NWSLite: A Light-Weight Prediction Utility for Mobile Devices

Selim Gurun Chandra Krintz Rich Wolski
Computer Science Department
University of California, Santa Barbara
{gurun,ckrintz,rich}@cs.ucsb.edu

ABSTRACT

Computation off-loading, i.e., remote execution, has been shown to be effective for extending the computational power and battery life of resource-restricted devices, e.g., hand-held, wearable, and pervasive computers. Remote execution systems must predict the cost of executing both locally and remotely to determine when off-loading will be most beneficial. These costs however, are dependent upon the execution behavior of the task being considered and the highly-variable performance of the underlying resources, e.g., CPU (local and remote), bandwidth, and network latency. As such, remote execution systems must employ sophisticated, prediction techniques that accurately guide computation off-loading. Moreover, these techniques must be efficient, i.e., they cannot consume significant resources, e.g., energy, execution time, etc., since they are performed on the mobile device.

In this paper, we present NWSLite, a computationally efficient, highly accurate prediction utility for mobile devices. NWSLite is an extension to the Network Weather Service (NWS), a dynamic forecasting toolkit for adaptive scheduling of high-performance Computational Grid applications. We significantly scaled down the NWS to reduce its resource consumption yet still achieve accuracy that exceeds that of extant remote execution prediction methods. We empirically analyze and compare both the prediction accuracy and the cost of NWSLite and a number of different forecasting methods from existing remote execution systems. We evaluate the efficacy of the different methods using a wide range of mobile applications and resources.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Modeling and prediction, Measurements*

General Terms

Measurement, Performance

Keywords

Prediction, Network Performance Estimation, CPU Availability Estimation, Fidelity, Remote Execution, Resource-restricted devices

MobiSYS'04, June 6–9, 2004, Boston, Massachusetts, USA.

1. INTRODUCTION

Remote execution is emerging as a promising technique to extend the computational power and battery life of resource-restricted environments, e.g., hand-held, wearable, and pervasive computers. Using remote execution, parts of program execution are off-loaded from battery-powered mobile devices to wall-powered, higher performance platforms. As such, remote execution can significantly extend the usefulness of devices by enabling execution of a wide-range of resource-intensive applications, e.g., augmented reality, natural language translation, feature recognition, collaborative computing, etc., in a mobile environment.

Key to the successful implementation of remote execution, is *prediction*. That is, we must predict when the cost of performing remote execution will not outweigh its benefits. For example, if a task will take significantly less time if executed remotely due to superior resource performance, the system should off-load the task. Alternately, if the cost of remote execution will exceed that of local execution, e.g., if significant amounts of data must be transfered to perform the task or if the remote host is busy, then the system should perform the operation locally.

The cost of a remotely executed operation consists of the time to transfer data (and possibly code) from the device to the target, the time to transfer result information, e.g., data, status, rendered graphics, etc., from the target back to the device, and the time to execute the operation at the target. The cost of a locally executed operation is the local execution time. These costs can be decomposed or translated to consider other metrics, e.g., battery consumption, response time, application fidelity. However, regardless of their form, these values must reflect what the costs *will be* when the operation is eventually performed.

To complicate matters, each of these costs is dependent upon the highly variable performance available from the underlying resources as well as the application input data. Network bandwidth and latency dictate the time required for communication and CPU availability on both the device and the target impacts local and remote execution time, respectively. Moreover, execution cost is dependent upon the length of time the application tasks will execute, which is commonly dependent upon program inputs.

To predict these costs, extant remote execution systems employ statistical techniques that use past behavior to predict the future [23, 8, 10, 2]. The goal of these systems is to enable high prediction accuracy. That is, they attempt to reduce prediction error – the difference between predicted values and the measurement values that they predict. Techniques that result in large prediction errors can cause incorrect decisions to be made about the best execution choice for the device. As such, prediction accuracy plays an important role in the performance of a remote execution system.

However, another characteristic of prediction techniques that is

often not considered, is computational cost – performing the prediction itself consumes device resources. This cost can be high (particularly in terms of power consumption) since statistical techniques commonly use floating-point operations and most mobile devices do not implement a floating-point co-processor. Instead, they rely on software emulation of floating-point instructions making them highly resource-intensive operations.

In this paper, we consider both the accuracy and cost of commonly used forecasting technologies in remote execution settings. We evaluate techniques that are currently used for remote execution and present a novel resource performance prediction utility for resource-restricted 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 [11] environments [30, 33, 4, 29, 28]. We modified the NWS forecasting model to reduce its resource consumption footprint to enable its use in a mobile setting.

