the run. The dotted horizontal bar represents the targeted 30%/hr SLA.



(a) `Profile-1` Run-4



(b) `Profile-3` Run-1

Figure 7: Profile-Guided Optimization Traces. Figure (a) is trace from the 4th run of `Profile-1` in Figure 6. Figure (b) is trace from the 1st run of `Profile-3` in Figure 6. All other legends are identical to Figure 3.

4.5 Stochastic vs. Non-Stochastic Approaches

In this section, we compare Aeneas stochastic optimizer with a non-stochastic approach, profile-guided optimization. In this latter approach, each configuration is chosen for a fixed number of times, after which the execution commits to the configuration with the best average reward. We use `Profile-X` to indicate a profile-guided run where each configuration is chosen for X number of times. We use the same knob and reward settings as done in Section 4.3.

Figure 6 shows the results of targeting an SLA of 30%/hr drain rate. Our results show the main drawback of profile-guided optimization is its *instability*. Its effectiveness is dependent on how representative the profiled runs are. When the level of uncertainty is high, profile-guided optimization can make a poor choice that make a large impact for the rest of the run. In general, profile-guided optimization becomes more stable as the number of samples

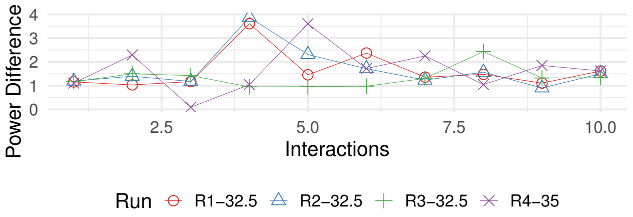


Figure 8: Self Optimization. We show the last 10 interactions *after* the convergence has been reached by the inferred knob algorithm. Each point represents the observed power difference (reward) between 10s and 1s GPS update interval configurations. Run are labeled with the interaction rate converged on by the algorithm.

drawn increase. `Profile-3` runs have a narrower range of difference (20.3%hr-24.8%hr) than `Profile-1` (16.5%/hr-31.2%/hr). Due to its shorter sampling period, when `Profile-1` makes a poor choice, as is the case for the 4th run for `Profile-1`, the penalty is severe. While `Profile-3` is more stable, there is an inherent cost of increasing the sampling for profile-guided optimization: it samples 60 times, requiring 30 minutes, before making a choice.

In contrast, Aeneas only needs to sample each configuration once in the warm-up stage — behaving like `Profile-1` in the first 20 interactions – but then subsequently relies on stochastic exploration to mitigate potentially unreliable samples in the warm-up stage. As a result, `VDBE-5.0` has the narrowest range (24.4%/hr - 25.7%/hr) compared with all profile-guided runs, demonstrating stability.

To zero in on the detailed behavior of profile-guided optimization, Figure 7 shows the traces of two representative runs: the fourth run of `Profile-1` and the first run of `Profile-3`. In the first case, the run unfortunately picked a sub-optimal configuration after the 1-sample-per-configuration profiling, and can no longer recover for the rest of the run. In the second case, the run did choose a relatively reward-optimal configuration, but that only happened after 60 interactions. The rest of the run does indeed have favorable rolling average drain rate, but it may take a long time for the rest of the run to compensate for the significant sub-optimal battery drain during the profiling stage.

4.6 Aeneas Self Optimization

In Section 4.2 we mentioned that we set the Aeneas interaction rate to 32.5 seconds per interaction. This value is neither hard-coded nor heuristics-based; instead, we computed it through our self optimizer described in Section 3.4. In that experiment, we alternate between 10s and 1s as the slowest and fastest update interval respectively.

Results from representative runs are presented in Figure 8. The self-optimizer converges on 32.5 seconds per interaction as the interaction rate for R1, R2, and R3, and 35 seconds for R4. Among all runs, nearly all converged on a value between 32.5 seconds and 35 seconds. We hence set the 32.5 seconds as the default setting for interaction rate for the experiments.

4.7 Aeneas Overhead

Overhead introduced from Aeneas comes from knob, configuration, and MAB bookkeeping, *i.e*, the maintenance of metadata for each configuration—accumulated reward and knob profiles—as well as running the selected MAB algorithms, and Aeneas thread management. We measure the cost of running Aeneas in terms of additional energy overhead by comparing against a baseline native `MAPS.ME` with the 3 knobs relevant to our experiments—GPS Update Interval, GPS Priority and GPS Wait Time—set to 500ms, `HIGH_POWER` and `1x`. The Aeneas run performs all bookkeeping and stochastic optimization, but is forced to select the same configuration as the native `MAPS.ME` one after each interaction. We repeated the experiment 10 times. The mean energy consumption of the native runs is 5361 (J) with a relative standard deviation of 6.97%, and the mean energy consumption of the Aeneas runs is 5511 (J) with a relative standard deviation of 9.25%. Aeneas incurs an average additional 150.22 (J), representing 2.8% energy overhead.

