`maybe` We Should Enable More Uncertain Mobile App Programming

Geoffrey Challen, Jerry Antony Ajay, Nick DiRienzo, Oliver Kennedy,
Anudipa Maiti, Anandatirtha Nandugudi, Sriram Shantharam,
Jinghao Shi, Guru Prasad Srinivasa, and Lukasz Ziarek
Department of Computer Science and Engineering
University at Buffalo
maybe@blue.cse.buffalo.edu

ABSTRACT

One of the reasons programming mobile systems is so hard is the wide variety of environments a typical app encounters at runtime. As a result, in many cases only post-deployment user testing can determine the right algorithm to use, the rate at which something should happen, or when an app should attempt to conserve energy. Programmers should not be forced to make these choices at development time. Unfortunately, languages leave no way for programmers to express and structure uncertainty about runtime conditions, forcing them to adopt ineffective or fragile ad-hoc solutions.

We introduce a new approach based on *structured uncertainty* through a new language construct: the `maybe` statement. `maybe` statements allow programmers to defer choices about app behavior that cannot be made at development time, while providing enough structure to allow a system to later adaptively choose from multiple alternatives. Eliminating the uncertainty introduced by `maybe` statements can be done in a large variety of ways: through simulation, split testing, user configuration, temporal adaptation, or machine learning techniques, depending on the type of adaptation appropriate for each situation. Our paper motivates the `maybe` statement, presents its syntax, and describes a complete system for testing and choosing from `maybe` alternatives.

1. INTRODUCTION

All programmers must deal with uncertainty about runtime environments. For example, the most appropriate algorithm for a given task may depend on inputs that are unknown when the function is written, and its performance may depend on device-specific features that are impossible for the developer to anticipate. In other cases, real-world testing may be required to choose between several approaches, which has led to the popularity of split testing.

Uncertainty is especially problematic for mobile app programmers. Specific device features, such as accurate GPS location, may or may not be available. Networks come and go and their properties change: from fast and free Wifi links to slower metered mobile data connections. Energy may be plentiful or scarce, depending on the device's battery capacity and the user's energy consumption and charging patterns. These constantly fluctuating exogenous conditions make writing effective mobile apps particularly challenging.

```
if (plugged == false && batteryLevel < 10) {
  // Try to save energy
} else {
  // Don't try to save energy
}
```

Figure 1: **Typical Error-Prone Energy Adaptation.** The threshold is arbitrary and the attempt to conserve energy may succeed only at certain times, only for certain users, only on certain devices, or never.

Today programmers are forced to anticipate these changing conditions at development time and implement the required adaptation themselves. Figure 1 shows an example of an Android app attempting to adapt to the device's battery level by establishing regular- and low-battery code paths, with the latter attempting to save energy—possibly by utilizing a slower but more energy-efficient algorithm, computing an approximate result, or deferring the computation.

Unfortunately, this approach has several serious weaknesses. The most important is perhaps the least obvious: it is unclear that the different code paths achieve the desired result. There may be no differences between the alternatives (neither conserves energy), the alternative designed to conserve energy may actually consume more due to bugs or incorrect assumptions, or the outcome may depend on other factors not considered by the programmer, such as the type of network the device is currently using.

In addition, attempts at pre-deployment adaptation frequently produce arbitrary decision thresholds. Even if the two code paths in Figure 1 achieve the desired result, it is unclear what battery level threshold should trigger the energy-saving path, whether a single threshold will work for all users, and whether the threshold should depend on other factors such as how frequently the app is used.

Finally, the current approach to adaptation fails to support post-deployment testing. While it is possible to enable flexibility, runtime adaptation, and split testing using the languages currently used to program mobile systems, these tasks require writing large amounts of error-prone boilerplate code that retrieves settings from remote servers and adjusts values at runtime.

HotMobile'15, February 12–13, 2015, Santa Fe, New Mexico, USA

ACM 978-1-4503-3391-7/15/02 ...$15.00.
http://dx.doi.org/10.1145/2699343.2699361

```
maybe {
  // Try to save energy (Alternative 1)
} or {
  // Don't to save energy (Alternative 2)
}
```

Figure 2: **Example `maybe` Statement.** The programmer provides multiple alternatives. The system determines how to choose between them.

