The future is cloudy: Reflecting prediction error in mobile applications

Brett D. Higgins[^] Kyungmin Lee[*] Jason Flinn[*] T.J.Giuli[†] Brian Noble[*] Christopher Peplin[†]
Arbor Networks[^] University of Michigan[*] Ford Motor Company[†]

***Abstract*—Mobile applications often predict the future to make decisions in the present. Although such predictions are inherently uncertain, applications typically assume that they are completely accurate. This assumption can lead to incorrect decisions resulting in unnecessary delays, wasted resources, or worse.**

Instead, prediction error should be a fundamental consideration in mobile systems. Applications should consider uncertainty when weighing alternatives. When one alternative is not clearly superior to another, *redundant strategies* are often appropriate, resulting in much better performance at a very modest cost.

To illustrate these ideas, we describe and implement several methods for quantifying uncertainty in mobile environments. Our system allows applications to explicitly weigh the tradeoff between the performance gained via redundancy and the cost of extra energy and cellular data resources spent, tailoring decisions to their relative importance. We adapt two systems to use this approach. Compared to both simple and adaptive strategies that do not reflect prediction error, our library improves application performance by up to a factor of two.

I. Introduction

Current mobile applications greatly overestimate their ability to predict the future. Unlike the desktop environment, many resources such as network availability, speed of network, power level can rapidly change due to the mobility nature of a mobile device. As circumstances change, mobile systems and applications adapt their behavior to take best advantage of their environments. These adaptive decisions are made based on *predictions* of the future—network performance and availability, expected computational loads, the presence and capabilities of support services in the infrastructure, etc.

Unfortunately, these predictions are rarely certain, yet they typically are used as if they were perfectly precise. Overconfidence in prediction leads to incorrect adaptations and lost opportunities, with consequences visible to the end user in terms of performance, power, and network costs. The problem is that mobile applications typically modularize their decision processes. First, an application calculates estimated values for quantities such as bandwidth and application compute needs. Even though the calculation of an estimated value takes into account underlying distributions and measurement uncertainty, the act of collapsing the information into a single scalar value means that the inherent notion of error in the underlying prediction is no longer captured. Second, applications use the estimated values to choose the single option that maximizes the difference between estimated benefit and estimated cost. As a result, even applications that select the optimal strategy based on the predicted values are bound to make wrong decisions at least some of the time—when reality does not match the predictions.

We argue that the system and applications should *explicitly* consider the uncertainty of their predictions when making decisions based on them, and they should use new information to reevaluate those decisions when necessary. In particular, considering prediction uncertainty throughout the decision process lets applications properly consider the benefit of *redundant strategies*. For example, a network selection application may decide to send the same data redundantly on multiple networks when it is quite unsure which network will provide the best response time. By using the first data to arrive and discarding the second, the application gets the best performance possible and masks the effects of the predictor uncertainty.

However, the advantages gained by employing redundancy come at a price. Sending data on two networks spends additional energy and potentially also spends cellular data, which for many users is limited to a fixed budget per month. From the perspective that resource conservation is of utmost importance, redundancy may appear to be obvious folly, since it always spends more resources than choosing the correct strategy from the start. In the face of uncertainty, however, the correct strategy cannot always be known. Faced with uncertainty, redundancy is a powerful mechanism that can spend resources to purchase an improvement in performance and a reduction in variability. In this way, redundancy can be seen as an insurance policy against inaccurate predictions.

Of course, the use of redundancy as a hedge against uncertainty is not a new idea. It is used in several other domains, including cloud servers [8] and route selection in wireless networks [1]. But such techniques have not been applied throughout the systems and application stack in a principled way, and we argue that mobile applications are missing substantial opportunities by not doing so.

To explore these ideas, we have built an API that applications can use to capture the uncertainty in their predictions and incorporate it into their decision-making. Our framework implements three different methods for making decisions while taking predictor uncertainty into account, each with different tradeoffs. Each explicitly weighs the importance of performance and resource conservation, employing redundancy only when spending resources to purchase better performance is likely to be cost-effective.

We have modified two applications to use our framework: network selection and speech recognition. Our experimental results show that there is significant benefit to be gained from redundancy when resources are sufficient to justify the tradeoff. Compared to the both the simple strategies that these applications typically employ, as well as adaptive strategies that consider performance and cost without considering predictor error, our framework improves application performance by up to a factor of two.

II. Example

We illustrate the ideas in this paper with a (very simple) motivating example. Consider two servers that can execute an offloaded computation, with the following known distributions on response time: server A takes 10 seconds half the time and 20 seconds otherwise, and server B takes either 12 or 22 seconds with 50-50 probability. A system that only considers non-redundant solutions would calculate the expected response time for each server (15 and 17 seconds, respectively) and execute the computation on server A. But, since the response times are independent, a system that considers redundant strategies would calculate the expected fastest response time from either server over the joint distributions as 13 seconds (the fastest response with equal probability will be 10, 10, 12, or 20 seconds).

Thus, if response time is the only consideration, redundant execution offers an expected benefit of 2 seconds over the best non-redundant solution. However, redundancy uses additional resources. A principled approach would balance the expected 2 second benefit against the 2x server resource usage and choose redundancy only if the value of improved latency exceeds the added resource cost.

In addition to improving expected response time, redundancy can help mask outlier behavior. Consider a distribution in which each server takes 10 seconds 99.9% of the time and 100 seconds 0.1% of the time. Redundant execution reduces the chances of the user experiencing the outlier behavior from 0.1% to 0.0001%. Cloud systems currently apply this well-known principle to reduce tail latency [8].

Finally, it is important to reevaluate decisions based on new information. While it is often unreasonable to modify applications to provide explicit notifications about progress, the absence of a response can provide valuable insight.

For instance, in the previous example, consider a system that made the decision to execute on one server (because the resource cost of redundant execution is high and outlier behavior is unlikely). Assume that the computation has run on that server for 11 seconds without a response. Now, the conditional probability distribution reveals that the expected completion time is an additional 89 seconds (the outlier behavior is essentially certain in this simplified example). A new evaluation at this point reveals that starting a computation on a second server is extremely likely to substantially reduce response time. Thus, the negative information embodied by a lack of response changes the decision about whether or not to employ redundancy. While timeouts or other failover methods can provide an ad-hoc solution, consideration of the conditional distributions provides a more general and precise method of handling unexpected delay.

This paper advocates for the *principled* use of redundancy in mobile applications. Too often, mobile systems eschew redundant strategies altogether by collapsing predicted distributions into expected values (as in the first example), or they do not consider negative information (as in the final example). They therefore miss many opportunities to improve the user experience.