5 RELATED WORK

PowerDial [12] is a dynamic adaptation framework for power optimization. Built with a control-theoretic core, PowerDial allows a number of system knobs — Dynamic Voltage and Frequency Scaling (DVFS) and server consolidation in particular — to be dynamically adjusted to balance the trade-off between performance and Quality of Service (QoS). JouleGuard [11] employs a two-tier optimization approach, where system-level energy optimization such as through DVFS can be maintained by MAB, with which the application-level approximations can be coordinated through a PowerDial-like algorithm. JouleGuard is interesting for its formal control-theoretic guarantees, well suited for co-optimization problems where the boundary between the application and the system is well-carved. In our interested problem domain, GPS applications, it is a challenge to differentiate the two. For example, GPS update intervals are set by programmers, and may be adjusted within the Google LocationServices APIs. It is unclear whether this is an application knob or system knob. From this perspective, Aeneas explores a different — perhaps complementary — non-hierarchical path where application knobs (and system knobs if any) are unified through MAB. Desirable properties of energy optimization — such as adaptiveness, convergence, and stability — thus go hands in hands with the MAB algorithm core. From a practical perspective, this unified approach avoids the need for reconciling and tuning two adaptive approaches — both MAB and control come with algorithm-level parameters — significantly simplifies deployment.

A few application-level frameworks — especially in the form of programming models — exist to support uncertainty. In Uncertain<T> [4], programming abstractions are provided so that program values may carry types that indicate data/results may be uncertain, and the programming language provides a Bayesian network semantics to ensure values of such types are correctly flown in the program. Aeneas may complement Uncertain<T> by providing online adaptive support for modeling the level (*e.g.*, distribution) of uncertainty without prior knowledge before program execution, and the reasoning framework of Uncertain<T> may provide feedback to Aeneas to correlate the knobs used by MAB. This complementary relationship may also apply to Rely [6] and Chisel [23], two programming languages where program types may include a *reliability specification*, capturing how the values may be located on underlying unreliable hardware.

There are a number of other programming and annotation frameworks to support energy-aware programming, such as Eon [31], Green [3], EnerJ [29], Energy Types [8], LAB [15], Eco [34], Ent [5], and APE [24]. Most of these frameworks facilitate adaptive energy management. For example, Ent allows programmers to specify mode-alternative behavior through a programming abstraction called mode cases; an APE programmer may annotate programs with policies managed by timed automata. A large body of work establishes the complex and dynamic nature of energy behavior in modern applications [14, 19, 25–27]. None of these approaches directly address uncertainty, and Aeneas may serve as an underlying framework for them to model uncertainty and guide the semantics-driven program runtimes to make judicious choices among programmer-defined alternatives.

MAB algorithms can be broadly viewed as a form of reinforcement learning (RL). The dilemma between exploration and exploitation is a classic problem in RL, and one of the most classic RL algorithms — Q-learning — involves iterative value update of a value function, a similar process as MAB algorithms. Indeed, we presented the MAB model in Section 2 in a recurrent Q update form to show this connection. RL has been used on the CPU-level energy optimization for optimal DVFS voltage-frequency selection [13, 30]. A temporal difference learning method coupled with Bayesian classification [33] is designed for balancing power and latency in storage/network systems. RL has also recently been used to guide resource consolidation in cloud computing [10] and data center optimization [22]. None of these efforts is focused on higher layers of the computing stack, such as how application-level energy management should be supported, and how the interface between the application and the runtime should be streamlined in the presence of uncertainty. Aeneas is also distinctive in its domain: a systematic study on energy optimization in the presence of uncertainty in mobile/transportation systems.

GPS energy consumption has long been recognized as significantly impacting battery life of mobile devices, with a large body of solutions. For example, A-Loc [21] presents a Bayesian estimation framework to model user location and sensing errors. Location sensing requests may also be optimized, such as substituted or suppressed [35]. GPS optimization may also take into account of context information, such as space and temporal data [9].

6 CONCLUSION

Aeneas is an application-level energy optimization framework with a simple programming model and a sophisticated stochastic runtime optimizer. Aeneas has been extensively evaluated in the context of GPS applications, through significant in-field validation involving driving, biking, and hiking. Aeneas is an open-source project. The source code of our framework, all raw data collected for our experiments, and additional trace graphs omitted from the paper, can be found at the project website.

ACKNOWLEDGMENTS

We thank Barry Williams for his help with device setup, and Timur Babakol for useful discussions. This work is funded by US NSF grant CCF-1526205, JST CREST, and KAKENHI 18H03219.

REFERENCES

[1] Bloomberg smartphone article, https://www.bloomberg.com/news/articles/2016-01-12/charging-a-smartphone-while-driving-isn-t-as-free-as-you-think.