The root of the problem is that **today's languages force programmers to be certain at a moment when they cannot be: at development time.** While this problem is endemic to existing programming languages, when developing mobile apps it is magnified by the amount of variation developers must confront. Our solution is simple: (1) provide developers with a way to express *structured uncertainty*, and (2) use the resulting flexibility to enable a large array of downstream tools for resolving uncertainty by choosing from the alternatives provided by the developer.

In this paper, we present a novel a language construct for expressing structured uncertainty: the `maybe` statement. Unlike previous approaches to adaptation that relied on language support, `maybe` does not encourage programmers to provide more information about their app to allow the compiler to improve performance or guarantee correctness. Rather, `maybe` allows the programmer to express and structure uncertainty by providing two or more different *alternatives* implementing multiple approaches to runtime adaptation. Together, multiple alternatives can produce the same energy-performance or energy-accuracy tradeoffs described previously. Conceptually, `maybe` extends the process of compilation and optimization to include post-deployment testing while also enabling flexible adaptation that may produce per-user, per-device, or time-varying decisions.

Figure 2 shows how our earlier example can be easily rewritten using a `maybe` statement. Unlike the previous example, `maybe` does not rely on the developer to implement a decision process or correctly predict the effects of each alternative. Instead, the `maybe` system makes runtime choices about which alternative to use by measuring the tradeoffs produced by each alternative and (in this case) activating an energy-saving alternative when appropriate. When they are unsure what to do, all developers have to do is provide alternatives; the `maybe` system does the rest. `maybe` allows developers to respond to uncertainty with flexibility, which is used to enable testing-driven adaptation.

The rest of our paper is structured as follows. We begin by providing a more complete description of the `maybe` statement in Section 2. Section 3 describes several techniques for transforming development-time uncertainty into runtime certainty. We continue by describing several example use cases in Section 4, discussing the implications of the `maybe` statement in Section 5, and presenting related work in Section 6. We conclude in Section 7.

2. `MAYBE` STATEMENT SEMANTICS

To begin we provide an overview of the `maybe` statement's semantics describing how it allows developers to structure uncertainty. We refer to each of the values or code paths a `maybe` statement can choose from as an *alternative*.

2.1 Setting Variables

Variables can be used to represent uncertainty. Examples include an integer storing how often a timer should trigger

```
// Setting variables
int retryInterval = maybe 1-16;
String policy = maybe "auto", "quality", "perf";

// Function alternatives
@maybe
int myFunction(int a) { /* First alternative */ }
@maybe
int myFunction(int a) { /* Second alternative */ }

// Inlining evaluation code
maybe {
  ret = fastPowerHungryAlgorithm(input);
} or {
  ret = slowPowerEfficientAlgorithm(input);
} evaluate {
  return { "repeat": false,
           "score" : nanoTime() + powerDrain() }
}
```

Figure 3: **More `maybe` Statements**

communication with a remote server, or a string containing the name of a policy used to coordinate multiple code blocks throughout the app. Figure 3 shows examples of an integer that can take on values between 1 and 16, and a string that be set to either "auto", "quality", or "perf".

2.2 Controlling Code Flow

Code flow can also represent uncertainty. Examples include using multiple algorithms to compute the same result or multiple code paths representing different tradeoffs between performance, energy, and quality. Figure 2 shows the `maybe` statement in its simplest form, controlling execution of multiple code blocks. If multiple alternatives are specified, the system chooses one to execute; if only one alternative is specified, the system chooses whether or not to execute it. Single-alternative `maybe` statements can encapsulate or reorganize logic that does not affect correctness, but may (or may not) produce some desirable outcome.

Figure 3 shows several extensions of the `maybe` statement providing syntactic sugar. `maybe` function annotations allow uncertainty to be expressed at the function level, with the alternatives consisting of multiple function definitions with identical signatures. `maybe` statements that require custom evaluation logic can include an `evaluate` block as shown in the final example. `evaluate` blocks provide app-specific *a posteriori* logic to evaluate the selected alternative. The `evaluate` block must return a single JSON object with two components: (1) a positive integer `score`, with smaller being better; (2) and a boolean `repeat` indicating whether the system must use the same alternative next time. Hints and custom evaluation logic can also be applied to other types of `maybe` statements through annotations.

While it should be possible to nest `maybe` statements, it may require compiler support to provide guarantees about how `maybe` decisions are maintained across multiple code blocks. As we gain more experience with our rewrite-based prototype, described next in Section 3, we will revisit the question of nesting in future compiler-based `maybe` systems.