III. Uncertainty and redundancy

Uncertainty and redundant strategies go hand-in-hand. The following conditions make redundancy attractive:

- There must be multiple strategies available to accomplish some task. The strategies must not interfere with each other, or such interference must be minimal.
- Future conditions on which strategy selection depends (e.g., resource supply and demand) must be uncertain.
- The benefits of better performance must outweigh the greater resource cost.

These conditions are present in the cloud computing domains; for instance, Google systems mask variable component response time with hedged requests that initiate a redundant request to a second server if the first server does not respond quickly [8]. Redundancy is also used extensively in wireless networks to compensate for unreliable links between nodes [1] where transmissions can be scheduled to minimize interference.

These conditions are also present in mobile application design. Specifically, non-interfering strategies exist in network selection (data may be sent over cellular and WiFi networks simultaneously) and in cyber foraging [9] (computation may be concurrently executed on both a mobile computer and a remote server). Compared to controlled data center environments, mobile environments are even more variable and future conditions are even more difficult to predict. While resource usage such as energy consumption is a very important concern in mobile computing, performance is also a paramount concern (otherwise, users would not buy the latest and greatest smartphone with the fastest and most power-hungry processor).

The paradox is that, in hindsight, redundancy is always wrong! Sending on an extra network always spends more energy than if the application had just chosen the best network to begin with. With perfect predictors, therefore, redundant strategies are never attractive. However, when predictions may be wrong, redundancy offers a performance benefit by giving applications an opportunity to hedge their bets. As the expected error increases, the performance benefit of redundancy improves.

IV. Related work

Many prior systems offload computation by migrating software components to remote servers. MAUI [7] partitions applications by running methods either locally or remotely. It monitors method runtime, energy, and network conditions and uses a global optimization to choose the partitioning that minimizes energy usage while keeping added latency under 5%. It starts a new computation only if it detects a remote failure via timeout or dropped connection. CloneCloud [5] partitions applications by running threads either locally or remotely. It profiles applications offline to generate optimal partitions for various network, CPU, and energy conditions. It measures these conditions when an application starts and chooses a partition via table lookup. Spectra [10] also runs software components either locally or remotely. It measures supply and demand of CPU, network, energy, and storage at runtime. It calculates an expected value for each and uses those values to decide which execution is best.

Chroma [4] builds on Spectra and so uses the same techniques. It selects from a wider set of tactics (strategies for partitioning and adapting the fidelity of applications). Additionally, Chroma provides a mechanism for executing redundant computation on multiple servers; however, it does not provide any policy for determining when redundancy

should be employed. It is precisely such a policy that is the focus of our work. Because Chroma does not implement such a policy, it is similar to Spectra in that its decisions consider only the expected values of resource supply and demand. Like Chroma, Slingshot [30] runs computation on multiple servers, but it always instantiates redundant computations when multiple servers are available. Thus, it does not use estimates and provides no policy for deciding whether or not to execute redundantly.

Odessa [28] partitions stages in pipelined processing by executing each stage locally or remotely. It also varies data parallelism within a stage by adding or removing threads; all threads of a stage run on a single computer. It makes such decision by considering average processing time, CPU speeds, and network and bandwidth estimates.

In summary, none of these prior solutions, nor any other offloading system of which we are aware, provide policies for considering when to execute redundantly. This is because all partition computation by running components either on the mobile device or on one or more servers. The two systems that execute redundantly on more than one server (Chroma and Slingshot) provide a mechanism but no policy for deciding whether or not to execute redundantly. Thus, our work would improve all of these prior solutions by providing a policy that identifies when to employ redundancy.

Current mobile devices (e.g., iOS and Android) generally use a trivial network selection policy: prefer WiFi over cellular when available, unless the user has disabled WiFi. This assumes WiFi will outperform cellular, which is not always the case, especially when the user is moving and with modern cellular technology such as LTE.

Several prior systems send data concurrently over multiple mobile networks. R-MTP [24], pTCP [14], and mTCP [34] attempt to aggregate throughput by striping data across multiple networks. Note that striping is *not* a redundant strategy; any given unit of data is only sent on a single network. Striping hinders low latency (the focus of our work) because the aggregate transmission experiences the latency of the slowest network. Careful scheduling is required because a single network estimated poorly is enough to stall the entire pipe when ordered delivery is required; the designers of FatVAP eschew striping within flows for exactly this reason [18]. Before our modifications, Intentional Networking [13] identified latency-sensitive traffic, but it sent such traffic over only a single network. Compared to these prior systems, our work is the only one to explicitly consider redundantly sending the same piece of data over multiple networks to lower latency.

Network measurement is itself a well-studied area, encompassing a variety of passive [17], [19], [20], [31] and active [15], [16], [29] techniques. Most of these techniques produce spot measurements of available bandwidth and latency without providing a measure of the uncertainty in those measurements. Jain and Dovrolis [16] characterize available bandwidth with a variation range, but offer no policy for using it to make decisions.

Redundancy is used in cloud computing [2], [8] to reduce tail latency of compute jobs by sending them to multiple servers. Routing in ad-hoc sensor networks has long used multiple redundant paths to provide fault tolerance in the face of packet loss and node or path failure (e.g., [1]). These prior systems use domain-specific algorithms to determine when to employ redundancy. We focus on the more general domain of mobile applications, and therefore offer more generic algorithms for quantifying prediction error and deciding when to employ redundant strategies.

V. Design

We define a *predictor* to be a software component that estimates the future value of some quantity. For example, commonly-used predictors in mobile computing estimate resource supply, such as network bandwidth and latency; failure probability, such as the chance of moving out of range of a hotspot; and application or user behavior, such as the length of a speech utterance.

Applications use the output of predictors to select from among different *strategies*, which we define to be different means of accomplishing the same task. Strategies may be non-redundant, such as sending data over a cellular or WiFi network, or redundant, such as sending the same data over both types of network.

Adding redundancy through the simultaneous execution of multiple strategies tends to improve performance since the application can use the result of the fastest strategy to complete. However, executing multiple strategies increases resource utilization since each new strategy consumes network bandwidth, battery energy, etc. A redundant strategy should therefore be used when its expected benefits in terms of improved performance exceed its expected costs in terms of additional resource usage.