[2] Google locationservices, https://developers.google.com/android/reference/com/google/android/gms/location/LocationServices.

[3] Baek, W., and Chilimbi, T. M. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI'10*, pp. 198–209.

[4] Bornholt, J., Mytkowicz, T., and Kathryn, M. Uncertain<t>: a first-order type for uncertain data. In *ASPLOS '14*, pp. 51–66.

[5] Canino, A., and Liu, Y. D. Proactive and adaptive energy-aware programming with mixed typechecking. In *PLDI 2017*.

[6] Carbin, M., Misailovic, S., and Rinard, M. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA '13*, pp. 33–52.

[7] Carroll, A., and Heiser, G. An analysis of power consumption in a smartphone. In *USENIXATC 2010*.

[8] Cohen, M., Zhu, H. S., Emgin, S. E., and Liu, Y. D. Energy types. In *OOPSLA '12*.

[9] Donohoo, B., Ohlsen, C., Pasricha, S., and Anderson, C. Exploiting spatiotemporal and device contexts for energy-efficient mobile embedded systems. DAC 2012.

[10] Farahnakian, F., Liljeberg, P., and Plosila, J. Energy-efficient virtual machines consolidation in cloud data centers using reinforcement learning. In *PDP 2014*.

[11] Hoffmann, H. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pp. 198–214.

[12] Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., and Rinard, M. Dynamic knobs for responsive power-aware computing. In *ASPLOS '11*.

[13] Juan, D.-C., and Marculescu, D. Power-aware performance increase via core/uncore reinforcement control for chip-multiprocessors. In *ISLPED 2012*.

[14] Kambadur, M., and Kim, M. A. An experimental survey of energy management across the stack. In *OOPSLA 2014*.

[15] Kansal, A., Saponas, S., Brush, A. B., McKinley, K. S., Mytkowicz, T., and Ziola, R. The latency, accuracy, and battery (lab) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *OOPSLA '13*, pp. 661–676.

[16] Kleinberg, R. Anytime algorithms for multi-armed bandit problems. In *SODA 2006*.

[17] Kuleshov, V., and Precup, D. Algorithms for the multi-armed bandit problem. *The Journal of Machine Learning Research 65*, 12 (2016).

[18] Lai, T., and Robbins, H. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics 6*, 1 (1985), 4–22.

[19] Li, D., Lyu, Y., Gui, J., and Halfond, W. G. Automated energy optimization of http requests for mobile applications. In *ICSE 2016*.

[20] Li, D., Tran, A. H., and Halfond, W. G. J. Nyx: A display energy optimizer for mobile web apps. In *ESEC/FSE 2015*.

[21] Lin, K., Kansal, A., Lymberopoulos, D., and Zhao, F. Energy-accuracy trade-off for continuous mobile device location. In *MobiSys 2010*.

[22] Lin, X., Wang, Y., and Pedram, M. A reinforcement learning-based power management framework for green computing data centers. In *IC2E 2016*.

[23] Misailovic, S., Carbin, M., Achour, S., Qi, Z., and Rinard, M. C. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *OOPSLA 2014*.

[24] Nikzad, N., Chipara, O., and Griswold, W. G. Ape: An annotation language and middleware for energy-efficient mobile application development. In *ICSE 2014*.

[25] Pinto, G., Canino, A., Castor, F., Xu, G., and Liu, Y. D. Understanding and overcoming parallelism bottlenecks in forkjoin applications. In *ASE 2017*.

[26] Pinto, G., Castor, F., and Liu, Y. D. Understanding energy behaviors of thread management constructs. In *OOPSLA 2014*.

[27] Ribic, H., and Liu, Y. D. Energy-efficient work-stealing language runtimes. In *ASPLOS 2014*.

[28] Robbins, H. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society 58*, 5 (1952), 527–535.

[29] Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., and Grossman, D. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI'11*.

[30] Shen, H., Lu, J., and Qiu, Q. Learning based dvfs for simultaneous temperature, performance and energy management. In *ISQED 2012*.

[31] Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M. D., and Berger, E. D. Eon: a language and runtime system for perpetual systems. In *SenSys '07*, pp. 161–174.

[32] Tokic, M. Adaptive ϵ-greedy exploration in reinforcement learning based on value differences. In *KI 2010*.

[33] Wang, Y., and Pedram, M. Model-free reinforcement learning and bayesian classification in system-level power management. *IEEE Transactions on Computers 65*, 12 (2016).

[34] Zhu, H. S., Lin, C., and Liu, Y. D. A programming model for sustainable software. In *ICSE'15* (2015), pp. 767–777.

[35] Zhuang, Z., Kim, K.-H., and Singh, J. P. Improving energy efficiency of location sensing on smartphones. In *MobiSys 2010*.