As a final remark, note that structured uncertainty is not randomness. Randomness weights multiple options statically—there is no right or wrong decision. In contrast, the `maybe` statement indicates that during any given execution one alternative may better than the others. The goal of the system is to determine which one.

3. FROM UNCERTAINTY TO CERTAINTY

While `maybe` allows programmers to specify multiple alternatives, ultimately only one alternative can be executed at runtime. Either a single, globally-optimal alternative must be identified, or a deterministic decision procedure must be developed. Before discussing options for adapting an app to its runtime environment, we first explain our runtime's support for `maybe` alternatives, including *a posteriori* evaluation and data collection. Then, we discuss how `maybe` testing enables a variety of different adaptation patterns.

3.1 Evaluating Alternatives

The optional `evaluate` block of a `maybe` statement allows programmers to provide app-specific *a posteriori* evaluation logic. However, in many cases, we expect that `maybe` statements will be used to achieve common objectives such as improving performance or saving energy. To streamline application development, our current system evaluates `maybe` statements without a `evaluate` block by measuring both energy and performance. In cases where one alternative optimizes both, that alternative will be used—although the decision may still be time-varying due to dependence on time-varying factors such as network availability. When alternatives produce an energy-performance tradeoff we are exploring several options, including collapsing both metrics into a single score by computing the energy-delay product (EDP) of each alternative, or allowing users to set a per-app energy or performance preference.

`evaluate` blocks can also record other information to aid adaptation. While the `score` value is used to evaluate the alternative, the entire JSON object returned by the `evaluate` block is delivered to the developer for later analysis. This allows `maybe` statements to be connected with end-to-end app performance metrics not visible on the device. We expect that some `evaluate` blocks may need to know which alternative was executed to compute a score—for example, if the two alternatives produce different quality output. We are exploring the use of automatically-generated labels to aid this process.

If a `maybe` alternative throws an error, the system will bypass the `evaluate` block and give it the worst possible score. By integrating a form of record-and-replay [5], it may be possible to roll back the failed alternative and retry another. While `maybe` is intended to enable adaptation, not avoid errors, the existence of other alternatives provides a way to work around failures caused by uncertainty. Fault tolerance may also encourage developers to use `maybe` statements to prototype alternatives to existing well-tested code.

A final question concerns when a `maybe` alternative should be evaluated. Some alternatives may require evaluation immediately after execution. Others may require repeated execution over a longer period of time to perform a fair comparison. As described previously, `evaluate` blocks can indicate explicitly whether or not to continue evaluating the alternative, and we are determining how to make a similar choice available to `maybe` statements without `evaluate` blocks. In addition, `evaluate` blocks can store state across multiple alternative executions allowing them to evaluate not only micro- but also macro-level decisions. In both cases, however, the `maybe` system allows developers continuous per-statement control over alternative choice and evaluation as described in more detail later in this section.

3.2 `maybe` Alternative Testing

We next describe the pre- and post-deployment testing that helps developers to design an *adaptation* policy, a strategy for ultimately selecting between alternatives. While the `maybe` system automates many of the tedious tasks normally associated with large-scale testing, we still provide ways for the developer to guide and control any step in the process.

3.2.1 Runtime control

To begin, we briefly outline how our Android prototype implements the `maybe` statement. We (1) rewrite each `maybe` conditional to an `if-else` statement controlled by a call into the `maybe` system and (2) generate a similar setter for each `maybe` variable. Variable values and code branches are now all under the control of a separate `maybe` service which can be deployed as a separate app or incorporated into the Android platform. It is responsible for communicating with the global `maybe` server to retrieve adaptation parameters for all `maybe`-enabled apps on the smartphone. When possible, we avoid interprocess communication during each `maybe` decision by caching decisions in the app, with the `maybe` service delivering cache invalidation messages when particular decisions change. The `maybe` service tracks when alternative decisions change, runs `evaluate` evaluation logic when appropriate, and returns testing results to the `maybe` server.

Because unexpected runtime variable changes could cause crashes or incorrect behavior, we only alter `maybe` variables when they are (re)initialized, not at arbitrary points during execution. If the app wants to enable periodic readaptation of certain variables, such as the interval controlling a timer, it can do so by periodically resetting the value using another `maybe` statement. This ensures that `maybe` variables only change when expected.

