NWSLite: A General-Purpose, Nonparametric Prediction Utility for Embedded Systems

SELIM GURUN, CHANDRA KRINTZ, and RICH WOLSKI
University of California, Santa Barbara

Time series-based prediction methods have a wide range of uses in embedded systems. Many OS algorithms and applications require accurate prediction of demand and supply of resources. However, configuring prediction algorithms is not easy, since the dynamics of the underlying data requires continuous observation of the prediction error and dynamic adaptation of the parameters to achieve high accuracy. Current prediction methods are either too costly to implement on resource-constrained devices or their parameterization is static, making them inappropriate and inaccurate for a wide range of datasets. This paper presents NWSLite, a prediction utility that addresses these shortcomings on resource-restricted platforms.

Categories and Subject Descriptors: D.4.8 [**Operating Systems**]: Performance—*Modeling and Prediction*

General Terms: Measurement, Performance

Additional Key Words and Phrases: Computation offloading, CPU availability estimation, embedded systems, network performance estimation, prediction algorithms

ACM Reference Format:

Gurun, S., Krintz, C., and Wolski, R. 2008. NWSLite: A general-purpose, nonparametric prediction utility for embedded systems. ACM Trans. Embedd. Comput. Syst. 7, 3, Article 32 (April 2008), 36 pages. DOI = 10.1145/1347375.1347385 http://doi.acm.org/10.1145/1347375.1347385

1. INTRODUCTION

The lifetime of batteries is a critical factor in the design of mobile, resource-constrained devices. Current battery technology however, is unable to increase capacity, significantly without increasing size and weight. This trend is unacceptable given the increasing demand by consumers for smaller, lighter devices. Instead, software techniques are needed that reduce energy consumption and increase battery life.

Authors' address: Selim Gurun, Chandra Krintz, and Rich Wolski, University of California, Santa Barbara, Santa Barbara, California 93106.

Key to the efficacy of such techniques is the low cost of their use and accurate prediction of future application, workload, and resource behavior. Techniques that optimize energy use must estimate how the device will be used (demand) and what resources will be available to it (supply) *in the future*, to determine which optimization to apply and when. If the estimates are incorrect (inaccurate) or the application of the optimization introduces significant overhead, the techniques may be unable to extend battery life or actually shorten it.

One popular and important energy optimization technique is remote execution (aka computational offloading) [Flinn et al. 2001; Rudenko et al. 1998, 1999; Kremer et al. 2001; Li et al. 2001]. Remote execution extends the computational power and battery life mobile devices by off-loading parts of the execution from a battery-powered devices to a wall-powered, more capable, system. Remote execution has the potential to significantly increase the utility of devices by enabling execution of a wide-range of resource-intensive applications, e.g., augmented reality, natural language translation, feature recognition, collaborative computing, in a mobile environment.

Remote execution requires accurate prediction and low overhead computation of the estimations. A remote execution system must predict the cost of performing local and remote execution to determine when off-loading a computation will require less energy than performing it locally.

To predict the required parameters (i.e., network bandwidth, latency, local, and remote CPU supply), extant systems employ statistical techniques [Rudenko et al. 1999; Flinn et al. 2001; Flinn and Satyanarayanan 1999; Balan et al. 2003]. These prediction algorithms are parameterized and statically hand-tuned for a specific dataset, decreasing the effectiveness of predictors when the characteristics of input data changes. Moreover, these techniques are computationally intensive and require floating-point calculations, which can consume significant battery power. Since these computations are performed by the optimization system *on the device*, their cost must be low enough to be amortized by the optimized execution. Unfortunately, extant approaches to predicting resource supply and demand do not consider the cost of the estimation technique itself.

In this paper, we present a novel alternative to resource prediction for mobile, resource-constrained devices, called *NWSLite*. NWSLite is a low-cost, yet, highly accurate prediction service that is an extension of the network weather service (NWS), a resource performance measurement and prediction toolkit originally developed for scheduling high-performance, scientific applications in computational Grid [Foster and Kesselman 1998] environments [Swany and Wolski 2002; Wolski et al. 1999; Berman et al. 1996; Sucu and Krintz 2003; Spring and Wolski 1998]. NWSLite is a modification to the NWS forecasting model in a way that reduces its resource consumption footprint to enable its use in a mobile setting.

NWSLite can be incorporated by users into any mobile framework that uses prediction. It makes nonparametric, light-weight, forecasts of any resource for which measurement values can be supplied. As such, we can use it for prediction of CPU load, memory availability, and network bandwidth and latency, as well as file I/O and execution time of an application's operations (tasks).

In this study, we empirically compare both the accuracy and cost of NWSLite to the original NWS and extant prediction algorithms. We analyze these performance characteristics for a wide range of applications and resources: application execution time, availability, wired-network bandwidth and latency, and wireless bandwidth. Our results show that NWSLite enables prediction accuracy that in many cases significantly exceeds that of the predictors to which we compare. In addition, it consumes significantly fewer computational resources than its predecessor and enables more effective remote execution that was previously possible.

In the following sections, we describe the use of resource performance prediction to facilitate remote execution. We then describe the prediction methodologies to which we compare our work. Next, we detail the design and implementation of NWSLite in Section 3 and present an empirical evaluation of its efficacy both in terms of prediction accuracy and resource consumption (Section 4). In Sections 5 and 6, we discuss our remote execution scenarios that use NWSLite. Q2 Finally, we conclude in Section 7.

2. EXTANT PREDICTION ALGORITHMS USED IN EMBEDDED SYSTEMS

Prediction of resource availability and performance is a widely studied field of research. In this section, we overview representatives of common resource prediction methods that are employed for various uses by embedded devices. We focus on techniques that are online, require no modification to the application, and that are executed *on the device* itself, for which the overhead of the approach is as important as the accuracy it achieves. We describe each prediction strategy in the context of the particular resources (CPU, network bandwidth, etc.) for which they are used.

2.1 CPU Availability

Systems employ CPU availability prediction to estimate the CPU time a process or subprocess task consumes [Flinn et al. 2001; Grunwald et al. 2000]. These predictions are used by the execution environment to guide task scheduling and processor scaling decisions [Weiser et al. 1994; Govil et al. 1995; Pering et al. 1998; Kremer et al. 2001; Flinn et al. 2001; Grunwald et al. 2000]. A common technique for estimating CPU load is one that gathers load statistics via various operating system utilities and interfaces, such as `vmstat` and `top` in UNIX. CPU estimation techniques range from very simple to complex and thus vary in agility, overhead, and accuracy. Agility is the degree to which a prediction utility can react to and adjust for variance in measured, history data.

PAST scheduling [Weiser et al. 1994] assumes that the CPU load in the next interval will be the same as the most recent CPU load measurement. This forecaster is very agile since it immediately responds to changes in CPU load. However, such a response can have a negative effect on accuracy when recent CPU spikes are outliers (*noise*) and short-lived, i.e., not good estimates of future behavior.

To overcome such limitations, other CPU prediction techniques filter out noise using more sophisticated techniques. The Odyssey prediction system

represents such systems. Odyssey estimates CPU availability by first assuming that CPU cycles are evenly distributed among all processes. It then uses an exponential decay technique (i.e., a smoothing filter) to filter out noise. The Odyssey CPU prediction model is:

$$S_{cpu} = \frac{P}{N+1} \tag{1}$$

where P is the processor clock speed, N is the number of runnable processes, and S_{cpu} is the available CPU cycles. Odyssey uses a smoothing filter to estimate the number of processes in the next interval:

$$N_{t+1} = \alpha N_t + (1-\alpha)n(p) \tag{2}$$

In this equation, n is a function of observed number of processes in the current interval and defined as:

$$n(i) = \left\{ \begin{array}{ll} n_r - 1 & \text{If } p \text{ is runnable} \\ n_r & \text{Otherwise} \end{array} \right\}$$

where n_r is the number of runnable processes at time t.

Q3 The AVG_n policy [Weiser et al. 1994] is another popular CPU prediction technique that directly decays the measured CPU load over the last k intervals. Since this policy is simply an extension to PAST policy, it inherits the same
Q4 weaknesses (i.e., static, parameterized). AVG_n policy is less agile than PAST, but it is more resilient to noise in the network. There are also several other CPU load prediction techniques that are based on observation heuristics [Govil et al. 1995]. Such techniques, however, are less general, since they have many parameterized heuristic rules designed to optimize performance on the particular workload that they are intended for.

A more recent study [Sinha and Chandrakasan 2001] indicates that a single, parameterized method may not be the best choice across different workloads. In their study, Sinha et al. compared four CPU load-prediction techniques, including exponential smoothing, moving averaging, least mean squaring, and a purely probabilistic technique called expected workload state, using three real workloads. Their results showed that least mean squaring was better than the others, on average. However, the best predictor varied from one workload to another [Sinha 2001].

2.2 Network Latency and Bandwidth

Many embedded systems employ prediction techniques for network latency and bandwidth. Two common uses of such techniques are task scheduling for distributed devices and computation offloading. Computation offloading is a technique in which the system executes processes or tasks on more capable or wall-powered computer systems to conserve the battery power or extend the capability of mobile, resource-constrained devices [Flinn et al. 2002; Noble et al. 1997; Kim and Noble 2001; Balan et al. 2003].

To estimate network latency, extant prediction systems use passive observations of RPC packets to compute the round-trip time and throughput of

network [Noble et al. 1997]. Since, network performance is highly variable, noise can severely degrade the accuracy of network bandwidth estimations. To improve accuracy, other prediction systems use an exponential smoothing filter much like that used above for CPU [Noble et al. 1997]:

$$new = \gamma(measured) + (1 - \gamma)\, old \tag{3}$$

The value of the exponential decay factor (γ) determines the agility of the method. A larger value increases the responsiveness, but decreases the technique's ability to filter out noise. Thus, the accuracy of the method is highly dependent on the choice of the parameter. Since network latency and bandwidth exhibit different performance characteristics, users must identify multiple parameterizations (γ) for the filter function of each. Moreover, for a single metric (latency or bandwidth), the filter requires different parameterizations for *different* network technologies to achieve the best accuracy.

To overcome the limitations of a large number of parameterizations and instability, researchers have developed a network performance estimator that implements two exponential smoothing functions in a single forecasting system [Kim and Noble 2001], a so-called flip-flop predictor. The parameters used by this predictor are commonly at opposite ends of the spectrum to capture the benefits of both agility and smoothing. Both predictors execute concurrently. However, the estimator uses the one with the larger parameter (that enables more smoothing) as long as the approximate standard deviation of the predicted value is in a predetermined range with respect to the smoothed mean. The estimator switches to the agile version otherwise. This design has an important advantage over previous models as it is more accurate and can adapt more effectively to dynamic changes in the system.

2.3 Power Consumption

Power consumption is another important resource in mobile, embedded device. As such, its measurement as well as the prediction of remaining battery life has been the focus of much research [Kremer et al. 2001; Li et al. 2001; Rudenko et al. 1999; Krintz et al. 2004]. Many studies depend on parameterized, static performance models to estimate power consumption [Kremer et al. 2001; Li et al. 2001].

Our focus is on online techniques for power estimation. One such representative system is the remote processing framework (RPF) [Rudenko et al. 1999]. RPF predicts task power consumption online to determine whether the task should be executed locally on the device or remotely on a wall-powered server, i.e., whether computational offloading should be performed. RPF collects history data on the power consumption of previous tasks using a battery monitor on the device and uses it to predict the power consumption of future tasks. To estimate task power consumption, RPF uses this smoothing filter:

$$f_{n+1} = (1 - \alpha) * \frac{\sum_{i=n-k}^{n} v_i}{k} + \alpha * f_n \tag{4}$$

where f_i is the forecasted value, v_i is the measured value, and i is the measurement index. α and k determine how conservative the forecaster is: A

small k combined with a large α will result in higher responsiveness to recent changes.