Our library considers response time, energy consumption, and cellular data usage in deciding which strategy to employ. In order to compare quantities expressed in different units, it assumes that the current relative importance of the metrics is specified through importance factors given by the user. For example, an importance factor of 0.0001 expresses the following tradeoff: "I am willing to spend 10 J or 10 KB to avoid 1 second of waiting." We set these factors using the goal-directed adaptation of Odyssey [26] and IMP [12] in which feedback is used to raise or lower the importance of energy and data usage depending on how current usage relates to a budgeted allocation.

The *weighted cost* of a strategy is calculated by multiplying the raw values for time, energy, and data usage with their respective importance factors and summing the weighted values together. The best strategy is the one that has the lowest weighted cost.

Note that when uncertainty is low and measurements are probably accurate, a non-redundant strategy should be preferred. This is because a decision that prefers one non-redundant strategy over the others is likely to be correct. On the other hand, when prediction uncertainty is high, the value of redundancy is greater because the likelihood of choosing the wrong strategy increases. Hedging bets by choosing multiple strategies has a greater likelihood of significantly improving application performance.

There are many possible methods for incorporating uncertainty in decision-making, each of which has strengths and weaknesses. Since we were not sure which method would work best in mobile computing, we implemented three methods in our library. The first two, which are described in Sections V-A and V-B, directly measure prediction error by comparing the measured value of a quantity with the

previously-predicted value. The sequence of error measurements collected over time is used to predict the uncertainty of future measurements. In the third method, described in Section V-C, our library measures decision error by comparing the strategy selected by the application with the strategy it should have taken.

So far, we have described only how our library makes an initial decision to employ a redundant or non-redundant strategy. Such decisions should be reevaluated when new information arrives in the form of updated predictions or even in the form of negative information such as the lack of response from a remote server. Section V-D describes how our library performs reevaluation.

A. Brute force

Our first method quantifies predictor error by measuring it over time and maintaining an empirical distribution. For each new measurement, our library calculates relative error of a prediction as the measured value divided by the previously predicted value. A relative error greater than one indicates that the last prediction underestimated the true value, whereas an error sample less than one indicates that it overestimated. The set of relative errors observed over time forms the empirical distribution of predictor error.

In order to incorporate the error distributions into a redundancy decision, our library calculates the expected completion time and resource costs for each non-redundant strategy as a weighted sum over the joint error distributions of the predictors. For each redundant strategy, the completion time at each point in the joint distribution is the minimum of the completion time of any component strategy and the resource costs are the sum of the resource costs of all component strategies. These values are likewise calculated as weighted sum over the joint error distributions. Our library combines the individual values to calculate a weighted cost for each strategy.

A number of possible methods exist for storing distributions; e.g., storing all samples forever, binning samples, random sampling, etc. We note that predictor error is constantly changing; e.g., a quantity such as network bandwidth that was unpredictable seconds ago may have since settled. Thus, our library ages samples using an exponential weighted moving average so that the most recent error samples are given greater weight. Aging also limits the number of samples stored; older samples are removed after they drop beyond a threshold where their impact on the calculation is negligible.

The expected advantage of the brute force method is simplicity and high accuracy; the expected disadvantage is computational cost. For instance, with a Nexus One phone, decisions for the applications in Section VI-C can take hundreds of milliseconds. Our library mitigates this cost by moving brute force calculations off the critical path. For instance, a network selection application can make an initial non-redundant decision about which network to use, then asynchronously run the slower redundancy decision to decide whether to use an additional network. Additionally, the predictor error distributions change slowly, so error calculations are cached and re-used for several decisions.

B. Probabilistic error bounds

Our second method measures and ages samples of predictor error in the same way as the brute force method, but it uses probabilistic bounds rather than calculations over an empirical distribution. Compared to the brute force method, use of probabilistic error bounds is less accurate but has substantially lower computational costs.

Previous work in network queuing uses Chebyshev bounds to give the worst-case error in packet delay prediction [11]. We initially tried this approach but found the resulting bounds too loose to be useful. Instead, our library uses a Student's-t distribution over the observed relative error samples to construct a statistical *prediction interval* ($\alpha = 0.05$) on the mean error value. Whereas a *confidence interval* represents the likelihood that the true value of a sampled population lies within a computed range, a prediction interval represents the likelihood that the *next* sample of the quantity falls in that range. Since we are attempting to make decisions based on observations of quantities such as network bandwidth and latency, whose true values are neither known nor fixed, the prediction interval fits our purpose more closely.

Given the calculated error bounds, we must decide how to use them to make a decision. First, we can use the bounds on all predictors to calculate the resulting bounds on time, energy, and cellular data; for example, the minimum network transfer time occurs at the upper bound of bandwidth and the lower bound of latency. Then, if no two non-redundant strategies have overlapping time bounds, this means that, with the chosen confidence, redundancy does not offer benefit. In this case, we simply choose the best non-redundant strategy.

If any non-redundant strategies' time bounds do overlap, we then evaluate redundancy as follows. First, we identify the best non-redundant strategy. Next, we calculate the maximum possible benefit from redundancy and the minimum additional cost, and we choose redundancy if the benefit exceeds the cost. Since a redundant strategy's completion time is the time of its fastest non-redundant strategy, redundancy provides the most benefit when the supposedly best non-redundant strategy turns out to be worse than expected and the supposedly worse non-redundant strategy turns out to be better than expected. Therefore, the maximum benefit from redundancy is the minimum time of the redundant strategy subtracted from the maximum time of the best non-redundant strategy. Similarly, the additional cost of redundancy is calculated using the bounds of the additional strategies used, not including the best non-redundant strategy.

Because the error bounds method captures the distribution of error with less precision than the empirical distribution of the brute force method, and because the computed bounds are fairly loose, the error bounds method tends to overestimate the benefit of redundancy. The choice to use the upper bound on redundancy benefit and the lower bound on redundancy cost also biases this method towards redundancy.

C. Bayesian estimation

Our last method quantifies uncertainty by calculating the *posterior distribution* of the actual predicted quantities with the predictor values as *evidence*. This process is known as Bayesian estimation. It centers around the Bayesian view of statistics and probability, which provides a way to formulate sound beliefs about an environment given some prior beliefs

and new evidence. Prior work applied this technique to computation offload but only considered the bandwidth of one network and did not consider redundancy [32].

In our case, the prior beliefs are our historical measurements of predicted quantities such as bandwidth and latency, and the new evidence is the most recent decision made by the library; e.g., "based on the predicted bandwidth and latency, WiFi is better than cellular." From Bayes' Theorem, the *posterior* distribution of bandwidth and latency is proportional to the product of the *prior* bandwidth and latency distributions and the *likelihood* of the evidence (the predictors' decision) given the prior observations:

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{p(\text{evidence})}$$

where $p(\text{evidence})$ is a normalizing factor that ensures that the posterior distribution is a PDF (that it sums to 1).