3.2.2 Simulation or emulation

Pre-deployment simulation or emulation may provide a way to efficiently evaluate `maybe` statements without involving users. Building simulation environments that accurately reflect all of the uncertainties inherent to mobile systems programming, however, is difficult. To complicate matters, `maybe` alternatives may depend on details of user interaction that are difficult to know *a priori*, particularly when new apps or features are being developed. So in most cases we believe post-deployment testing will be required.

However, pre-deployment testing may still be a valuable approach, particularly when a large number of `maybe` statements are being used. Since this can explode the adaptation space, simulations may be able to help guide the developer's choices of which `maybe` statements may have a significant impact on performance and should be evaluated first. Other `maybe` statements can be evaluated later or eliminated.

3.2.3 Split testing

Eventually code containing a number of `maybe` statements will be deployed on thousands or millions of devices. At this point, large-scale split testing and data-driven learning can begin. If the user community is large enough, it may be possible to collect statistically-significant results even for all possible permutations of `maybe` alternatives. For apps with a small number of users, or a large number of `maybe` statements, we can collect data for variations of one or several `maybe` statements while holding the rest constant. As an adaptation policy is designed and deployed for the statement

being tested, we begin to vary and measure the next group of `maybe` statements. Developers can observe and control the testing process through a web interface.

Each time a `maybe` statement is reached or `maybe` variable is set, the `maybe` system records:

- what `maybe` was reached;
- what alternative was used and why. This includes all environmental features used to make the decision, as well as any other available provenance information;
- what `evaluate` block evaluated the alternative, and the entire JSON object it returned, including the score;
- and a variety of other environmental and configuration parameters that the user permits access to: A user identifier; device and platform information; networking provider and conditions; location; battery level; and so on.

This dataset is periodically uploaded to the `maybe` server and used to drive the adaptation approaches discussed next.

3.2.4 Simultaneous split testing

While large-scale split testing is intended to provide good coverage over all possible sources of uncertainty we have discussed, it still normally requires that only one decision be made at any given time—implying that two alternatives may never be evaluated under identical conditions. For `maybe` statements, however, we are exploring the idea of performing *simultaneous* split testing. In this model the app forks at the top of the `maybe` statement, executes and scores all alternatives, and then continues with the outputs from the best alternative at the bottom of the `maybe` statement. On single-core devices this can be done in serial, while the growing number of multi-core smartphones provides the option of doing this in parallel. The benefit of this approach is that each alternative is executed under near-identical conditions. The drawbacks include the overhead of the redundant executions and the possibility for interference between alternatives executing in parallel.

3.3 `maybe` Endgames

The entire `maybe` approach is predicated on the fact that there does exist, among the alternatives, a right decision, even if it depends on many factors and uncertainties. We continue by discussing how the dataset generated by post-deployment testing can be used to determine how to correctly choose `maybe` alternatives at runtime.

3.3.1 Simple cases

In the simplest case, testing may reveal that a single alternative performs the best on all devices, for all users, at all times. In this situation, the `maybe` system may offer a way for the developer to immediately cease testing of that alternative and even automatically rewrite that portion of code to remove the `maybe` statement. However, it is also possible that the situation may change in the future when a new device, or Android version, or battery technology is introduced, and so the programmer may also choose to preserve the flexibility in case it is useful in the future.

The slightly more complicated case is when testing reveals that alternatives provide stable tradeoffs between energy and performance—one alternative always saves energy at the cost of performance. In this case the system only has to determine whether to prioritize energy or performance. While this decision seems simple, it is itself complicated by differences in battery capacity, charging habits, mixtures of installed apps, and the importance of the app to each user. However, the stability of the alternatives' outcomes means that once an energy or performance policy decision has been made, the choice of alternative has also been made.

3.3.2 Static adaptation

In the more complicated cases, testing reveals that the choice of alternative depends on some subset of the factors driving uncertainty in mobile systems programming. We break this group into two subsets, depending on whether the adaptation is time varying (dynamic) or not (static). We begin with the second, somewhat easier case.

If the alternative is determined through static adaptation then the correct decision is a function of some unchanging (or very-slowly changing) aspect of the deployed environment. Examples include the device model, average network conditions, the other apps installed on the device, or user characteristics such as gender, age, or charging habits. In this case it is possible that the correct alternative can be determined through clustering based on these features, and once determined will remain the best choice for a long time.