Note that the RPF smoothing filter (Equation 4) is the same as the equation for CPU prediction prediction (Equation 2) when $k = 1$. In addition, the smoothing filter is the same as the bandwidth and latency prediction function in Odyssey (Equation 3), when $k = 1$ and $\alpha = 1 - \gamma$.

2.4 Application CPU Demand

The CPU demand of an application is highly dependent on the nature of the application itself. However, when no application-specific information is available or its collection is infeasible, prediction systems can estimate CPU demand using application history logs. The prediction system described in Narayanan and Satyanarayanan [2003] employs such a methodology. This methodology is popular and likely to be successful for embedded devices, since it does not require any effort by the user or application programmer, access to program source code, or no modification to the program.

In such systems, an online, learning, predictor maintains program-specific coefficients that are used to model the CPU demand of the application for a particular input dataset. Computing the *initial values* of coefficients unfortunately requires off-line training. However, once the initial values are set, the system updates the coefficients using recursive least-squares regression with exponential decay (LSQ). Given the characteristics of exponential decay, more weight is given to the recent observations.

LSQ can efficiently predict the value y when it is dependent on a set of parameters x, such that $y = Ax + w$, and w is the measurement error or noise. The general formula for recursive LSQ to estimate CPU load of tasks is:

$$\begin{aligned} A_k &= A_{k-1} - P_k\left\{x_k x_k^T A_{k-1} - x_k y_k\right\} \\ P_k &= \left\{P_{k-1} - P_{k-1}x_k\left[\alpha + x_k^T P_{k-1} x_k\right]^{-1} x_k^T P_{k-1}\right\}/\alpha \end{aligned}$$

where α is the decay factor and y_k is the measurement at time k. In the equation above, y_{k+1} is predicted by $A_{k+1}x_k$. The P_k matrix is commonly referred to as the *history* or *filtering factor* [Young 1984].

This technique performs well for augmented reality applications – a popular application domain for mobile devices. Such programs render pictures as a camera scans a set of scenes. Since scenes commonly overlap, their transitions are smooth. That is, the resource consumption behavior for the generation of a scene is similar to that of a neighboring scene. As a result, the performance data varies smoothly from scene to scene, enabling a prediction system that uses exponential smoothing to produce accurate predictions of CPU demand. As mentioned previously, a limitation of recursive least-mean squaring is that numerical computation errors can accumulate after each recursion-causing algorithm to become unstable and diverge [Bottomley and Alexander 1991].

3. NWSLITE

All extant prediction methodologies require user-specified parameterizations to forecast the cost of various resources. Users must identify the appropriate

parameters through empirical evaluation or using a complex, off-line learning process. Unfortunately, the parameters are specific not only to the executing application, but also to individual tasks within an application. As a result, the parameterization may not work well across applications or even across the tasks of a single application. There are also methods that mitigate this problem by requiring more user or application feedback [Flinn 2001]. Moreover, existing systems use a number of different prediction strategies (each requiring training and parameterization) for different resource types (e.g., CPU, network performance, and power consumption).

Our approach to the problem of resource prediction employs a different methodology. Specifically, it is one that is *nonparametric, automatic, adaptive, and agnostic of resource type and application behavior*. That is, we employ a single system that makes accurate predictions of any resource type for any application—without requiring application modification or participation by users for parameterization and off-line training. Moreover, our system is appropriate for resource-constrained, mobile, systems, i.e., it consumes few device resources to make accurate predictions. The system is called *NWSLite*.

NWSLite is an extension of the network weather service (NWS) [Wolski 1998], a freely available toolkit [NWS], originally developed for the *computational grid* [Foster and Kesselman 1998; Berman et al. 2003]. The computational grid is a computing paradigm for the development of software systems that enables dynamic acquisition of resources from a heterogeneous and nondedicated resource pool. Grid systems are high-performance, large-scale, distributed systems that require applications to adapt to the dynamically changing systems on which they are executed, as well as to highly variable resource performance. To extract performance from these systems, application schedulers must use predictions of future resource behavior to determine how the application can best use the available resources.

The NWS operates a distributed set of performance sensors, from which it periodically, and unobtrusively, collects performance measurements. The sensors apply a set of statistical forecasting techniques to individual performance histories and generate forecast reports for the resources being monitored. The NWS disseminates these reports via a number of different APIs in near real-time [Wolski et al. 1999]. Currently, the NWS provides sensors for end-to-end TCP/IP bandwidth and latency, available CPU and memory, battery power, and disk storage, and is used in a large number of different grid technologies.

NWS prediction uses a mixture-of-experts approach to prediction, instead of relying on a single model. It implements a large set of models, each having its own parameterization. Given a performance history of observed measurement values, it generates a forecast for each measurement. NWS ranks each predictor by computing the prediction errors (the difference between measured and forecasted values). Each time a forecast is requested, NWS recalculates the ranking across all predictors using the most recent history and chooses the most accurate model. The ranking of the predictors are done using the mean squares of the prediction errors. However, NWS allows the user to use other metrics (i.e., mean absolute percentage error). The implementation of NWS that we extended uses the 24 prediction models, shown, in Table I.

Table I. NWS Forecasters and the Approximate Costs of Each[a]

	Name	Average cost
1	**Last Value**	0
2	**Running Mean**	3
3	**5% Exp Smooth**	3
4	10% Exp Smooth	3
5	15% Exp Smooth	3
6	**20% Exp Smooth**	3
7	30% Exp Smooth	3
8	40% Exp Smooth	3
9	50% Exp Smooth	3
10	75% Exp Smooth	3
11	90% Exp Smooth	3
12	5% Exp Smooth, with 0.1% trend	10
13	10% Exp Smooth, with 0.1% trend	10
14	15% Exp Smooth, with 0.1% trend	10
15	20% Exp Smooth, with 0.1% trend	10
16	30% Exp Smooth, with 0.1% trend	10
17	Median Window 31	88
18	**Median Window 5**	16
19	Sliding Median Window 31	124
20	Sliding Median Window 5	26
21	30% Trimmed Median Window 31	106
22	30% Trimmed Median Window 51	169
23	Adaptive Median Window 5-21	171
24	Adaptive Median Window 21-51	455

[a]We show cost in column three as the number of floating-point operations performed.

This mixture-of-experts method achieves its accuracy by employing a wide range of statistical models, each of which may be most appropriate at a given time, for a given resource. Last value simply uses the last measurement as a prediction of the next measurement. Consequently, last value is very responsive to sudden changes. Unfortunately, last value is very susceptible to noise in measurements. Running mean keeps a running tabulation of the average measurement and uses that as a prediction at each time step. Exponential smoothing predictors use different parameterizations of Equation (3). Median predictors exercise a median filter over the data series. Some of these filters are enhanced using a sliding window, and α-trimming, which is obtained by sorting the values and removing a fraction from the high and low ends. By tracking the prediction error of each predictor, NWS identifies the most appropriate predictor at runtime and dynamically switches to that predictor.

Figure 1 gives the pseudocode for NWS forecaster selection. Given a history of measurement values (a trace), a forecast is generated for each, using each forecaster, over all the trace, up to the current value. In addition, NWS uses different windows (configured at compile time) of previous data and records the winning predictor for each. The winner predictor is the one that has the least prediction error across all window sizes. For example, if an exponential smoothing predictor is the most accurate predictor at one point in time and conditions change so that another predictor becomes the most accurate, the system will choose the latter predictor as winner, only if the change is persistent

```
for each window size (including the entire history)
        for each forecaster implemented in NWS
                forecast over current window size using
current forecaster
                record aggregate prediction error for
current forecaster
        end for
        record forecaster with lowest aggregate error
for this window size
end for
        choose forecaster and window size with lowest
aggregated error and make final forecast using it
```

Fig. 1. Pseudocode for NWS forecast selection.

enough to cause the aggregated error of the latter to be smaller than the former one. A detailed discussion of NWS forecaster selection is given in [Wolski 1999, 2003; Wolski et al. 1999].

The mixture-of-experts method that NWS employs also has other important advantages. First, even though the individual NWS models may be parametric, the overall system is not. The only input to the system is the measurement history, i.e., the NWS is agnostic of the resource to which the measurement belongs. Second, NWS can easily adjust itself to changes in the characteristics of the data series by switching to another model. Third, it can be used on any type of data for which measurements can be made. There is no distinction between CPU availability and network bandwidth, for example.

To illustrate how NWS can better adapt than parametric models, we are going to use an example, in which, we use CPU demand measurements that we collected on a Pentium III laptop, while a user was navigating in a 3D scene using a rendering application. As the user navigates, each change in viewpoint (a trace step) triggers a rendering task whose CPU demand is strictly dependent on the number of polygons that are visible from the user's viewpoint. Since the viewpoint has to follow a certain track, the future CPU demand is mostly predictable as a function of previous demand [Narayanan and Satyanarayanan 2003]. In our example, we use castle as 3D scene, and radiosity as application. We describe both the scene and application fully in Section 4.1.

Figure 2 shows the CPU demand measurements of castle. The CPU demand measurements indicate a significant amount of variation in task execution time (5 to 50 ms). There is a strong pattern in CPU demand. For example, from trace step 200–600, the CPU demand decreases regularly and then, after a few steep changes, it stays flat for most of the next 400 steps. We exercise NWS and its individual models on this trace.

Figure 3 compares the error performance of NWS to the individual forecasting models that NWS integrates. The horizontal axis shows the square root of mean squared prediction error. The first error bar shows the error rate if we had an oracle to choose the best of the 24 NWS forecasters before each prediction. The next bar shows the error rate of NWS, which dynamically chooses the forecaster depending on past error rate. The remaining 24 error bars show the error rate of each individual model. The NWS slightly exceeds (typically, it is

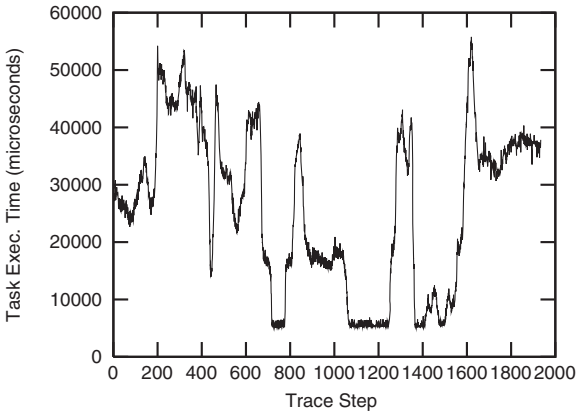


Fig. 2. Measured task execution times as user navigates in castle (a 3D scene).

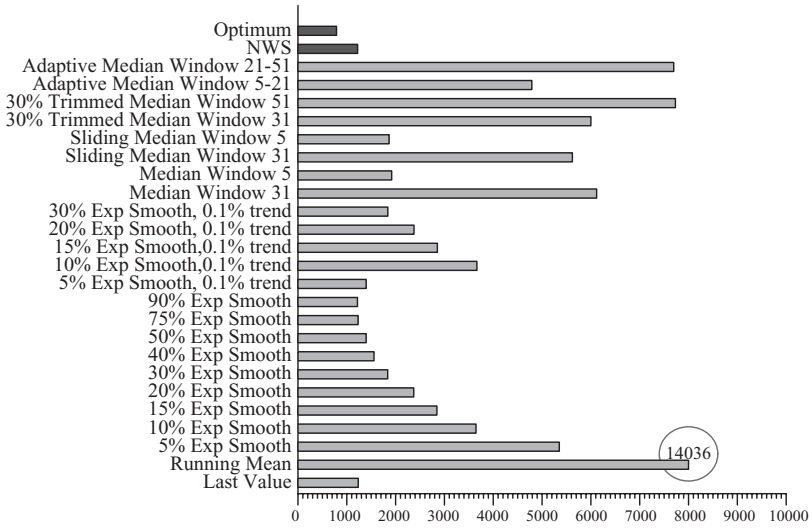


Fig. 3. NWS performance in Castle. Figure compares the square root of mean squared prediction error of each forecaster to NWS and the optimum case.

much better) the performance level of its best forecaster. While it is always possible to manually construct a data series for which NWS performs worse than the individual models, we do not observe such a case in any of the empirical data series that we evaluate.