It may not be immediately clear how the Bayesian formulation incorporates the *error* of the predictors, as was explicitly done in the prior two methods. To clarify this, we draw an analogy to a textbook example of Bayesian reasoning: a simple problem of weather forecasting, answering the question, "will it rain tomorrow?" In this problem, the prior is the historical frequency of rain, and the evidence is a forecast (a prediction of "rain" or "no rain"). Since weather forecasting is imperfect, a forecast will sometimes predict rain when none occurs, and sometimes it will predict clear skies when rain occurs. This inaccuracy is captured in the likelihood measure, which answers the questions: "when it *does* rain, how often does the forecast agree?" and "when it *doesn't* rain, how often does the forecast agree?"

The likelihood measure answers similar questions for our library; e.g., "when bandwidths and latencies have been such that WiFi was better, how often has WiFi been predicted to be better?" Thus, the Bayesian formulation captures the accuracy of the predictors, but in a more abstract sense than the previous methods: by measuring how the inaccuracy affects the correctness of the resulting decisions.

In order to compute posterior distributions, our library keep empirical records of the information specified in Bayes' Theorem: the prior distributions, the likelihood of the predictor decisions given the priors, and the normalizing factor $p(\text{evidence})$, which in our case is the strategy frequency regardless of the priors. These values are binned and kept as histograms to reduce storage costs.

Each time we obtain a new measurement, we check the predictors' decision (e.g., which network is best) *just before* the measurement was taken. We update the $p(\text{evidence})$ histogram with the indicated strategy, and we update one histogram in the likelihood array, using the new measurements to choose which histogram to update. Note that this is in keeping with the Bayesian formulation of the problem discussed above; we are tracking how often the predictors have agreed with the next observed measurements, for various values of those measurements.

Given the posterior distributions of predicted quantities, our library calculates the expected benefit and cost of redundancy in the same way as in the brute force method. That is, it iterates over the joint prior distributions and uses the information that it has tracked to compute the posterior joint distribution. It then uses the joint posterior distribution to compute the weighted sums as before, and it uses the same method to decide whether the benefit of redundancy is worth the cost.

Our library ages samples so that older observations have decreasing impact. In the Bayesian method, it accomplish this by aging the bins in each prior distribution's histogram; each bin in the histogram is now a weight rather than a count. When a new sample is added to a bin, the weight for all other bins in that distribution decreases.

It may at first appear that the Bayesian method is equivalent in performance to the brute force method, since it still produces a joint distribution over all predictors. In practice, however, the likelihood distribution is sparsely populated; when our library calculates the weighted sum, it avoids including the portions of the joint prior distribution not included in the likelihood distribution, because it knows they are zero. Thus, whereas the the brute force method has complexity $O(n^m)$ with m predictors having n samples each, the Bayesian method has complexity $O(m \times n)$, or the total number of samples across all predictors.

D. Reevaluation from new information

The arrival of new information can change the decision about which strategy to employ. New information may arise from an explicit event such as association with a new access point. Alternatively, it may arise from the lack of an expected event, such as not receiving a response from a remote server within some time.

Our library reevaluates a decision by repeating the prior calculations using conditional probability distributions for the predictors, based on the new evidence. Regardless of the evaluation method chosen, the library restricts consideration to only the portion of the distributions that fit within observed conditions. For instance, given that a remote operation has already taken at least `x` seconds, our library, using the brute force or error bounds method, considers only the portion of the joint error distribution that agrees with a duration of at least `x` (e.g., the region of bandwidths and latencies that would cause the operation to take at least `x` seconds). Similarly, if the library is using the Bayesian approach, it only considers the regions of the prior distribution that agree, which produces a conditional posterior distribution.

The notion of what information is sufficient to trigger reevaluation is application-specific, so our library allows an application to trigger a reevaluation whenever it observes a sufficient change in the environment. In the case of an explicit event, deciding to trigger reevaluation is straight-forward. However, it can be challenging to determine when the lack of information is sufficient to justify a reevaluation, so our library provides additional support for this case.

On application request, the library will calculate the "tipping point" at which the conditional error distribution calculations will cause the decision to change. It does a binary search over the range of a given predictor, noting the strategy chosen at each point. The search terminates when the tipping point is found within a specified granularity. Since the strategy selection is cleanly separable from the details of how a method is implemented, this binary search works for all three methods without modifications.

Note that the binary search strategy assumes that only one predicted value changes based on new information. More

Application → Library		Library → Application	
`make_strategy(time_fn, energy_fn, data_fn, s_arg)`	→ strategy	`time_fn(s_arg, c_arg)`	→ double
`make_redundant_strategy(strategy[])`	→ strategy	`energy_fn(s_arg, c_arg)`	→ double
		`data_fn(s_arg, c_arg)`	→ double
`create_predictor()`	→ predictor		
`get_predictor_value(predictor)`	→ double		
`add_measurement(predictor, measurement, new_prediction)`			
`set_predictor_bound(predictor, bound, {UPPER,LOWER})`			
`clear_predictor_bounds(predictor)`			
`register_strategies(strategy[], enum method)`	→ evaluator		
`choose_strategy(evaluator, c_arg)`	→ strategy		

This table shows the API for incorporating uncertainty into application decisions.

TABLE I: Application programming interface

complex applications for which this assumption is not valid may simply schedule periodic reevaluations instead.

VI. Implementation

A. Library

We built a library that makes it easy for mobile applications to consider predictor error in their decisions. Table I shows its API. An application first specifies the non-redundant strategies from which it will decide. Each specification includes strategy-specific functions that calculate the time, battery energy, and data usage of executing the strategy given predicted quantities. Next, the application creates redundant strategies; each such strategy is composed of two or more non-redundant strategies that will be executed concurrently. Our library will calculate the time and resource usage of the redundant strategies using one of the three methods in the previous section.

An application next specifies the predictors, each of which is responsible for tracking a single quantity such as network bandwidth or latency. It may specify any combination of custom predictors or the generic predictors that we describe in the next section. Our library tracks the error in the predictions. Thus, each time an application makes a new measurement, it passes the measurement, the previous prediction for the quantity, and the new prediction for the quantity to the library. In turn, when the library runs its evaluation method, it uses its error-adjusted estimates when invoking the application-supplied strategy functions, which those functions obtain via the predictor interface.