3.3.3 Dynamic adaptation

If the choice of alternative depends on dynamic factors such as the accuracy of location services, the amount of energy left in the battery, or the type of network the device is currently connected to, then it is possible that no single alternative can be chosen even for a single user. Instead, the `maybe` system allows developers to evaluate one or more strategies to drive the runtime alternative selection process.

Note that `evaluate` blocks are *not* intended to accomplish this kind of adaptation. First, they run after the `maybe` statement has been executed, not before. Second, per-`maybe` strategy defeats the flexibility inherent to the `maybe` approach and would devolve into the fragile decision-making we are trying to avoid.

Instead, the `maybe` system allows developers to experiment with and evaluate a variety of different dynamic adaptation strategies deployed in a companion library, with the decision guided by post-deployment testing. For example, if the performance of an alternative is discovered to be correlated with a link providing a certain amount of bandwidth, then that adaptation strategy can be connected with that particular `maybe` statement.

Observe that in some cases of dynamic adaptation, what begins as a `maybe` statement may end as effectively `if-else` statement switching on a static threshold—the same approach we attacked to motivate our system. However, through the process of arriving at this point we have determined several things that were initially unknown: (1) what the alternatives accomplish, (2) that a single threshold works for all users, and (3) what that threshold is. And by maintaining the choice as a `maybe` statement, they can continue adaptating as devices, users, and networks change.

Another benefit of this approach is that time-varying decisions can be outsourced to developers with expertise in the particular area affecting adaptation decisions. For example, by exposing an energy-performance tradeoff through a `maybe` statement, a developer allows it to be connected to a sophisticated machine learning algorithm written by an expert in energy adaptation, instead of their own ad-hoc approach.

3.3.4 Manual adaptation

In some cases even our best efforts to automatically adapt may fail, and it may be impossible to predict which alternative is best for a particular user using a particular device at a particular time. If the differences between the alternatives are small, then it may be appropriate to simply fall back to a best-effort decision. However, if the differences between the alternatives are significant then the `maybe` alternatives may need to be exposed to the user through a settings menu. Fortunately, information obtained through testing can still be presented to the user to guide their decision. Note that this requires labeling alternatives in a human-readable way.

3.4 Continuous Adaptation

Finally, even once a decision process for a particular `maybe` alternative has been developed, it should be periodically revisited as users, devices, networks, batteries, and other factors affecting mobile apps continue to change. To enable continuous adaptation, developers can configure `maybe` statements to continue to periodically experiment with alternatives other than the one selected by the alternative testing process. Changes in alternative performance relative to the expectations established during the last round of alternative testing may trigger a large-scale reexamination of that `maybe` statement using the same process described above.

4. EXAMPLE USE CASES

The `maybe` system is inspired by our frustrations building smartphone apps that confront the uncertainties inherent to mobile systems programming. In this section we describe several examples of how to use the `maybe` statement drawn from our own experiences.

4.1 PocketParker App

PocketParker [8] estimates parking lot availability by using the smartphone's accelerometer to detect users entering and leaving parking spots. To do this is an energy-efficient manner, we initially developed a custom activity recognition algorithm that duty-cycled the accelerometer to conserve energy. Towards the end of our development, Google released their own activity recognition API as a part of their Google Play Services framework. Based on several small-scale tests there was no clear winner when comparing the two algorithms, and so we decided to use Google's implementation to offload the maintenance burden. Supporting both algorithms, switching between them at runtime, and assessing the resulting impact on a larger user population would have required a significant amount of development effort.

Such runtime decisions fit naturally into the `maybe` framework. Instead of having to choose based on small-scale local testing, the `maybe` system can manage the transition from a mature but app-specific and expensive to maintain algorithm to a potentially-immature but canonical library implementation. As the library implementation improves and begins to out-perform the hand-tuned alternative, the `maybe` system can conduct repeated testing and move more users over to the library implementation. For some users or on some devices, the library implementation may never outperform the app-specific algorithm, in which cases `maybe` allows both alternatives to coexist safely while ensuring that each user enjoys whichever approach is most effective for them.

4.2 PhoneLab Conductor

PhoneLab is a large scale smartphone platform testbed at the University at Buffalo [9]. We leverage the Android `logcat` subsystem as a data collection mechanism—experiments log their data into a system-wide log buffer and we collect and upload this data on their behalf.