Because the NWS was originally designed to support high-performance applications in wired settings, its designers put a premium on speed and extensibility. As such, it consumes significant resources to perform a single prediction, since many models are evaluated at once. The *average cost* column of Table I shows the number of floating-point instructions executed for each predictor (all are computed for each forecast made), on average. To enable its use in resource-restricted environments, we have significantly reduced this consumption

without sacrificing appreciable accuracy. To this end, we first evaluate the cost of NWS prediction in terms of dynamic floating-point instructions.

Given a history of measurements and their predicted values, we define prediction error using the square of the errors:

$$E = \sum_{i=1}^{n} (f_i - v_i)^2 \tag{5}$$

where f_i is the output of the predictor, v_i is the measurement, and n is the length of history.

Since the NWS uses a mixture-of-experts approach, all forecasters are invoked logically in parallel and a single winner is selected and used for the next estimation. We use zero-one integer variables $s_{i,j}$ to denote the winning forecaster:

$$s_{i,j} = \left\{ \begin{array}{ll} 1 & \text{If model } j \text{ is used to predict} \\ & \text{measurement } i \\ 0 & \text{Otherwise} \end{array} \right\}$$

Specifically, if $s_{i,j}$ is 1, the ith forecast is made using predictor j. If $s_{i,j}$ is 0, the predictor is not the winner for the ith forecast. If we set k to be the number of models in NWS, using Equation (5), we can formulate prediction error of NWS as:

$$E = \sum_{i=1}^{n} \sum_{j=1}^{k} (f_i - v_i)^2 s_{i,j} \tag{6}$$

Similarly, we can compute the cost of using the winning forecasters (in terms of floating-point instructions, c) as:

$$C = \sum_{i=1}^{n} \sum_{j=1}^{k} c_j s_{i,j} \tag{7}$$

Theoretically, it is possible to optimize NWS by running it with a different combination of internal models on a set of representative data and then removing the least efficient ones. However, the search space is prohibitive: There are a total of 2^{24} combinations. To reduce the search space, we use a heuristic that evaluates how much the total computation cost and error would change if we substitute a forecaster u with another forecaster v throughout the series.

Formally, we express this process as:

$$s'_{i,j} = \left\{ \begin{array}{ll} 1 & \text{If model } j \text{ is winner forecaster} \\ & \text{for measurement } i \text{ and } j \neq u \\ 1 & \text{if model } j \text{ is not winner forecaster} \\ & \text{for measurement } i \text{ and } j = v \\ 0 & \text{Otherwise} \end{array} \right\}$$

where we define $E_{u,v}$ and $C_{u,v}$ as Equations (6) and (7) using $s'_{i,j}$ instead of $s_{i,j}$

We employ real measurement data (i.e., performance traces) to empirically evaluate the overhead and accuracy of each NWS predictor from various embedded system resources: CPU load, wireless and wired network bandwidth and

	1	2	3	4	22	23	24
1	0	0	0	0	0	0	0
2	13.8	0	0.1	0.1	1.1	1.0	4.8
3	28.3	6.9	0	0	3.4	3.9	9.5
4	26.7	8.4	0	0	2.6	3.3	8.7
22	0.4	0.1	0	0	0	0	0.1
23	0.3	0.4	0	0	0	0	0
24	0	0	0	0	0	0	0

(a) Error matrix

	1	2	3	4	22	23	24
1	0	0	0	0	139.0	373.0	32.0
2	−2.0	0	0	0	305.0	819.0	71.0
3	−5.0	0	0	0	343.0	924.0	80.0
4	−6.0	0	0	0	12.0	33.0	3.0
22	−6.0	−6.0	−6.0	−6.0	0	10.0	−5.0
23	−18.0	−17.0	−17.0	−17.0	−11.0	0	−16.0
24	0	0	0	0	0	0	0

(b) Cost matrix

Fig. 4. Error and cost matrix for a real input. The matrices show the change in error and cost in percentages when forecaster v (in rows) is substituted with forecaster u (in columns). For example, in this particular dataset, substituting forecaster 2 (running mean) instead of 22 (30% trimmed median window 51) increases error only 0.1%, with a 6% decrease in cost.

latency, and task CPU demand. Our dataset includes more than 300 traces. We use six of these traces (randomly chosen among CPU load, network bandwidth, and network latency traces) to identify NWS forecasters that enable high accuracy at low cost. We use the rest of the dataset (i.e., not including these six) to evaluate NWSLite. We provide a complete discussion of our data traces in Section 4.1.

We compute $E_{u,v}$ and $C_{u,v}$ for every pair of u and v using these six traces and record the results in a matrix with u as the rows and v as the columns. This representation provides a very compact form with which we can evaluate the efficiency of each model: Every column of the matrix shows how much the error rate changes if we use v instead of u. For example, $E_{2,1}$ shows the new error if we use *last value* instead of *running mean*. If the $E_{2,1}$ is smaller than original NWS's error rate for *all* the trace files, then we consider *last value* to be a better predictor than *running mean*. Similarly, if in an extreme case, all the values of column 2 are smaller than original NWS's error rate, then the *running mean* outperforms the original NWS. Even though this is theoretically possible, we did not come across an example of such a case.

In Figure 4, we show the error and cost matrices for an example dataset. The numbers to the leftmost and topmost of the matrices show the enumeration of forecasters (given in Table I). The numbers in the matrices show the change in error and cost in percentages when we substitute a forecaster u (in rows) with a forecaster v (in columns). For example, when we substitute 30% trimmed median window 31 (number 22-shaded row) with running mean (number 2-shaded column), the error increases by 0.1% and the cost decreases by 6%.

The error and the cost matrices indicate how efficient individual NWS models are. We use these matrices to derive the rules with which we eliminate the models that are least efficient, i.e., the models that are never or rarely used (winners), and the models that are too expensive to justify the extra accuracy that they provide.

Since the last eight models of NWS (forecasters 17–24) are significantly more expensive than the remaining, we first concentrate our efforts on this group. These models use different parameterizations of sliding windows over previous measurements to provide a forecast based on the median of these measurements. Depending on the sliding window size and the model adaptiveness, these models may use up to a couple of hundred floating-point operations for each prediction, which is a magnitude of order more than the other NWS models.

Fortunately, our profiles indicate that seven of the eight forecasters in this group can be substituted by other forecasters with little loss in accuracy. As an example, substituting these forecasters with running mean and conservative exponential smoothing forecasters (i.e., gain factor $\leq 20\%$) increase the error rate by up to 0.7%. The fairly low increase in error rate indicates that these models are either rarely winners or their predictions are not significantly different from the predictions of these predictors that we substitute them with. The only exception to this rule is sliding median window 31.

Substituting sliding median window 31 with any other single forecaster, including the running mean, increases the error rate by at least 5.6%. While such an increase is large, we have experienced that a hybrid solution of multiple (and less expensive) predictors are more resilient and thus justifies removing this expensive predictor. In this particular case, for example, when we subsitute the aforementioned model with median window 5, the error increases by only 1.1% in one trace while it increases as much as 43.5% in another one. However, when we consider that sliding median window can be substituted by any of the median window 5, running mean and exponential window models; for each profiled trace, we find that there is at least one forecaster that, when substituted, can achieve an increase in error rate that is 5% or less. Thus, we remove this forecaster from our set.

We conclude that the forecasters 8 to 11 are rarely *used* in the profiled traces, because the increase in error rate is *zero* for any of the 23 other forecasters that we substitute them with. We find only one trace that these models are used extensively, and in this case, substituting them with less agile exponential smoothing predictors (i.e., the predictors 3–6) increases the error only marginally, by 0.5%. We also find that the forecasters 12–15 are rarely winners and thus substituting them with others do not significantly increase the error rate.

Fortunately, we find that the most heavily used predictors are the first eight, which are also the cheapest in terms of computational cost. Here, we find that last value and running mean are used extensively and cannot be substituted with others without a significant increase in error rate. While last value is an extremely responsive forecaster, running mean is highly conservative and more resilient to noise. Thus, these two forecasters complement each other well. However, there are many cases that these two are not enough. We find that at least in one-half of the profiled traces, keeping only running mean and last value increases error by 32 to 82%.

The reason for this high error rate is the lack of forecasters that fall in between running mean and last value in terms of their responsiveness. We observe that the exponential smoothing models (models 3–7), which are

exponential smoothing predictors with relatively small gain factors (5–30%), are also heavily used in our traces. Here, we find that 5% exp smoothing and 10% exp smoothing are generally interchangeable, with an error increase of 0.6%, at most. Similarly, 20% exp smoothing, 30% exp smoothing, and 40% exp smoothing are also interchangeable.

We formalize these observations into three rules with which we eliminate forecasters. We remove any model

—that we can replace with another model with 1% or less error rate across all traces. Most median-based models satisfy this rule and are replaced with other models.
—for which there is another model with significantly lower cost that can replace it with a small increase in error (<5%). We replace sliding median window 31 with median window 5 using this rule;
—for which there is a combination of other models that enable a similar error rate. Most exponential smoothing predictors satisfy this rule, as we described above, and, thus, are eliminated.

As a result of profiling, we identify the predictors that are most heavily used and that have the best performance. The predictors that we identify include both the agile (last value) and conservative predictors (running mean) and the ones that are in between (median window 5, 5 and 20% Exponential Smoothing). These five predictors (shown in bold in Table I) trade off cost and prediction error most effectively. However, these findings are valid only for the traces that we profiled. To understand the generality of our results, we evaluated them on a second, completely different, set of six traces. We briefly summarize our findings below.

In the second set, we also find the impact of median models to be marginal. Here, we find only two traces where these predictors are used extensively and, in both cases, the exponential smoothing and running mean predictors can substitute them with an error that is less than or equal to 0.5%. There is only one case where median window 5 is significant and cannot be substituted. Thus, we eliminate all predictors except median window 5 from this group.

In addition, we find three traces in which exponential smoothing models 7–11 are used extensively. However, in each case, we find that it is possible to substitute them with one of the previously identified predictors without changing the error significantly. For example substituting 40% exp smoothing with 5% exp smoothing increases the error rate by 2.2%. Overall, we find only slight changes to our initial results.

Figure 5 gives the pseudecode for NWSLite. The only significant change with respect to NWS is the removal of windowing system. In NWSLite, we only consider the entire history of the model in forecaster selection. While NWS allows us define multiple windows to the past, enabling this mechanism is extremely costly. Each defined window requires NWS to exercise all the predictors for this window and thus increments the cost to levels that are prohibitive on our target devices. Furthermore, there is no mechanism that we know that can automatically choose the best window size. Consequently, enabling one window

```
for each of the five forecaster implemented in NWSLite
        forecast using current forecaster
        record aggregate prediction error for current forecaster
end for
        choose forecaster with lowest aggregated error and make final forecast using it
```

Fig. 5. Pseudocode for NWSLite forecast selection.

size is generally not enough; (just like NWS) we have to enable multiple windows and iteratively exercise the forecasters in each. Finally, as we show in the next section, the NWSLite performance without the windowing mechanism approaches to that of NWS. However, incorporating the windowing mechanism in a cost-effective manner and evaluating its impact on prediction accuracy is the primary goal of our future research.