The predictor interface also allows applications to bound the distribution of possible values based on new information. For instance, if a cyber foraging application has not received a response in time `x`, it sets a lower bound of `x` on the response time. The library will use only the portion of the distribution with values greater than `x` to estimate response time.

Finally, an application passes the list of strategies to our library and receives back an *evaluator*. An evaluator is a handle, which the application can ask our library to choose the best strategy. The evaluation method is specified when the evaluator is created, and the evaluator provides a interface for choosing a strategy and scheduling deferred reevaluation. The evaluator also takes responsibility for caching previously computed decisions and invalidating the cached decisions when new measurements arrive.

B. Predictors

We provide several generic and customizable predictors for applications to use. We briefly describe here the specific prediction algorithms that we drew from the literature.

Network bandwidth and latency. The network predictor implements a flop-flip filter [19] to smooth estimates of network quality. Separate latency and bandwidth estimates are calculated via linear interpolation over varying transmission sizes. The predictor provides separate estimates for each available network (WiFi and cellular).

Network dwell time. This predictor estimates the time the mobile computer will remain in range of the current WiFi access point. It fits the dwell time to a Weibull distribution via maximum likelihood estimation using the methods reported by [21]. The effect is that it has low confidence in the reliability of a newly-associated WiFi network, but the confidence increases over the time of association. It provides per-AP estimates given sufficient samples for the current AP; else, it provides a generic prediction.

Application compute time. This predictor estimates the time to complete a application-specific computation using an approach similar to that of Narayanan et al. [25]. It posits a linear relationship with computation parameters and performs a regression to generate the best fit. This is application-specific, so the application must specify the value of the regression parameters for each observation.

Application energy usage. This predictor uses the PowerTutor models [33] to estimate energy usage as a function of communication and computation. Similar to the previous predictor, the amount of computation and communication is a linear fit over application-specified parameters.

C. Applications

1) Network selection

Network selection allows data to be sent over either cellular or WiFi, as conditions warrant. Prior work often focuses on throughput and either sends data over a single network at a time or stripes data by sending different data over different networks. Our work focuses on user-facing activities, so latency, not throughput, is the relevant metric. In such scenarios, it may make sense to redundantly transmit the same data over multiple networks and use the first instance to arrive at the remote endpoint.

We modify Intentional Networking [13] (though several systems provide similar multi-network functionality). Our modified version estimates network bandwidth, latency, dwell time, and energy usage for each available network using the predictors described above. For each transmission, it calls into the library for a decision as to whether the transmission should be sent using WiFi, cellular, or both networks redundantly.

The decision of when to reevaluate is application-specific. Our Intentional Networking application reevaluates its choice of strategy whenever the mobile device associates with a new network or a network it is currently using fails. In addition, the strategy choice is reevaluated when a response is not received after a delay of two times the expected completion time. This avoids the wasted time and resources of spurious retransmissions when networks are stable, but still allows for quick recovery from real failures. The reevaluation is repeated every 200 ms until the decision changes or the transfer completes. The effect of reevaluation is a graceful response to failure in which a redundant transfer is started on a second network as the Intentional Networking application loses confidence in its initially chosen network.

2) Speech recognition

Speech recognition is a classic application of cyber foraging [3], the offloading of computation from a mobile client to a remote server. We modified the PocketSphinx [6] library to perform recognition on the mobile device, on a remote server, or concurrently at both locations.

This application uses the above predictors for network latency, bandwidth, and dwell time, as well as the predictors for application compute time and energy usage. The regression parameter for compute time and energy usage is the length of the spoken utterance. However, since recognition starts immediately and proceeds in parallel with the speaking of the utterance, the precise length of the utterance to be spoken is not known when the initial decision about where to execute recognition is made.

Therefore, this application also uses an application-specific predictor to estimate the length of the utterance using the utterance lengths that have been observed in the past. The output of this predictor is used to compute recognition time (measured from the point when the user finishes speaking to the time the recognized text is displayed) and energy (measured over the entire recognition).

In our experience, the upload bandwidth of both 3G and LTE networks is too low for remote execution—it is faster and more energy-efficient to run recognition locally. Thus, remote recognition takes place only when WiFi is available.

The speech application reevaluates decision in three cases. First, a decision to execute locally is reevaluated if the mobile device associates with a new WiFi network (since remote recognition may now make sense). Note that if the local recognition is likely to complete soon, starting a remote recognition may not make sense even if network quality is good. Second, the utterance may be significantly longer than predicted. The application uses the library's binary search feature to precalculate the tipping point for utterance size and changes its recognition strategy if the tipping point is reached (e.g., by starting a redundant remote execution). Finally, the application also uses the binary search feature to calculate a tipping point in case remote recognition takes longer than expected. This is the point at which a redundant local recognition is started.

VII. Evaluation

A. Experimental Setup

1) Testbed

We run all applications on the AT&T version of the Nexus One, running Android 2.3.4. We modified Android to allow the simultaneous use of WiFi and cellular networks. To ensure repeatable experiments, the phone connects to an isolated WiFi access point and a private Cisco MicroCell that is connected to AT&T's network. Since the MicroCell acts as a miniature cellular tower, our evaluation captures the effects of the cellular wireless medium on network bandwidth and latency. We emulate network conditions by passing all traffic through a computer that inserts delays with the netem [23] network emulator and throttles throughput using the Linux Advanced Routing and Traffic Control tools [22]. We run servers for each application on a Dell Optiplex GX270 desktop with a 2.8 GHz Pentium 4 processor and 1GB DRAM.

We measure energy usage by applying PowerTutor's Nexus One power model [33]. We measure cellular data usage by reading the number of bytes sent and received through the Linux `sysfs` interface.

2) Scenarios

Performance, energy, and cellular data usage have varying importance to different users. We capture this through the importance factors described in Section V. Our experimental results are thus given as a weighted cost that is derived from using these factors to equate the diverse metrics of time, energy consumption, and data usage. One can contextualize these factors in order to give them concrete meaning; e.g., "I am willing to spend X Joules to save Y seconds of waiting."