We developed an app called the PhoneLab Conductor for this purpose which provides a good example of custom `maybe` evaluation logic. The goal of our app is to collect data reliably while minimizing energy consumption, storage usage, and metered data usage. With the `maybe` statement branching between multiple policies for uploading data—such as always waiting until the user reaches a plug, or always initiating an upload once the storage allocated is 50% full—the evaluation logic would provide the worst possible score if data had been lost, or otherwise a score combining the multiple attributes the app is trying to minimize.

Due to the uncertainties we faced during development, we implemented a configuration interface that periodically retrieves parameters from our server and uses them to reconfigure variable components of the app. This allows us to control aspects of program behavior such as the amount of storage space we use on each device for logs, how often we check for updates, and how to decide whether to upload data. Several of these features have proven essential after deployment—for example, when an upload policy that worked previously abruptly stopped working on a newer Android version. Development of this app would have been considerably easier using `maybe`, which could automate the process of pushing policies to clients in an energy-efficient way, and enabling per-user goal-driven adaptation.

4.3 Navjack Sensing Platform

The Navjack project is exploring hijacking in-car navigation devices built from recycled smartphones [3] and deployed in personal vehicles to enable city-scale sensing. Volunteers install a device that acts as a dedicated navigation aid and car performance monitor but also continuously collects sensor data, utilizing the car's battery for power, the car's driver for maintenance, and inexpensive machine-to-machine (M2M) data plans for telemetry.

Navjack's goal is to produce high-quality data in response to queries while minimizing cellular data usage and car battery energy consumption. Many uncertainties specific to this app complicate the process. Some car cigarette lighters and USB charging docks provide power while the vehicle is off, while some do not. Users drive their vehicles for different durations and at different frequencies. Cellular coverage varies, altering the energy-per-bit required to offload data. All of these factors complicate post-deployment adaptation.

`maybe` statements in Navjack can be used to control the sampling rate, the set of sensors that are used, and the conditions under which uploads are attempted. This app provides an example of an adaptation state space that can potentially get quite large, and so it may provide a good chance to evaluate both our ability to perform pre-deployment simulations to reduce the state space and the success of post-deployment clustering techniques to identify salient user differences.

5. DISCUSSION

We have yet to determine how natural programmers will find the `maybe` statement. Encouragingly, `maybe` statements are similar to the ubiquitous `if-else` statement, and in

many cases can directly replace `if-else` statements that attempt runtime adaptation. To coordinate the adaptation of multiple code paths a single `maybe` variable can be used to control multiple `if-else` statements.

Overuse of the `maybe` statement may cause problems. If dependencies exist between `maybe` statements, the overall configuration space may expand exponentially, complicating post-deployment adaptation. Compile-time analysis may be required to detect dependencies between `maybe` statements and encourage programmers to limit their use of `maybe` to ensure that downstream optimization remains feasible.

`maybe` statements should not be used when adaptation can be refactored into a library. As an example, an app should not use `maybe` to decide which network interface to use when attempting to achieve a common objective, such as maximizing throughput. This adaptation should be refactored into a dedicated library, which might use its own `maybe` statements. Not only is the resulting codebase smaller, but the total number of `maybe` statements to test is reduced.

However, the `maybe` statement represents a fundamentally different approach to runtime adaptation than systems that rely on libraries because library development still requires development-time certainty. While library developers are more likely to be experts at the type of adaptation their library performs, we still believe that even the most skilled programmers will benefit from being able to express structured uncertainty. `maybe` allows all developers—including both app and library writers—to shed the burden of producing a single certain approach and instead write uncertain code containing the flexibility required to enable powerful data-driven approaches to post-deployment adaptation.

6. RELATED WORK

New systems such as EnFrame [12] reflect growing interest in managing uncertainty at the language level. EnFrame focuses on enabling programming with uncertain data, rather than the runtime adaptation enabled by `maybe`.

Aspect oriented programming (AOP) [6] aims to increase modularity through the separation of cross-cutting concerns. The programmer expresses cross-cutting concerns in stand alone modules, or aspects, which specify a computation to be performed as well as points in the program at which that computation should be performed. Fundamentally, the goals of AOP and the `maybe` statement differ, with AOP focusing on modularity and `maybe` focused on enabling adaptation by expressing uncertainty.