4. NWSLITE EFFICACY

To empirically evaluate the efficacy of NWSLite, we performed experiments using a wide range of datasets, applications, and metrics. In the following subsections, we describe the experimental methodology (datasets and applications), detail the metrics we use in Section 4.2, and present our results using these metrics in Section 4.3.

4.1 Experimental Methodology

To empirically compare the resource forecasting system that we present in this paper, NWSLite, to extant approaches to resource performance forecasters, we collected traces from a wide range of resource types: CPU demand (execution time) of application tasks, wired and wireless network bandwidth, wired network latency, and CPU availability. We then used the NWSLite and competitive approaches to make predictions using the trace data. In total, we performed experiments on 346 traces, which produced more than seven million predictions. All of the traces, with the exception of application execution times, were made freely available to us via websites of research groups around the country [NWS; Balachandran et al. 2002; GrADS]. We provide the details on the different datasets in Table II and we refer to each of the different types of data sets (application execution times, CPU availability, bandwidth, latency, etc.) as "groups."

We generated execution time traces, i.e., CPU demand, ourselves using the 3-D rendering applications used in similar studies [Narayanan and Satyanarayanan 2003; Narayanan 2002]. The applications and inputs that we considered are shown in Table III.

GLVU [2002] allows navigating inside a 3D scene by rendering the scene from any viewpoint of user. From an augmented reality view, Radiator [Willmott 1999] complements GLVU by computing the lighting effects for a given scene. Both applications can easily be divided into *operations* [Narayanan and Satyanarayanan 2003], which is a suitable unit for remote execution and

Table II. Datasets Used for Evaluation

Name	Trace size	Description
Application	20 Traces 17,870 Predictions	Interactive, 3D rendering application CPU demand. Measurements are CPU time from user request to program response.
Network bandwidth [NWS]	132 Traces 750,476 Predictions	Observations of 64 KB–1MB TCP data transfers. Three configurations: UIUC LAN (intercluster), UIUC campus-wide network (intracluster), and cross-country Internet (UIUC–UCSD)
CPU load [NWS]	59 Traces 6,000,697 Predictions	Fraction of CPU occupancy time a standard user process can obtain Observations are in 10-s intervals.
Network latency [NWS]	134 Traces 750,305 Predictions	Round trip time of TCP. Transferring 4 bytes and measuring acknowledge time. Granularity levels same as network bandwidth.
Wireless bandwidth [Sigcomm01traces]	1 Trace 3,028 Predictions	Four access points on same subnet. Traces include 195 users, 300,000 flows and 4.6 GB of network traffic. Bandwidth computed in 1-min periods

Table III. Applications and Inputs Used for Evaluation[a]

		Applications	
Input scene	Scene size (bytes)	GLVU	Radiosity
castle	385,391	Yes	
cessna	200,553	Yes	Yes
chevy	678,806	Yes	
cloister	7,816,848	Yes	
cup	97,113		Yes
dragon	3,382,396		Yes
ground-table-land	640,939	Yes	Yes
ground-riverain-valley	634,007		Yes
shuttle	15,658	Yes	Yes
venus	3,483,433		Yes

[a]We collected ten trace files per application (3D scene-rendering programs) using different inputs and navigation paths. Empty entries indicate that the application failed to process the particular scene; "Yes" entries are those inputs we employed for this study. We processed some inputs multiple times (to total ten) using different navigation paths.

fidelity adjustment. An operation (which we also refer to as a task) is the smallest user-visible execution unit, such as viewpoint change in a rendering operation. For each application, we rendered a set of ten scenes, which produced a total of 17,870 operations. We employed all of the inputs shown in Table III; we processed some inputs multiple times using different navigation paths. We consider the prediction performance for applications to be the accuracy with which the prediction system forecasts the CPU demand of each task.

The bandwidth, CPU availability, and latency data were collected as a part of the NWS project [NWS]. NWS network sensors use active network probes to collect TCP/IP latency and bandwidth data on a group of geographically distributed hosts connected via local, wide area, and Internet networks. Each probe establishes a TCP connection, transmits a fixed amount of data, and tears down the connection. Network sensors measure network bandwidth using a 64 KB data transfer and network latency using a 4-byte data transfer.

The NWS CPU sensors combine the information from Unix system utilities *vmstat* and *uptime* with periodic active CPU occupancy tests to provide measurements of CPU availability. The *uptime* utility reports the average number of processes in the run queue over the last 1, 5, and 15 min. The sensor uses the average load over the 1 min period and computes the CPU availability by using the idle, user, and system time output from *vmstat* utility. The CPU availability is measured as the fraction of CPU occupancy time a standard user process can obtain.

The wireless bandwidth traces we used were collected during the SIGCOMM'01 conference [Sigcomm01traces]. The conference building was covered with four 802.11b access points. The traces span a 3-day period capturing 300,000 flows generated by 195 users consuming a total of 4.6 GB of bandwidth.

Note that, NWSLite includes the five forecasters that we selected using profiling (marked in bold color in Table I). We use these five forecasters across all our evaluations.

4.2 Evaluation Metrics

We present our empirical evaluation of the different prediction systems in terms of both accuracy and computational cost. We use three metrics, described in this section, to evaluate predictor accuracy. We use instruction count (both total and floating point) as the metric for predictor cost.

The first of the three metrics we use to evaluate predictor accuracy is error deviation. We define error deviation as:

$$MSE = \frac{\sum_{i=1}^{n}(x_i - y_i)^2}{n}$$
$$\text{Error deviation} = \sqrt{MSE} \tag{8}$$

where x is the set of n predictions and y is the set of n corresponding observations. The mean square error (MSE) is the average square prediction error over the n pairs, (x, y). The error deviation is the square root of the mean square error. Error deviation describes the error in absolute terms and represents (in analogy) the *standard deviation* of the errors with respect to the *expectation* constituted by the forecast. Error deviation accounts for outliers and is more sensitive to incorrect predictions than is *absolute error* in which the absolute value of the error is used.

However, the error deviation is most meaningful when comparing the performance of predictors on the same time series. To provide a comparison across different series, we use a second metric that is the ratio of error deviation over the average observed value, i.e., the relative error rate:

$$\text{Relative error rate} = \frac{\sqrt{MSE}}{observed\_mean} \tag{9}$$

This metric provides insight into how severe the error is in terms of the magnitude of the average measured value. For example, an error of 2-MB/s is large in a 10-MB/s link, but may not be significant in a 100-MB/s link.

The third metric we use for reporting prediction error is similar to relative error rate, however, instead of using the mean as the expected value, we use the

Table IV. Error Deviation for a Set of Representative Traces[a]

Description	Units	Avg	NWSLite	NWS	LSQ	RPF
APP1—best		148.85	5.29	5.36	8.18	22.01
APP2—median	s	9.18	1.32	1.33	2.39	5.70
APP3—worst		169.75	135.12	138.06	145.39	186.43
BW1—within cluster		65.80	17.16	16.96	52.11	17.19
BW2—cross-cluster	Mbits/s	76.52	13.31	13.33	59.28	13.51
BW3—cross-country		4.54	0.88	0.86	78.06	1.16
CPU1—best		1.99	0.02	0.02	13.90	0.03
CPU2—median	CPU	0.54	0.02	0.02	14.45	0.05
CPU3—worst	fraction	1.39	2.67	2.68	3.11	2.66
LAT1—within cluster		13.94	16.87	16.89	41.12	17.05
LAT2—cross-cluster	ms	2.34	8.31	8.32	46.83	8.34
LAT3—cross-country		77.22	14.29	12.75	81.82	13.15
WBW	Kbits/s	206.67	193.78	194.50	255.25	261.74

[a]The third column is the average of the measured values, the next four columns show the error deviation for each of the prediction systems. The APP and CPU datasets are sorted with respect to *error deviation / average* and best, median, and worst cases are shown. For the BW and LAT datasets, the average error deviation within cluster, across cluster, and across country are reported.

absolute value of the forecast. This metric, called predictability, indicates how predictable the series is relative to the forecasts it generates. It differs from the *relative error* in that it treats each forecast as a *conditional expectation* that it uses to normalize the error, instead of using the overall measurement mean. We compute predictability as:

$$\frac{\sum_{i=1}^{n} \frac{|x_i - y_i|}{|x_i|}}{n} \tag{10}$$

4.3 Predictor Accuracy

We next present the results from our empirical comparison between NWSLite and competing prediction systems: The network weather service (NWS), Odyssey (LSQ and ODY-BW,LAT), and the remote processing framework (RPF). We implemented all of forecasters as efficiently as possible using the C language; we compiled each using gcc and -O2 optimization. Unlike NWSLite and the NWS, the LSQ and RPF methods are parametric models and, hence, require parameterization. For each model, we created a pool of parameter settings, that included the published values [Narayanan and Satyanarayanan 2003; Flinn et al. 2002; Noble et al. 1997] as well as our own values, resulting in 18 different forecasters. For conciseness, we selected the best performing parameterization for each over all of the datasets we considered. The full results are available in our technical report [Gurun et al. 2003].

Table IV compares the error deviation (Equation 8) of the predictors using three representative traces, for brevity. In the application (APP) and CPU availability (CPU) datasets, we sorted the traces with respect to the *error deviation / average* of NWSLite and selected the best, worst, and median, which we report in the table. For the wired network data (bandwidth (BW) and latency (LAT)), we instead report data for three different types of links: intracluster, intercluster, and intercampus (across country). For wireless (WBW), we only have a single trace and, thus, show data only for it.

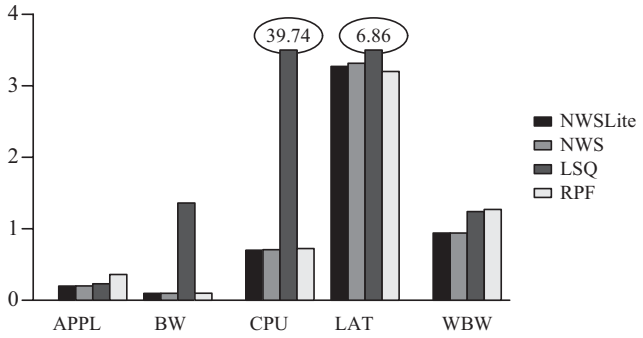


Fig. 6. Relative error rate (Equation 9). This metric shows how severe the error is with respect to the average measured value. The LAT has the highest relative error rate among all forecasters. However, as most latency observations are very small (around 1 msecs), the absolute error is small.

The first three columns of the table shows the description, trace name, and value units for each trace. The third column, Avg, shows the average observed value. The final four columns show the error deviation for each of the four predictors: NWSLite, NWS, LSQ, and RPF. LSQ and RPF are parameterized as described in Section 2 and identify the best-performing, converging parameterizations of each technique.

The NWS and NWSLite have almost identical error deviations in every case. LSQ performs well for applications (as was shown in prior work [Narayanan and Satyanarayanan 2003]), but it is the worst-performing predictor for all other types of data. NWSLite performs better than LSQ and RPF in almost every case and is significantly better than both LSQ and RPF in most cases. For example, in the application group, for both *shuttle* and *cloister* NWSLite performs three times better than RPF. The wireless dataset is especially challenging. All the forecasters show a high error rate.