We evaluate our results in four different scenarios, each of which reflects a different tradeoff between performance (interactive delay) and resource consumption (energy and data usage). The first is the `no-cost` scenario, in which performance is the only consideration (energy and data usage are not considered). Table III shows the time-energy tradeoff in the remaining three scenarios: `low-cost`, `mid-cost`, and `high-cost`. In the `low-cost` scenario, a user is willing to spend 100 Joules to save 1 second of waiting. With that amount of energy, the user could watch 36 seconds of streaming video, or the user could leave the phone idle for an additional 37 minutes. Based on the average behavior (including both idle and usage periods) given by a recent study [27], this would equate to 6 minutes of extended battery lifetime. In the `mid-cost` scenario, 1 second of waiting equates to 3.6 seconds of video watching, 3.75 minutes of idle time, or 36 seconds of average usage. In the `high-cost` scenario, 1 second of interactive delay purchases only 0.36 seconds of video playing or 22.5 seconds of battery lifetime on a completely idle phone. This already seems like a very poor tradeoff, so we did not investigate higher cost scenarios.

Similarly, in the `low-cost` scenario, 1 second of waiting time equates to saving 100 KB of data usage. In the `mid-cost` and `high-cost` scenarios, 1 second of waiting equates to 10 KB and 1 KB of data, respectively.

3) Trace-driven evaluation

We use trace-driven emulation to provide experimental repeatability and allow for meaningful comparison between

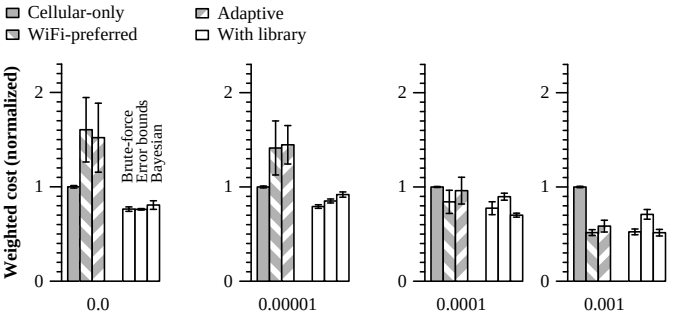
Scenario	WiFi Coverage	WiFi session length		3G Bandwidth (Kbps)			
				Downlink		Uplink	
		Median	Max	Median	Max	Median	Max
Walking	69%	41 sec	5 min	137	737	48	454
Driving	27%	7 sec	2 min	368	1200	40	74

TABLE II: Details of the network traces used for evaluation

		Battery life reduction		
Usage	Expected battery life	Low-cost (100 J)	Mid-cost (10 J)	High-cost (1 J)
Idle	125 hours	37.5 min	3.75 min	22.5 sec
Average	20 hours	6 min	36 sec	3.6 sec
Heavy	2 hours	36 sec	3.6 sec	0.36 sec

This figure shows the approximate battery life impact of spending various amounts of energy to save 1 second of delay in different evaluation scenarios.

TABLE III: Performance-energy tradeoff in various scenarios



(a) No cost (b) Low cost (c) Mid cost (d) High cost

Fig. 1: Network selection, walking trace

strategies. We gathered these traces by continuously measuring the bandwidth and latency via active probing to a server at the University of Michigan. We use two traces that illustrate different mobility scenarios: a walking trace gathered on our institution's campus, and a vehicular networking trace used in the evaluation of Intentional Networking [13]. The vehicular trace illustrates a highly-mobile scenario with challenging network dynamics. The walking-trace has longer associations with WiFi access points and better overall connectivity. Table II details the network characteristics for the two traces.

When running benchmarks, we replay the traces on the emulation computer, which throttles bandwidth and delays packets for each network according to the conditions observed. When no WiFi or cellular coverage is observed in a trace, the throttling computer drops the connection. The Android OS typically discovers the network disconnection after several seconds. Since the collected traces are longer than our experiments, we use only the first portion of each trace.

B. Results

1) Network selection

To evaluate the effectiveness of redundancy in network selection for small, interactive transfers, we constructed an experiment in which the application executes a series of request-response exchanges using Intentional Networking. These requests and responses are of random sizes, normally distributed about 1 KB and 4 KB, respectively. The application pauses between requests for a think time uniformly distributed between 10 and 30 seconds.

We compare the results using our library to two simple strategies: only use cellular, or use WiFi when available and fall back to cellular when it is not. WiFi-preferred is effectively the strategy currently employed by Android and iOS.

For comparison, we also built an adaptive strategy that uses the same cost weights and formulas as our library to calculate the time, energy, and data cost in order to select either cellular or WiFi (when available). This is a standard adaptive strategy that treats predictions as oracles and does not incorporate uncertainty. It also does not reevaluate its decisions based on new information.

Each experiment runs for 20 minutes. We report the total weighted cost over this 20-minute interval. All results are the mean of 5 runs; error bars are 95% confidence intervals. Application performance is the time the user spends waiting for a response to each request. We first discuss the results for the brute force method, and later compare the three evaluation methods.

Figure 1 shows weighted cost results for the walking trace, normalized to the weighted cost of the cellular-only strategy. Redundancy provides the most benefit in the `no-cost` scenario, when performance is of utmost importance. Compared to a cellular-only strategy, the user spends 24% less time waiting when redundancy is employed. The reduction in waiting time due to redundancy is greater than a factor of two compared to both the WiFi-preferred and the adaptive strategies.

It may seem surprising that the slower cellular-only strategy provides better performance than the WiFi-preferred strategy that mobile devices commonly employ. The impact of WiFi failure detection delay accounts for this gap; even though it often provides better performance, WiFi is less reliable when the user is moving. A few long delays due to failure detection and failover when the mobile device moves out of range of an access point dwarf the performance difference between cellular and WiFi.

The `low-cost` scenario shows similar results to the `no-cost` scenario. However, since energy and cellular data now have non-zero weight, the WiFi-preferred and adaptive non-redundant strategies show improved results. Our library provides an improvement of 21% over cellular-only and 44% over WiFi-preferred and adaptive strategy.

The utility of redundancy drops as resource conservation becomes more important. In the `mid-cost` scenario, the strategies that prefer to send less cellular data have improved considerably. Our library's redundant strategies are equivalent within experimental error to the best non-redundant strategy (WiFi-preferred). However, for the brute force and Bayesian methods, both the average and the variance are still lower than WiFi-preferred, due to redundancy reducing the effect of the outliers caused by WiFi failover delay. This contributes to a

less variable user experience.

In the `high-cost` scenario, the brute force and Bayesian strategies are equivalent within experimental error to the best non-redundant strategy (WiFi-preferred). Thus, our library achieves benefit from redundancy when it is available and correctly chooses to avoid redundancy when it is not helpful.