`maybe` shares similarities with language-based approaches to adapting energy consumption such as Eon [11] and Levels [7]. However, these approaches still require programmers to express certainty by associating code with energy states, rather than allowing the `maybe` system to determine which energy states are appropriate. `maybe` can also enable adaptation driven by goals other than energy management.

Attempts to enable more adaptive mobile systems date back to systems such as Odyssey [10]. However, a taxonomy of approaches to enabling adaptation on early mobile systems [1] reflects the focus of early efforts on incorporating adaptation into libraries that could be used by multiple apps. As we have pointed out previously, while adaptation libraries are useful, `maybe` statements can make them more powerful by allowing programmers to express uncertainty.

Recent approaches that allow mobile devices to effectively offload computation by automating client-cloud partitioning are also related to the `maybe` statement. Systems such as Tactics [2] and MAUI [4] used a variety of approaches to enabling this form of adaptation but are narrowly-focused on harnessing opportunities for remote execution. At present `maybe` focuses on single-device adaptation, but we are interested in exploring the ability to use uncertainty to distribute computation between multiple devices as future work.

7. CONCLUSION

To conclude, we have described the `maybe` statement: a new language construct allowing developers to express structured uncertainty at development time and for that uncertainty to be resolved through later testing and adaptation. We are in the process of building a prototype of the `maybe` system for Android smartphones.

Acknowledgments

Students and faculty working on the `maybe` project are supported by NSF awards 1205656, 1409367, and 1423215. The `maybe` team thanks the anonymous reviewers and our shepherd, Mahadev Satyanarayanan, for their feedback.

8. REFERENCES

[1] Badrinath, B., Fox, A., Kleinrock, L., Popek, G., Reiher, P., and Satyanarayanan, M. A conceptual framework for network and client adaptation. Mobile Networks and Applications 5, 4 (2000), 221–231.

[2] Balan, R. K., Satyanarayanan, M., Park, S. Y., and Okoshi, T. Tactics-based remote execution for mobile computing. In Proceedings of the 1st international conference on Mobile systems, applications and services (2003), ACM, pp. 273–286.

[3] Challen, G., Haseley, S., Maiti, A., Nandugudi, A., Prasad, G., Puri, M., and Wang, J. The Mote is Dead. Long Live the Discarded Smartphone! In Proc. 15th Workshop on Mobile Systems and Applications (ACM HotMobile 2014) (Feb. 2014).

[4] Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. Maui: making smartphones last longer with code offload. In Proceedings of the 8th international conference on Mobile systems, applications, and services (2010), ACM, pp. 49–62.

[5] Gomez, L., Neamtiu, I., Azim, T., and Millstein, T. Reran: Timing-and touch-sensitive record and replay for android. In Software Engineering (ICSE), 2013 35th International Conference on (2013), IEEE, pp. 72–81.

[6] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., marc Loingtier, J., and Irwin, J. Aspect-oriented programming. In ECOOP (1997), SpringerVerlag.

[7] Lachenmann, A., Marron, P. J., Minder, D., and Rothermer, K. Meeting lifetime goals with energy levels. In ACM Conference on Embedded Networked Sensor Systems (SenSys'07) (November 2007).

[8] Nandugudi, A., Ki, T., Nuessle, C., and Challen, G. Pocketparker: Pocketsourcing parking lot availability. In Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing (New York, NY, USA, 2014), UbiComp '14, ACM, pp. 963–973.

[9] Nandugudi, A., Maiti, A., Ki, T., Bulut, F., Demirbas, M., Kosar, T., Qiao, C., Ko, S. Y., and Challen, G. Phonelab: A large programmable smartphone testbed. In Proc. 1st International Workshop on Sensing and Big Data Mining (SenseMine 2013) (November 2013).

[10] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. Agile application-aware adaptation for mobility. In SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles (Saint Malo, France, 1997), pp. 276–287.

[11] Sorber, J., Kostadinov, A., Brennan, M., Garber, M., Corner, M., and Berger, E. D. Eon: A Language and Runtime System for Perpetual Systems. In ACM Conference on Embedded Networked Sensor Systems (SenSys'07) (November 2007).

[12] van Schaik, S. J., Olteanu, D., and Fink, R. Enframe: A platform for processing probabilistic data. arXiv preprint arXiv:1309.0373 (2013).