Figure 6 shows the relative error rate of the predictors across all of the traces in each group. The information in the graph confirms the results of Table IV. NWSLite performance is very similar to that of the NWS; in all groups it enables the best prediction error. LSQ is ineffective for the bandwidth, CPU, and latency groups. RPF performs quite well for the CPU and bandwidth groups and exceeds NWSLite performance for network latency by 1.5%. RPF is the worst predictor however, for the application and wireless groups. For the application group, the average error rate of RPF is 86% higher than that of NWSLite.

We also compared the performance of predictors with Odyssey's specialized smoothing filters for bandwidth and latency, which we refer to as ODY–BW and ODY–LAT (omitted for clarity). ODY–BW performed 25% worse than NWSLite and ODY–LAT performed 19% worse than NWSLite.

Figure 7 shows the predictability (Equation 10) of the series given each predictor. This metric assumes that predictor is a valid conditional expectation that can be used to normalize the error at each point of the trace. The lower the value the more accurate the forecaster. Since the variance of the results is high, we normalized the results to NWSLite for each group.

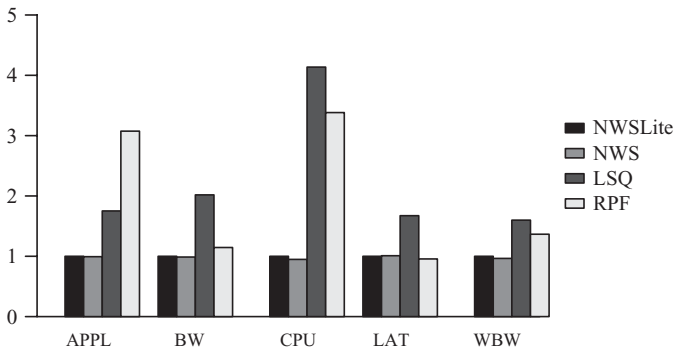


Fig. 7. Predictor predictability (Equation 10). Because of high variation among forecasters, the values are normalized to NWSLite for each group. The lower the value, the more accurate the forecaster.

The predictability results support our findings in Figure 6. NWS-Lite is as accurate as NWS in all cases and it performed significantly better than the parameterized forecasters, in most cases. The single exception is the latency dataset, in which RPF is the winner. However, the difference between RPF and NWSLite is very small. In contrast, the accuracy of RPF is significantly worse than NWSLite for the application, CPU, and wireless bandwidth data, emphasizing the difficulty of finding a good parameterization for the general case. These results also show that, with the exception of the application dataset, LSQ always performs worse than the predictors based on smoothingfilters. In the application dataset, LSQ is approximately 40% more accurate than RPF. However, it is still significantly worse than NWSLite. The predictability of NWSLite is considerably higher than even the highly tuned predictors ODY–LAT and ODY–BW (not shown in figure). For the latency dataset, ODY–LAT is 13% less predictable than NWSLite; whereas in bandwidth dataset, NWSLite does 21% better than ODY–BW.

An interesting case is the behavior of RPF in Figures 6 and 7; even though the relative error rate of RPF is small, its predictability is not. This is because of the characteristics of CPU dataset. The CPU availability values are in the range $(0, 1)$, or $(0, n)$ if there are n processors. As such, most of the time the values are a fraction of 1. This results in a small value for the sum of square errors, even though the errors are high relative to the expected value.

4.4 Computational Cost of Prediction

In addition to studying prediction error, we also considered the cost of performing prediction on a resource-restricted device. To our knowledge, no prior studies that use prediction on mobile devices consider the resource consumption of the predictors themselves.

We first compare the predictors in terms of instructions required for one prediction. We extracted this information by using the SimpleScalar [Burger and Austin 1997] simulator. Figure 8 shows the average cost of each predictor. NWSLite uses 55 floating-point instructions per forecast. Even though this is

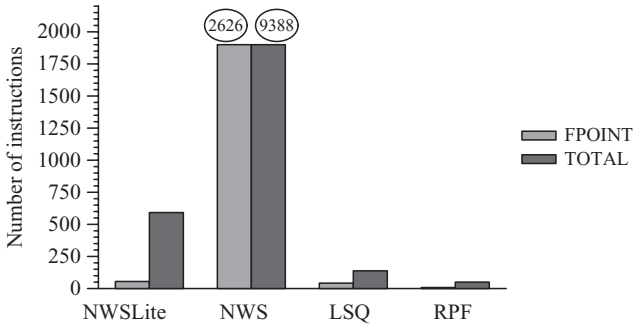


Fig. 8. Forecaster cost as number of instructions executed (floating-point (FPOINT) and TOTAL) per prediction.

Table V. Execution Cost Comparison per Prediction

Prediction system	Floating point	Total instructions	Execution time (μs)
NWSLite	55	592	381.34
NWS	2626	9388	10231.31
LSQ	42	138	295.27
RPF	8	50	154.9

more than the cost of RPF and LSQ, which use 8 and 42, respectively, the accuracy of NWSLite significantly exceeds both of these predictors.

As most resource-restricted devices lack a floating-point coprocessor, floating-point instructions are very expensive. We break down the instruction counts into floating-point and nonfloating-point instructions in the first two columns of Table V.

We also executed the predictors on a real resource-restricted device: An iPAQ H3800 hand-held computer from Compaq [Compaq Computer Corporation]. The iPAQ has a 206-MHz Intel StrongArm CPU and runs Familiar Linux, version 0.5.3. The execution times (in μs) are shown in the final column of the table. These times include the cost of IO to read the trace file from flash memory and to print the results.

The execution time of NWSLite is approximately 4% that of NWS, but enables prediction accuracy that is nearly equivalent. Given that it requires only 381 μs to execute a prediction, including the IO, NWSLite is a more attractive solution for on-line forecasting using resource-restricted devices, than the parametric and less accurate models of Odyssey and the RPF.

4.5 Evaluation Summary

We summarize the result of our findings in Table VI. To make our results comparable to previous studies [Narayanan and Satyanarayanan 2003], we report summary performance in terms of percentile error. We define the X percentile error, E_X, as the maximum *absolute* error for X% of the experiments. For example, for the bandwidth dataset, E_{95} of NWSLite is 25.6, meaning that 95% of the time the prediction error of NWSLite is within 25.6 kilobits/s. The

Table VI. Results in Summary: Percentile Error[a]

	APP		BW		CPU		LAT		WBW	
	E90	E95	E90	E95	E90	E95	E90	E95	E90	E95
NWSLite	3.32	7.34	10.27	25.70	0.02	0.04	15.77	24.57	198.13	351.09
NWS	3.34	7.46	9.60	25.58	0.02	0.04	15.80	24.50	202.77	358.80
LSQ	5.87	13.31	14.10	28.46	0.06	0.12	16.42	26.87	230.59	422.98
RPF	17.15	38.70	10.60	25.56	0.08	0.21	16.19	24.92	326.34	533.05
ODY–LAT	3.76	8.81	9.92	39.72	0.02	0.09	16.32	29.85	197.43	335.17
ODY–BW	3.46	7.89	7.38	42.54	0.02	0.08	16.88	31.49	192.99	354.56

[a]We define the X percentile error, E_X, as the maximum absolute error for X% of the experiments. The table compares the E_{90} and E_{95} of all forecasters for all five datasets and prediction systems studied.

reason we use absolute rather than relative error is to avoid skewed data in CPU and latency datasets. We report the results for NWS, NWSLite, LSQ, and RPF, as well as for the two other smoothing filters that we studied, ODY–LAT (the Odyssey network latency predictor) and ODY–BW (the Odyssey network bandwidth predictor).

The results show that NWS and NWSLite are general enough that they perform well in all datasets. Even though parameterized forecasters can match NWSLite in some datasets, they fail in others. As an example, the performance of ODY–BW is close to NWSLite in APP dataset, but it is significantly higher in BW, CPU, and LAT datasets. The same pattern also exists for ODY–LAT and RPF. RPF matches NWSLite in BW and LAT, but it is significantly worse in APP and CPU datasets.

Another pattern in the results is that both NWS and NWSLite perform better than all others when a higher percentage of predictions considered. This suggests that NWS and NWSLite can better adjust themselves to sudden changes in performance patterns by switching to another model; the other models must simply rely on their static parameters.

The wireless bandwidth dataset is significantly different than other datasets. The error rates are very high, i.e., E_{90} is around 200-Kbits/s on a 11-Mbits/s link; hence, none of the forecasters performed at a satisfactory level. This emphasizes the need for additional study of and novel forecasters for wireless network bandwidth data.

The success of NWSLite results from its capability to dynamically switch between a carefully chosen set of competing models, based on previously observed accuracy. If the dynamics of the observed dataset changes over time, NWSLite can adapt to the new conditions; the prediction systems of Odyssey and RPF cannot, and as such, are data (input) dependent. For example, exponential smoothing with a gain of 0.05 can be the most accurate predictor at some point, however, a transient or permanent change can occur so that the running mean can become the most accurate. In this case, NWSLite will respond by switching to running mean if the change is persistent enough to cause the aggregate error ranking to change. Odyssey and RPF are statically configured by a set of predetermined parameters. Thus, even though there are individual cases that other predictors can match the accuracy of NWSLite, they are unable to do well across dynamically changing series and to different types of resource performance data.

The flip-flop filter extension to Odyssey [Kim and Noble 2001], described in Section 2, incorporates some adaptivity by using two different parameter settings in its exponential smoothing predictor. However, exponential smoothing cannot always produce the best prediction accuracy (given any gain parameters). NWSLite incorporates exponential smoothing using two different gain factors but is more general and adaptive than this filter, since it considers a wide range of other prediction techniques that can enable significant improvements in accuracy at low computational cost.

5. ENERGY/PERFORMANCE BENEFITS OF PREDICTION ACCURACY

The previous section discusses NWSLite efficacy and compares it to other popular prediction methods. Our findings, which we gather using statistically sound metrics, show that NWSLite outperforms its competitors significantly. However, a key question that is remains is how much this improvement in accuracy translates into actual energy savings. Here, we answer this question by evaluating NWSLite within a remote execution system.

Offloading computation to remote, wall-powered, resource-rich servers can provide significant power savings. For offloading to be beneficial, it should be employed only when the benefits of remote computation exceeds the cost of offloading itself. This requires accurate modeling and prediction of local and remote resource supply, as well as the resource demand of the task. Our goal is to identify the degree to which higher accuracy impacts the decision whether to offload.

We next discuss the components required for a remote execution and detail the significant parameters. In Section 5.2, we describe our computation offloading setting. Finally, we show the results from our experiments in Section 5.3.

5.1 Remote Execution

Remote execution (aka computation offloading) is a popular technique that extends the computational capability of mobile, resource-restricted devices [Flinn et al. 2001; Rudenko et al. 1998, 1999; Kremer et al. 2001; Li et al. 2001]. Furthermore, depending on network communication cost, it can reduce the demand on local hardware resources and conserve significant amounts of energy. Figure 9 depicts the general design of a remote execution system. Using remote execution, application tasks are off-loaded from battery-powered mobile devices to wall-powered, higher-performance servers.

To decide whether a particular task should be offloaded, a remote execution system must first compute the *resource demand* of the application task. Demand can be defined using different metrics, and such as CPU cycles, network bandwidth, and memory pages, according to the overall goals of the system.