Note that since the cellular-only strategy never uses WiFi, a user could potentially achieve slightly better energy usage by disabling WiFi. However, the energy savings from doing so are minimal (1% reduction in weighted cost in the `low-cost` scenario to 4% in the `high-cost` scenario). Any potential gains are therefore dwarfed by the poor performance of the cellular-only strategy. Further, the user would lose the opportunity to employ WiFi when it is best (as it is in the `high-cost` scenario).

We also ran our experiment with the driving trace. Unlike the walking trace, opportunistic WiFi provides little benefit, because most sessions are short and unreliable. Thus, cellular only is best in all scenarios.

The unshaded bars in Figures 1 compare the relative weighted cost realized by the three redundancy evaluation methods. The results are mixed when comparing our three methods. The Bayesian method is often best in higher cost scenarios and the brute-force method is often best in lower cost scenarios. The brute force method has higher computational complexity, which may result in higher weighted cost. The Bayesian method incorrectly sends some transfers non-redundantly in the lower cost scenarios because it is less accurate than the brute-force method.

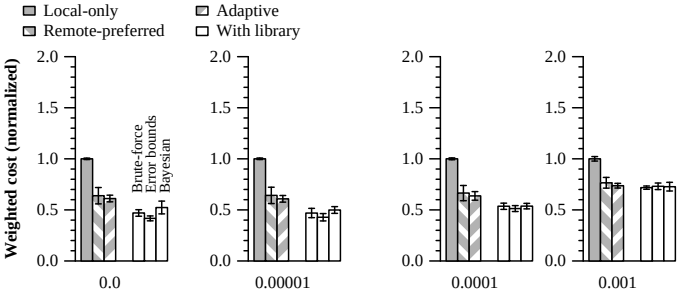
The error bounds method performs the poorest in general. It spends resources more readily, resulting in higher weighted cost in the scenarios where resource conservation is more important. Since the bounds are loose, the method is prone to having an inflated view of predictor uncertainty and thus naturally tends more towards redundancy.

2) Speech recognition

To evaluate the effectiveness of redundancy in a speech recognition application, we selected 20 utterances ranging in length from 1 to 8 seconds. We simulate a user speaking at the original sampling rate of 20 KB/sec, looping over the 20 utterances at 30-second intervals until 20 minutes have elapsed. The performance metric is the recognition delay, or the interval between the time when the user finishes speaking and the time with the transcribed text is available. We report the total weighted cost, which reflects only recognition delay and energy usage in these experiments because cellular usage is negligible.

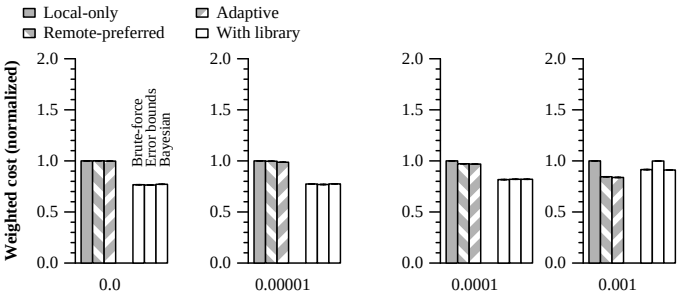
The two non-redundant strategies are to execute locally or to prefer remote recognition using only WiFi. We compare against an adaptive strategy that considers time and cost but not predictor error. The redundant strategy executes both locally and remotely.

Figure 2 shows the results for speech recognition on the walking trace. Note that the approximate break-even point for redundancy is even higher in this application—that is, redundancy still shows significant benefit in the `mid-cost` scenario. Speech recognition generally takes longer than a single small request/response exchange, so there is more delay to be saved for a given cost.



(a) No cost (b) Low cost (c) Mid cost (d) High cost

Fig. 2: Speech recognition, walking trace



(a) No cost (b) Low cost (c) Mid cost (d) High cost

Fig. 3: Speech recognition, server load trace

In the `no-cost` scenario, redundant strategies are generally best, with performance improvement of 23–35% over the remote-preferred and adaptive non-redundant strategies, and improvement of more than a factor of two over the local-only strategy. The exception is the Bayesian method, which shows high variance and thus is equivalent within error to the remote-preferred and adaptive non-redundant strategies.

In the `high-cost` scenario, the remote-preferred strategy dominates due to its reduced energy usage, and executing a local redundant recognition is not worth the cost. Our library correctly selects the non-redundant strategy in this scenario.

Note that, as was the case in network selection, no single strategy is always best, and our library reduces weighted cost by up to a factor of two compared to the simple strategies in scenarios where they are not best. Also, even though local-only is never the best strategy in any scenario, using local execution in addition to remote execution does provide benefit in reducing the uncertainty of remote execution, as it mitigates the impact of failover delays.

We also ran this application with the driving trace. Similar to the previous application, the poor network quality causes local recognition to always dominate remote recognition. Our library achieves equivalent performance by correctly selecting the local recognition in almost all cases.

Finally, we ran the speech application in a scenario designed to show the impact of server load. In this experiment, a high-quality WiFi connection is always available, but we add a background load of concurrent clients requesting speech recognition. Concurrent recognitions are performed by separate event-based server processes that compete for the CPU resource on the server. We vary the number of concurrent requests according to a Poisson distribution, with the average

load increasing from 6 to 10 concurrent clients during the course of a 10 minute experiment. Since the network is stable, this experiment shows how our library reacts to changes in recognition time due to server load and varying utterance lengths.

As Figure 3 shows, our library reduces recognition delay in the `no-cost` scenario by 23% compared to the local-only and remote-only strategies. The adaptive non-redundant strategy also performs poorly in this scenario because it frequently makes incorrect decision due to the difficulty in predicting future server load. Our library achieves improved performance by hedging its bets with redundancy. Our library achieves similar benefit in the `low-cost` and `mid-cost` scenarios by choosing local recognition in periods where the server load is likely to be high, remote recognition in period where the server load is likely to be low, and hedging its bets when there is uncertainty about which strategy is best. In the `high-cost` scenario, remote recognition is always superior. However, our library sometimes hedges its bets via redundancy. Therefore, it outperforms the local-only strategy but under-performs the remote-only and adaptive strategies (it saves time but uses more energy).

For this application, the error bounds method performs the best on the network trace. The brute force and Bayesian methods are generally equivalent within experimental error and best on the server load trace. As with the previous application, the Bayesian method is prone to underestimating uncertainty and choosing redundancy less readily. The brute force method again has high computation complexity.

3) Discussion

Overall, we observe significant benefit from redundancy: user wait time is reduced by up to a factor of two. Further, our simple back-of-the-envelope estimations suggest that sufficient resources are available quite often for the average user. When redundancy offers no benefit, our library usually approximates the best non-redundant strategy. The one exception is the `high-cost` scenario for speech recognition with CPU load: in this case, the library overestimates the inherent variability and sometimes chooses redundancy inappropriately.