To determine how best to accommodate demand, a remote execution system must evaluate how best to employ its *supply*—the set of resources, local and remote, that it has available to it for task execution. The system computes whether computation off-loading will be beneficial, according to its set of constraints, using a cost model. When cost of local execution (i.e., L_l) exceeds that of remote execution (i.e., L_r), the system off-loads work to the server. The cost model must

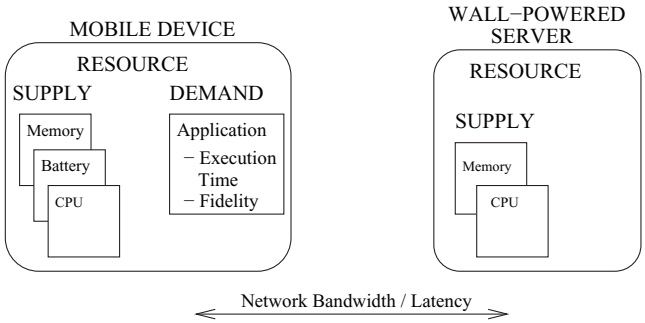


Fig. 9. Components of a typical remote execution system. The decision process includes forecasting the available resource supply both at the client and server and application resource demand.

consider both the task execution characteristics as well as the highly variable performance of the underlying resources that dictate computation and communication performance. However, constructing an exact cost function is nontrivial since hardware components have many shared resources, such as buses and DMA devices, that implement specific arbitration and priority policies.

Since the scope of this work is to compare NWSLite to other predictors, we employ a general cost model that assumes no I/O overlapping. We compute the available CPU cycles using the Odyssey model, which we give in Equation (1). We compute the local and remote execution cost as:

$$L_l = \frac{D_{cpu}}{S_{lcpu}} \tag{11}$$

$$L_r = \frac{D_{tx}}{S_{tx}} + \frac{D_{cpu}}{S_{rcpu}} + \frac{D_{rx}}{S_{rx}} + D_{rtt}S_{rtt} \tag{12}$$

where L_l and L_r stand for local and remote execution latency, D_{cpu} is the number of CPU cycles that the application requires to complete the task, and S_{lcpu} is the available CPU cycles on local machine, averaged in a period of 1 s. The remote cost is the sum of four constituent operations:

1. The time required for network transfer given the size of the demand for network send and any needed program code (D_{tx}) and given the available bandwidth (S_{tx}) between the device and server;
2. The execution time at the server given the average number of CPU cycles available at the server (S_{rcpu});
3. The time for transfer of results, e.g., data, status and rendered graphics, back to the device given the available bandwidth between the server and device ($\frac{D_{rx}}{S_{rx}}$); and;
4. The time required for handshake to establish connection, given the number of packet exchanges between local and mobile device (D_{rtt}) and network latency (S_{rtt}).

Since (4) commonly consists of very short packet communication between the device and the server, the handshake operation is impacted by the latency in the

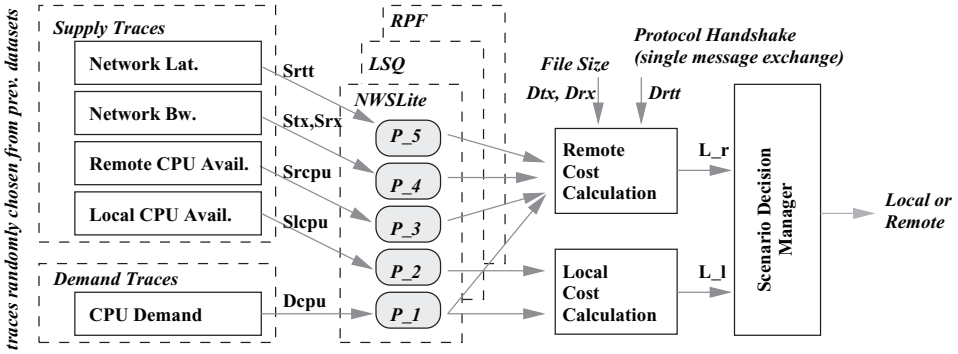


Fig. 10. Simulator components. The simulator reads traces that are chosen from Table II and predicts their future state using separate predictor instances (i.e., P_1 to P_5). The decision manager uses the forecasted values in choosing execution location.

network link between the client and server ($D_{rtt}S_{rtt}$). L_l and L_r can be enhanced to compute power, to integrate computation fidelity or battery lifetime into cost functions.

5.2 Methodology

To better understand how much the improved accuracy enabled by NWSLite matters to a real remote execution system, we constructed two scenarios and simulated those scenarios using the principles described previously. In our simulation, we limited the computation offloading scenarios to one mobile device and one remote execution server. We assumed that the local device is an HP iPAQ H3800 and the remote device is an IBM T23 laptop. The former machine has a 206-MHz StrongArm CPU, while the latter one uses a 1132-MHz Pentium III. We assume that both machines are only executing our applications.

The scenarios simulate computation offloading systems that have different goals. In Scenario1, the goal is to provide optimal userinteractivity. Many mobile applications, such as augmented reality applications, and games are user-interactive by their design, making response time a critical design parameter. For such applications, computation offloading is a viable option not only to improve response time but also to improve functionality [Narayanan and Satyanarayanan 2003; Kremer et al. 2001]. In Scenario2, the goal is to reduce power consumption as much as possible and to extend battery life. Scenario2 does not consider execution performance (latency) in offloading decisions.

Figure 10 shows the simulator components in detail. The simulator reads the measured values of each of the computation offloading parameters,—CPU demand, local and remote CPU supply, and network latency and bandwidth—from a file and predicts their future values by running separate instances of predictors for each type of data. Once the system computes the future values, it calls the decision manager, which determines whether local (a return value of 0) or remote computation (a return value of 1) will be used. The simulator also computes what the “right” or “best” decision is, once it reads the actual values, and computes various statistics for our use in the evaluation. The simulator

```
int likely_offload()
{ // returns 1 if remote execution chosen, 0 otherwise
        L_L = predict_local_latency();
        if (L_L < 50 milliseconds) {
                return 0;
        }
        L_R = predict_remote_latency();
        if (L_L > L_R) {
                return 1;
        } else {
                return 0;
        }
}
```

Fig. 11. Pseudocode for scenario1 decision manager.

Q7 reads and simultaneously predicts the supply and demand. We evaluate each CPU demand trace using 32 different TCP bandwidth, network latency, and CPU availability traces that are randomly chosen from our dataset that we described in Table II. We report the average results.

Each offloading decision requires prediction of application CPU demand and the state of four resources; network latency, network bandwidth, and local and remote CPU availability. Network latency is used to compute the cost of protocol handshake. Network bandwidth is needed to estimate cost of data transfer. CPU availability is used to compute the cost of local and remote computation. We use separate predictor instances on data histories to estimate next values of each of these resources. We use GLVU and radiosity demand traces (from Table II), for predicting CPU demand. We describe these traces in more detail, later in this section.

Figure 11 shows the pseudocode for the Scenario1 decision engine. The engine, which exists on the mobile device, offloads the task to the remote machine if the forecasted local execution time is more than 50 ms and the forecasted remote execution time is smaller than that of the forecasted local execution time. The tasks that are estimated to have an execution time less than 50 ms are never offloaded, since the human perception system can not recognize delays that are less than 50 ms [Card et al. 1983]. Thus, Scenario1 favors local execution, when appropriate, to reduce the stress on shared resources, such as the network and remote server. We discuss the implications of this choice on predictor efficacy in the next section.

In Scenario2, the decision process estimates power consumption for both local and remote execution and chooses the location that leads to lower power consumption. Unlike Scenario1, this scenario does not favor either local or remote execution (i.e., it does not take execution latency into consideration). Furthermore, its power computation function assumes that the local CPU and wireless interface in the idle state during remote execution. We detail the computation of power consumption later in this section.

We simulated Scenario1 using GLVU and Scenario2 using radiosity. As we explained in Section 4.1, both of these applications can be split into tasks that can be offloaded to a remote server or executed locally. To measure the task CPU demand (i.e., D_{cpu}), we captured the task execution times, in microsecond

resolution, as a user was navigating 3D scenes on a dedicated IBM T23 Linux laptop. We then computed the demand, in CPU cycles, as $D_{cpu} = t \times f$, where f is the CPU clock speed of the machine and t is the task execution time. Even though D_{cpu} is not completely accurate and portable across architectures because of differences in cache sizes and other CPU parameters (i.e., lack of floating-point coprocessor in StrongArm), we ignore such discrepancies as our focus is the accuracy of predictors, not the efficiency of computation offloading.

In our simulations, we assume that only the input data is transmitted across the network. The data is transferred to the remote device, processed, and transferred back to the local device. The executable never moves. This is similar to prior approaches in Kremer et al. [2001], Rudenko et al. [1999], and Flinn [2001]. Both GLVU and radiosity tasks operate on an object file that contains the current scene. Since the size of this file is known beforehand, there is no need to separately predict network demand.

Prior to the data transfer, the client and the server has to initiate a session. In our model, the initial handshake, which includes a single message exchange, and the data transfer are done reliably, using the TCP protocol. Other implementations tend to be more complex [Flinn 2001] and use protocols like remote procedure call, however, we do not discuss these for conciseness.

We use Equations (11) and (12) to compute local and remote execution latencies in Scenario1. In Scenario2, to compute power consumption, we extend these equations such that:

$$C_l = \frac{D_{cpu}}{S_{lcpu}} p_{busy}$$

$$C_r = \frac{D_{tx}}{S_{tx}} p_{tx} + \frac{D_{cpu}}{S_{rcpu}} p_{idle} + \frac{D_{rx}}{S_{rx}} p_{rx} + D_{rtt} S_{rtt} p_{tx}$$

In the first equation above, C_l stands for local execution energy consumption. We compute it by multiplying local execution latency with p_{busy}, which is the average power consumption of a highly loaded handheld computer. Similar to L_r; C_r, the energy consumption during remote execution, is a sum of four factors:

1. The energy required for network transfer, which is network transfer time multiplied by p_{tx}, the average power consumption during wireless transmission;
2. The energy consumption while waiting for execution at the server, which is remote processing time multiplied by p_{idle}, the average power consumption in sleep state;
3. The energy required for network receive, which is network receive time multiplied by p_{rx}, the average power consumption during wireless receive; and;
4. The energy required for handshake to establish connection, which, given the number of packet exchanges between local and mobile device, is equal to network latency multiplied by (p_{tx}).

Table VII gives the actual values of p, as measured by Li et al. [2001] on real handheld devices.

Table VII. Power Consumption of iPAQ Under Different Scenarios

Parameter	Power (mW)	Description
P_{idle}	550	CPU idle; wireless interface off
P_{busy}	1150	CPU highly busy; wireless interface off
P_{tx}	2200	Data send over wireless
P_{rx}	2100	Data receive over wireless

Table VIII. Overview of 3D Objects

		Object features		Number of	% of Offloading decisions		
		Size (KB)	Complexity	decisions	RPF	LSQ	NWSLite
Scenario1	Castle	385	Medium	78,528	32.37	32.23	32.39
	Shuttle	15	Low	14,080	64.32	66.82	65.00
	Ground-table	640	High	26,496	48.12	49.32	48.65
Scenario2	Cessna	200	Medium	12,736	28.33	31.83	29.72
	Venus	3483	Very High	2,720	9.63	12.21	16.58
	Ground-table	640	High	5,152	27.93	34.71	28.13

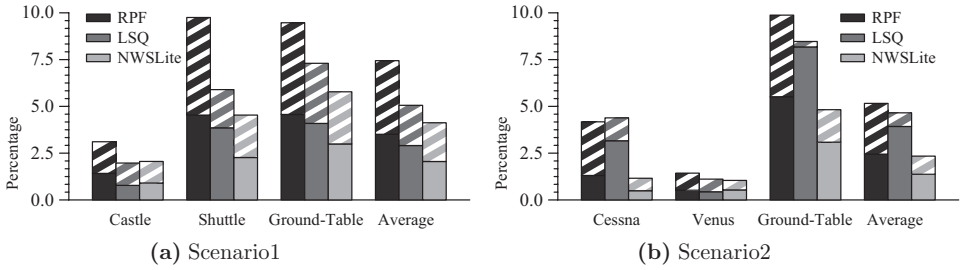
We simulated each scenario using three input scenes. We chose the scenes arbitrarily, from Table III, however, we were careful to choose one small, one medium and one large scene. For GLVU, we used castle, shuttle, and ground-table-land. For radiosity, we used cessna, venus, and ground-table-land. In each scenario, we compared the efficacy of NWSLite with RPF and LSQ using the best performing parameterization, as we described in Section 4.3. We did not include NWS in our evaluations because of its high cost.

5.3 Simulation Results

In this subsection, we evaluate how prediction effects the performance of the decision engine. There are two ways that the decision engine can fail for a given task: (1) the decision engine chooses local execution even though remote execution is more beneficial; (2) the decision engine chooses remote execution even though local execution is more beneficial. We refer to the former as *wrong locals* and the latter as *wrong remotes*. We use *wrong decisions* to refer to the sum of both wrong locals and wrong remotes.

Table VIII gives a brief overview of all the 3D objects that we used. The first part of the table describes object features, including size, in Kilobytes, and complexity, in scales that change from "low" to "very high." A higher complexity object has more vertexes and edges per unit area and more details, such as 3D information, and color. Such increase in complexity requires more network demand, but not necessarily more CPU demand, because rendering algorithms can intelligently prune out many details, such as the vertexes that are not visible during processing. For example, even though ground-table is almost two times larger than castle, its average rendering cost is approximately the same as that of the castle.

The rest of the table gives the total number of task offload decisions and the percentage of offload decisions given by each predictor. A high number of task offload decisions shows that the user navigated the object for a longer duration, generating a larger number of tasks. This is typically the case for the GLVU



(a) Scenario1 (b) Scenario2

Fig. 12. Percentage of wrong decisions. The striped and solid parts show wrong remote and local execution decisions, consecutively. NWSLite beats other predictors in each benchmark.

tasks in Scenario1, since the tasks are shorter than the computationally demanding Radiosity tasks. The ratio of tasks that the predictors chose to offload varies from 10% (venus in Scenario2) to 64% (shuttle in Scenario1), depending on task characteristics. However, these numbers show only the degree to which predictors utilized local and remote execution and does not indicate whether these decisions are correct.

In Figure 12, we compare each predictor in terms of wrong decisions. Each bar shows the percentage of wrong decisions. The striped and solid sections represent the wrong remotes and wrong locals consecutively. For example, for *castle*, approximately 2.6% of all decisions were wrong and the ratio of wrong locals and wrong remotes were approximately equal. The last three bars show the average. We compute average by equally weighing all benchmarks; for example, if a scene has 450 wrong offloading decisions among its 900 tasks, and another scene has 10 wrong offloading among 100 tasks, we compute the average of wrong offloading decisions as 30%, not 46%.

In a remote execution system, the computation offloading decisions show a boolean characteristic. The possibility of a wrong decision increases when the gap between the cost of local and remote execution is small. That is because the small gap cannot compensate any prediction errors. An example is venus in Scenario2. Venus is an extremely sophisticated scene. Because of its size, remote execution is very costly, however, local execution is not (i.e., even though venus is approximately 5 times larger than ground-table-land, the CPU demand is only 2.2 times larger, on average). The large margin between the cost of local and remote execution compensates most prediction errors; therefore, all the predictors can achieve very few wrong decision rates ($<$1%).

Overall, the wrong decision rate was less than 10% for all benchmarks. NWSLite was always better than the other predictors. In Scenario1, the wrong decision rate for NWSLite, LSQ, and RPF were 4.1, 5.1, and 7.5%, consecutively. In Scenario2, NWSLite performed even better. The rate was 2.3% for NWSLite and 4.7 and 5.2% for LSQ and RPF. This corresponds to 67% fewer wrong decisions than RPF and 14% fewer wrong decisions than LSQ for the first scenario. In the second scenario, the difference between NWSLite and other predictors is even larger; NWSLite gave 95 and 73% fewer wrong decisions than RPF and LSQ, consecutively.

Table IX. Impact of Wrong Decisions[a]

		No Prediction		Predictor Guided			
		All local	All remote	Oracle	RPF	LSQ	NWSLite
Scenario1 (s)	Castle	10548	7972	6946	7279	7266	7222
	Shuttle	711	339	339	351	354	341
	Ground-table	3515	3499	3099	3258	3249	3250
Scenario2 (J)	Cessna	7810	1747	1530	1629	1579	1569
	Venus	7912	4944	4320	5941	5571	5330
	Ground-table	6815	2032	1862	2014	1959	1928

[a]The table shows the total (i.e., across all experiments) cost of execution for each policy. The first two columns show the cost when we execute fully local and fully remote. The next column (Oracle) shows the cost when we have access to true values rather than predicting them. The next three columns show the cost for RPF, LSQ, and NWSLite.

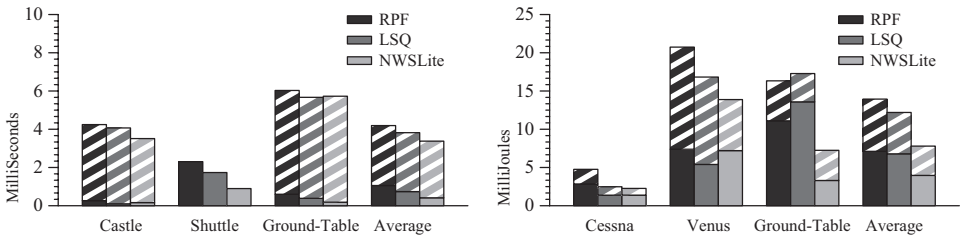
The wrong decisions were almost equally distributed among local and remote execution except the *ground-table-land* benchmark in Scenario2. In ground-table-land, only for the LSQ predictor, wrong locals were dominant. A plot of cost function unveils an interesting phenomenon: even though remote execution cost is stable, there are frequent, steep changes (i.e., dips) in local execution cost. When such a dip occurs, LSQ tends to over correct its parameters, resulting a steeper reduction (i.e., an underestimation) in local cost estimation, resulting in many wrong local execution decisions.

Even though a decision can be either *right* or *wrong*, not each wrong decision is equal in terms of its impact on the performance of the system. When the cost of local and remote executions are close, the impact of a wrong decision is relatively small. On the other hand, when the costs diverge, i.e., for example, when local execution is much cheaper than remote execution, an incorrect decision can be disasterous. In the latter case, however, it can be argued that most predictors can get the correct decision easily as the large delta between local and remote execution cost forgives most of the inaccuracy in prediction. Thus, in Table IX we compare the forecasters in terms of their impact on the total performance of the system.

The first two columns of the table show the cost of executing all the tasks locally, and remotely. Here, in scenario1, while computing the cost of *all remote* execution, we comply with the decision policy, and assume the tasks that are shorter than 50 ms are locally executed. In the next column, we show the cost when we use an Oracle that knows the true cost (an unrealistic, baseline scenario). The remaining three columns compare the execution cost when we use RPF, LSQ, and NWSLite for prediction. In the table, the costs are shown in seconds for Scenario1 and in joules for Scenario2.

In general, we find that an all remote execution policy is cheaper than an all local execution policy. Scenario2 benefits more from remote execution because of the increased computational complexity of radiosity tasks. In addition, both in Scenario1 and Scenario2, the predictors improve the results significantly (up to 15%) when compared to an all remote policy.

When we compare the predictors among themselves, NWSLite continues to beat the others. In Scenario1, all predictors are exceptionally successful and close to Oracle, leaving little margin to improve on. However, the impact of



(a) Scenario1 (b) Scenario2 **Q8**

Fig. 13. Cost of wrong decisions. The striped and solid parts show wrong remote and local execution decisions, consecutively. Scenario1 is very asymmetric; for castle and ground-table-land, almost all cost is because of wrong remote execution decisions and for shuttle all cost is a result of wrong local executions. This is expected because of asymmetric offloading rules. NWSLite beats other predictors in both scenarios.

NWSLite is still obvious. In castle, for example, the total execution time is 7279 s for RPF and 7222 s for NWSLite. In Scenario2, where most tasks are much larger (and each wrong decision is more costly), the performance difference between predictors is much more evident. Here, using NWSLite can save 30 to 241 J more than its closest competitor. Overall, we find that NWSLite either significantly outperforms the other predictors, or at least matches their performance.

In Figure 13, we compare the cost of wrong decisions. We compute the cost as $\sum c_i/n$, where c_i is the cost of a wrong decision i and n is the number of all tasks. We compute c_i as the amount of extra response time -or extra energy consumption, depending on the scenario between the correct decision and the wrong decision. In other words, this metric gives the expected wrong decision cost per task. The results are in ms for Scenario1 and in millijoules (mJ) for Scenario2.

Figure 13 shows that in Scenario1 there is a very uneven cost distribution among wrong locals and wrong remotes. In castle and ground-table-land, most cost is because of wrong remotes, in shuttle, all cost is results from wrong locals. This is because of the asymmetric nature of Scenario1; the computation offloading decision is given only when the task is expected to last more than 50 ms; therefore, only large tasks are offloaded and a wrong decision adds a huge error. We can see this effect clearly in castle and ground-table-land, but not in shuttle. Because of the relatively lower CPU demand of shuttle, (i.e., a very small scene of 15 KB), the predictors always estimated that local execution was adequate and never chose remote execution.

Table X shows the expected penalty for a wrong decision, in other words, it shows how costly a wrong decision is. We compute it by dividing the total cost of wrong decisions to the number of wrong decisions n_w, that is, $\sum c_i/n_w$. The results are in milliseconds for Scenario1 and in millijoules for Scenario2.

The expected penalty is not significantly different across predictors. In Scenario1, RPF had slightly lower penalty per miss, however, its effect was offset by the high number of wrong decisions. (i.e., Figure 12). In general,

Table X. Expected Penalty for a Wrong Decision[a]

		Wrong local execution decisions			Wrong remote execution decisions		
		RPF	LSQ	NWSLite	RPF	LSQ	NWSLite
Scenario1 (msecs)	Castle	18.76	12.90	18.15	235.28	332.29	290.32
	Shuttle	50.84	45.03	39.58	0	0	0
	Ground-table	13.15	9.47	6.18	110.76	164.45	198.96
Scenario2 (mjoules)	Cessna	217.47	43.66	280.86	66.77	89.85	131.85
	Venus	139.04	124.35	52.56	1464.00	1695.26	1315.97
	Ground-table	55.72	24.23	30.66	119.87	1278.40	227.11

[a]The cost of a wrong decision is proportional to the complexity and size of a scene. For example, for venus, the cost is extremely high, however for shuttle it is very low. The expected cost is almost same for NWSLite and other predictors.

predictors have fairly close results, however, ground-table-land in Scenario2, is marginal. As we explained before, this is because of the LSQ, which consistently underestimates the cost of local execution.

Table X also emphasizes the asymmetry in cost structure: In Scenario1, a wrong remote execution decision was much more expensive than a wrong local execution decision (i.e., 200- versus 20-ms latency). Therefore, in settings where power consumption is not a concern, it may be beneficial to continue local execution in parallel. The same cost structure also exists in Scenario2. Here, the wrong remote execution decision penalty was 410 mJ, in contrary to the wrong local execution decision penalty, which was only 107 mJ.

6. DISCUSSION

In this paper, we extend our prior work [Gurun et al. 2004] on general-purpose, non-parametric prediction for resource-restricted computers. In particular, we provide a thorough description of our heuristics-based approach with a justification of the parameters that we chose, and an investigation of the benefit of extra accuracy in a computation offloading setting. In this section, we provide a discussion of various trade-offs that impact the design of a prediction algorithm, and possible limitations of our approach and how to address these limitations.