VIII. Conclusion

Current mobile applications greatly overestimate their ability to predict the future. Overconfidence leads to mistakes—a strategy that is optimal for the predicted values can still be incorrect because the predictions on which the choice was based turn out to be wrong. We argue that applications should consider uncertainty throughout the entire decision process, and we provide a library with three methods for doing so. When uncertainty is correctly reflected in decisions, redundant strategies that hedge against that uncertainty are often best.

References

[1] Akkaya, K., and Younis, M. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks 3*, 3 (2005), 325–349.

[2] Ananthanarayanan, G., Ghodsi, A., Shenker, S., and Stoica, I. Effective straggler mitigation: Attack of the clones. In *Proc. NSDI* (April 2013), pp. 185–198.

[3] Balan, R., Flinn, J., Satyanarayanan, M., Sinnamohideen, S., and Yang, H.-I. The case for cyber foraging. In *the 10th ACM SIGOPS European Workshop* (September 2002).

[4] Balan, R. K., Gergle, D., Satyanarayanan, M., and Herbsleb, J. Simplifying cyber foraging for mobile devices. In *Proc. MobiSys* (June 2007).

[5] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A. CloneCloud: Elastic execution between mobile device and cloud. In *Proc. EuroSys* (April 2011).

[6] CMU Sphinx. http://cmusphinx.sourceforge.net.

[7] Cuervo, E., Balasubramanian, A., ki Cho, D., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. MAUI: Making smartphones last longer with code offload. In *Proc. MobiSys* (June 2010), pp. 49–62.

[8] Dean, J., and Barroso, L. A. The tail at scale. *Communications of the ACM 56*, 2 (February 2013), 74–80.

[9] Flinn, J. *Cyber Foraging: Bridging Mobile and Cloud Computing*. Morgan and Claypool Publishers, September 2012.

[10] Flinn, J., Park, S., and Satyanarayanan, M. Balancing Performance, Energy, and Quality in Pervasive Computing. In *Proceedings of the 22nd International Conference on. Distributed Computing Systems* (July 2002).

[11] Garroppo, R., Giordano, S., Pagano, M., and Procissi, G. On traffic prediction for resource allocation: A Chebyshev bound based allocation scheme. *Computer Communications 31*, 16 (Oct. 2008), 3741–3751.

[12] Higgins, B. D., Flinn, J., Giuli, T. J., Noble, B., and Watson, C. P. D. Informed mobile prefetching. In *Proc. MobiSys* (June 2012).

[13] Higgins, B. D., Reda, A., Alperovich, T., Flinn, J., Giuli, T. J., Noble, B., and Watson, D. Intentional networking: Opportunistic exploitation of mobile network diversity. In *Proc. MobiCom* (September 2010), pp. 73–84.

[14] Hsieh, H. Y., and Sivakumar, R. A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. In *Proc. MobiCom* (September 2002), pp. 83–94.

[15] Hu, N., and Steenkiste, P. Estimating available bandwidth using packet pair probing. Tech. Rep. CMU-CS-02-166, School of Computer Science, Carnegie Mellon University, September 2002.

[16] Jain, M., and Dovrolis, C. Ten fallacies and pitfalls on end-to-end available bandwidth estimation. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement* (October 2004), pp. 272–277.

[17] Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J., and Towsley, D. Inferring TCP connection characteristics through passive measurements. In *Proc. INFOCOM* (March 2004), pp. 1582–1592.

[18] Kandula, S., Lin, K. C.-J., Badirkhanli, T., and Katabi, D. FatVAP: Aggregating AP backhaul capacity to maximize throughput. In *Proc. NSDI* (April 2008), pp. 89–103.

[19] Kim, M., and Noble, B. D. Mobile network estimation. In *Proc. MobiCom* (July 2001), pp. 298–309.

[20] Lee, H., Kim, S., Lee, O., Choi, S., and Lee, S. Available bandwidth-based association in ieee 802.11 wireless LANs. In *Proc. MSWiM* (October 2008), pp. 132–139.

[21] Lee, K., Lee, J., Yi, Y., Rhee, I., and Chong, S. Mobile data offloading: How much can wifi deliver? In *Proc. CoNEXT* (December 2010), pp. 536–550.

[22] Linux Advanced Routing and Traffic Control. http://lartc.org/.

[23] Linux Foundation. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.

[24] Magalhaes, L., and Kravets, R. Transport level mechanisms for bandwidth aggregation on mobile hosts. *IEEE International Conference on Network Protocols* (2001).

[25] Narayanan, D., Flinn, J., and Satyanarayanan, M. Using history to improve mobile application adaptation. In *Proc. WMCSA* (August 2000), pp. 30–41.

[26] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. Agile application-aware adaptation for mobility. In *Proc. SOSP* (October 1997), pp. 276–287.

[27] Oliver, E., and Keshav, S. An empirical approach to smartphone energy level prediction. In *13th International Conference on Ubiquitous Computing* (September 2011), pp. 345–354.

[28] Ra, M.-R., Sheth, A., Mummert, L., Pillai, P., Wetherall, D., and Govidan, R. Odessa: Enabling interactive perception applications on mobile devices. In *Proc. MobiSys* (June 2011).

[29] Ribeiro, V., Riedi, R., Baraniuk, R., Navratil, J., and Cottrell, L. pathChirp: Efficient available bandwidth estimation for network paths. In *4th Passive and Active Measurement Workshop (PAM 2003)* (April 2003).

[30] Su, Y.-Y., and Flinn, J. Slingshot: Deploying stateful services in wireless hotspots. In *Proc. MobiSys* (June 2005), pp. 79–92.

[31] Vasudevan, S., Papagiannaki, K., Diot, C., Kurose, J., and Towsley, D. Facilitating access point selection in IEEE 802.11 wireless networks. In *Proc. IMC* (October 2005), pp. 293–298.

[32] Wolski, R., Gurun, S., Krintz, C., and Nurmi, D. Using bandwidth data to make computation offloading decisions. In *2008 IEEE International Symposium on Parallel and Distributed Processing* (April 2008), pp. 1–8.

[33] Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., and Yang, L. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. CODES+ISSS* (October 2010), pp. 105–114.

[34] Zhang, M., Lai, J., Krishnamurthy, A., Peterson, L., and Wang, R. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *Proc. USENIX ATC* (2004).