With NWSLite, we provide a fully automatic and dynamic tool that attempts to balance computational complexity and prediction accuracy. While it is often the case that prediction accuracy can be increased by using sophisticated and computationally expensive techniques, finding the optimum technique requires finding a balance between the available computational resources, the benefits of extra accuracy, and the nature of the workload itself.

NWSLite is adaptive and dynamic in the sense that it can choose the most appropriate predictor by evaluating the past prediction errors. However, for doing so, NWSLite has to run all its forecasters in parallel. There is no way we can dynamically enable/disable a group of forecaster models in NWSLite, since NWSLite (and NWS) forecaster selection algorithm chooses the model, depending on the past history (if forecasters are disabled, they do not have a complete past history) .

However, under certain conditions, the additional complexity of a more sophisticated prediction algorithm may be more desirable. For example, in an offloading setting, if the tasks are extremely large and computationally expensive, the additional cost of a computationally expensive forecaster can be amortized effectively. Furthermore, not all embedded systems are equal in terms of their computational resources. It is more desirable to have a mechanism that can scale up/down its computational complexity, depending on the available resources of the system.

While NWSLite cannot dynamically enable/disable its forecaster models at present, users can extend NWSLite (such as to have a minimum version with five predictors, a medium version with ten of them, and a full version) and then choose the best one, depending on the characteristics of the task. For example, if we want to offload a substantially large task, we can use the full version for the whole duration of time. For a smaller task that can be completed in milliseconds, we can use the minimum version, etc. While we did not evaluate such a mechanism within the scope of this work, since NWSLite is already close to NWS in accuracy, the framework that we developed for NWSLite (including the datasets, source code and experimental setup) can easily be used for any future studies toward this direction.

In our work, we evaluate NWSLite using a large set of traces including CPU and network availability and CPU demand. We collect CPU demand traces using augmented reality applications, which we believe will benefit from remote execution. In our offloading scenarios, we evaluate each scene individually, in other words we do not consider the case that the user switches back and forth in between scenes. Thus, in some way, one can argue that we parameterize the predictors with each scene. However, since the CPU demand in each scene varies greatly (for example, in venus the CPU demand changes 200-fold), we believe the impact of such parameterization is insignificant.

Finally, there are mobile applications other than remote execution (for example dynamic voltage scaling) that can benefit substantially from an accurate and low cost predictor. The framework that we developed within the context of this work can be very beneficial to evaluate and improve new predictors for these applications.

7. CONCLUSION AND FUTURE WORK

We present a light-weight, computationally efficient, prediction utility for mobile devices called NWSLite. The system is an extension of the network weather service (NWS), a dynamic measurement and forecasting toolkit designed and developed for adaptive application scheduling in computational-grid environments (performance-oriented distributed systems). We identify 5 of the 24 NWS forecasters for NWSLite implementation, that trade-off computational cost for predictor accuracy most effectively.

We evaluate NWSLite using over 300 different traces of application execution times, CPU availability, wired network bandwidth and latency, and wireless bandwidth. In addition, we compare NWSLite to the NWS and to two other extant remote execution prediction systems. We find that NWSLite consistently outperforms the latter and achieves prediction accuracy similar to that of the

NWS. However, NWSLite achieves this level of accuracy at a significantly lower execution cost than the NWS.

We show the utilization of NWSLite on a computation offloading platform, by evaluating it for resource supply and demand prediction using two computation offloading scenarios. In the first scenario, NWSLite beats two popular predictors, RPF and LSQ, by 67 and 14% fewer wrong decisions, consecutively. In the second scenario, NWSLite beats those predictors even with a higher margin: 95 and 73% fewer wrong decisions. NWSLite achieves this rate without any significant increase in cost.

In future, we are planning to extend our work toward a more scalable predictor that can dynamically scale up/down its computational complexity and prediction accuracy. In addition, we are planning to evaluate NWSLite against other predictors, including the flip/flop predictor that was developed in Kim and Noble [2001]. Finally, we are planning to evaluate NWSLite in other embedded system applications, including dynamic voltage scaling.

REFERENCES

BALACHANDRAN, A., VOELKER, G. M., BAHL, P., AND RANGAN, P. V. 2002. Characterizing user behavior and network performance in a public wireless lan. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*.

BALAN, R. K., SATYANARAYANAN, M., PARK, S. Y., AND OKOSHI, T. 2003. Tactics-based remote execution for mobile computing. In *MobiSys '03: Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*.

BERMAN, F., WOLSKI, R., FIGUEIRA, S., SCHOPF, J., AND SHAO, G. 1996. Application level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing*.

BERMAN, F., FOX, G., AND HEY, T. 2003. *Grid Computing: Making the Global Infrastructure a Reality*. Wiley, New York.

BOTTOMLEY, G. AND ALEXANDER, S. 1991. A novel approach for stabilizing recursive least squares filters. *IEEE Trans. Signal Processing 39*, 1770–1779.

BURGER, D. AND AUSTIN, T. 1997. The simplescalar tool set, version 2.0. Tech. Rept. 1342, UW Madison Computer Sciences. June.

CARD, S., MORAN, T., AND NEWELL, A. 1983. *The Psychology of Human–Computer Interaction*. Lawrence Erlbaum Associates, Mahwah, NJ.

Compaq Computer Corporation. *iPAQ Pocket PC*. Compaq Computer Corporation. http://www.compaq.com/products/handhelds/pocketpc/.

FLINN, J. 2001. Extending mobile computer battery life through energy-aware adaptation. Ph.D. thesis, Carnegie Mellon University.

FLINN, J. AND SATYANARAYANAN, M. 1999. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles*. 48–63.

FLINN, J., NARAYANAN, D., AND SATYANARAYANAN, M. 2001. Self-tuned remote execution for pervasive computing. In *Hot Topics in Operating Systems(HotOS-VIII)*, Germany. 61–66.

FLINN, J., PARK, S., AND SATYANARAYANAN, M. 2002. Balancing performance, energy, and quality in pervasive computing. In *International Conference on Distributed Computing Systems (ICDCS '02)*. 217–226.

FOSTER, I. AND KESSELMAN, C. 1998. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publ. Burlington, MA.

GLVU 2002. GLVU source code and documentation. `http://www.cs.unc.edu/~walk/software/glvu/`.

GOVIL, K., CHAN, E., AND WASSERMAN, H. 1995. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *ACM international conference on Mobile Computing and Networking (MoBiCom)*. 13–25.

GrADS. The grid application development software project (GrADS). http://hipersoft.cs.rice.edu/grads/.

GRUNWALD, D., LEVIS, P., MORREY, C., NEUFELD, M., AND FARKAS., K. 2000. Policies for dynamic clock scheduling. In *Operating System Design and Implementation(OSDI)*. 73–86.

GURUN, S., KRINTZ, C., AND WOLSKI, R. 2003. Efficient Prediction. Tech. Rept., 2003-34, University of California, Santa Barbara.

GURUN, S., KRINTZ, C., AND WOLSKI, R. 2004. Nwslite: A light-weight prediction utility for mobile devices. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*. ACM Press, New York. 2–11.

KIM, M. AND NOBLE, B. 2001. Mobile network estimation. In *Mobile Computing and Networking*. 298–309.

KREMER, U., HICKS, J., AND REHG, J. M. 2001. A compilation framework for power and energy management on mobile computers. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*.

KRINTZ, C., WEN, Y., AND WOLSKI, R. 2004. Application-level prediction of battery dissipation. In *International Symposium on Low Power Electronics and Design*.

LI, Z., WANG, C., AND XU, R.. 2001. Computation offloading to save energy on handheld devices: A partition scheme. In *Proceedings of International Conference on Compilers,Architectures and Synthesis for Embedded Systems (CASES)*. 238–246.

NARAYANAN, D. 2002. Operating system support for mobile interactive applications. Ph.D. thesis, Carnegie Mellon University CMU-CS-02-168.

NARAYANAN, D. AND SATYANARAYANAN, M. 2003. Predictive resource management for wearable computing. In *International Conference on Mobile Systems, Applications, and Services*.

NOBLE, B., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J., FLINN, J., AND WALKER, K. 1997. Agile application-aware adaptation for mobility. In *16th ACM Symposium on Operating Systems Principles*. ACM Press, New York. 276–287.

NWS. The Network Weather Service Home page – http://nws.cs.ucsb.edu.

PERING, T., BURD, T., AND BRODERSEN, R. 1998. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of International Symposium on Low Power Electronics and Design*. 76–81.

RUDENKO, A., REIHER, P., POPEK, G., AND KUENNING, G. 1998. Saving portable computer battery power through remote process execution. *Mobile Computing and Communications Review 2*, 1 (Jan.), 19–26.

RUDENKO, A., REIHER, P., G.POPEK, AND G.KUENNING. 1999. The remote processing framework for portable computer power saving. In *ACM Symposium on Applied Computation* San Antonio,TX.

Sigcomm01traces. Wireless LAN Traces from ACM SIGCOMM'01. http://ramp.ucsd.edu/pawn/sigcomm-trace/.

SINHA, A. 2001. Energy efficient operating systems and software. Ph.D. thesis, Massachusetts Institute of Technology.

SINHA, A. AND CHANDRAKASAN, A. P. 2001. Dynamic voltage scheduling using adaptive filtering of workload traces. In *Proceedings of the The 14th International Conference on VLSI Design (VLSID '01)*. IEEE Computer Society, Los Alamitos, CA. 221.

SPRING, N. AND WOLSKI, R. 1998. Application level scheduling: Gene sequence library comparison. In *Proceedings of ACM International Conference on Supercomputing*.

SUCU, S. AND KRINTZ, C. 2003. ACE: A resource-aware adaptive compression environment. In *International Conference on Information Technology: Coding and Computing (ITCC)*.

SWANY, M. AND WOLSKI, R. 2002. Representing dynamic performance information in grid environments with the network weather service. In *2nd IEEE International Symposium on Cluster Computing and the Grid* (Berlin).

WEISER, M., WELCH, B., DEMERS, A. J., AND SHENKER, S. 1994. Scheduling for reduced CPU energy. In *Operating Systems Design and Implementation*. 13–23.

WILLMOTT, A. J. 1999. Radiator source code and online documentation. http://www.cs.cmu.edu/~ajw/software/.

WOLSKI, R. 1998. Dynamically forecasting network performance using the network weather service. *J. Cluster Comput. 1*, 119–132.

WOLSKI, R. 1999. Predicting CPU availability on the computational grid using the network weather service. *J. Parallel Processing Lett. 9*, 4, 227–241.
WOLSKI, R. 2003. Experiences with predicting resource performance on-line in computational grid settings. *SIGMETRICS Perform. Eval. Rev. 30*, 4, 41–49.
WOLSKI, R., SPRING, N., AND HAYES, J. 1999. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems 15*, 5,6, 757–768.
YOUNG, P. 1984. *Recursive Estimation and Time-Series Analysis*. Springer-Verlag, New York.

Received September 2006; revised February 2007; accepted May 2007

Title: NWSLite: A General-Purpose, Nonparametric Prediction Utility for Embedded Systems

Authors: Selim Gurun, Chandra Krintz, and Rich Wolski

Author Queries

AQ1: Au: Pls check shorten running head is ok?
AQ2: Au: OK?
AQ3: Au: OK italic?
AQ4: Au: OK italic?
AQ5: Au: OK?
AQ6: Au: OK?
AQ7: Au: OK?
AQ8: Au: OK?