NWSLite can be incorporated into any mobile framework that uses prediction. It makes non-parametric 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 as well as to two prediction algorithms (described in [23] and [21]) currently used for prediction in remote execution frameworks. 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 less computational resources than its predecessor (the original NWS).

In summary, the contributions we make with this paper are:

- A light-weight, highly-accurate, prediction utility for resource-restricted environments,
- A predictive framework that uses algorithms that are *non-parametric*, i.e., they do not require externally determined, resource-specific parameters or manual fine-tuning,
- A general utility for mobile devices that can be used to forecast efficiently the performance of a wide range of resources,
- A comparison of the prediction accuracy and execution cost of the NWSLite to popular predictive techniques, and
- An empirical evaluation that shows that NWSLite can significantly improve prediction accuracy for remote execution.

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

2. BACKGROUND

Remote execution is a popular technique that is used to extend the computational capability of mobile, resource-restricted, devices [8, 24, 23, 16, 35]. Figure 1 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.

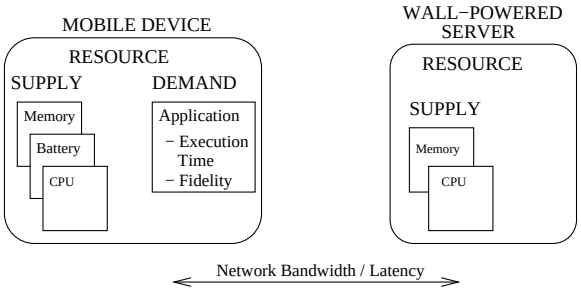

Figure 1: Components of a typical remote execution system. The decision process includes forecasting the available resource supply both at the client and server and application resource demand.

To decide whether a particular task should be off-loaded, a remote execution system must first compute the *demand* of the application task. In this work, we define demand as task execution time, however, additional constraints can be applied to this value, e.g., response time, computational fidelity, power consumption, etc., 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 exceeds that of remote execution (including all necessary communication), the system off-loads work to the server. The cost model must consider both the task execution characteristics as well as the highly-variable performance of the underlying resources that dictate computation and communication performance. This cost model can be formulated in terms of time as:

$$ET_L/CPU_L \leq$$

$$\frac{SZ_{input}}{BW_{L-R}} + (ET_R/CPU_R) + \frac{SZ_{output}}{BW_{R-L}} + (HS * LAT)$$

where the local cost is the time to execute the task locally on the mobile device (ET_L) given the available fraction of the CPU on the device (CPU_L). The remote cost is the sum of the time required for four constituent operations:

1. The time required for transfer of code and data required for execution (SZ_{input} bytes) given the available bandwidth between the device and the remote server (BW_{L-R});
2. The execution time at the server given the fraction of CPU available at the server (ET_R/CPU_R);
3. The time for transfer of results, e.g., data, status, rendered graphics, etc., (SZ_{output} bytes) back to the device given the available bandwidth between the server and device (BW_{R-L}); and;
4. The time for any handshake protocol to establish remote execution between the device and the server. This protocol commonly consists of very few, small packets (HS); as such, its communication is impacted by the network latency between the device and server (LAT).

Remote execution systems can consider other metrics, e.g., computational fidelity, response time, battery power, by extending this cost model. For example, the execution time at either end can be evaluated for different fidelity levels or broken down into fine-grain tasks to consider response time. Alternately, the execution time, transfer, and idle time at the device can be translated into battery consumption for the device.

The cost model must compute what the cost of remote and local execution *will be* when the task is eventually executed. That is, remote execution systems must *predict* this cost *on the device* to determine when to off-load. As such, these systems must employ sophisticated forecasting techniques to enable accurate prediction of the cost of remote (and local) execution. We describe the prediction technologies used in extant remote execution systems in the next section.

3. PREDICTION ALGORITHMS FOR REMOTE EXECUTION

Two advanced, prediction-based, remote execution systems to which we compare our work, are Spectra [8] and the remote processing framework (RPF) by Rudenko et al [23]. We briefly describe these systems in this section and discuss the predictors that each employs.

3.1 Spectra and Odyssey

Spectra is the remote execution component of the Aura [27] pervasive computing environment. Spectra allows the user to dictate goals, such as minimizing energy use, and then tries to achieve this goal by means of local, remote or hybrid, i.e., partially remote, execution decisions. To estimate the behavior of an application, Spectra (running on the device) employs the resource monitoring and performance prediction functionality implemented in Odyssey, a second Aura component that implements a set of operating system extensions to support mobility and application adaptation [21].

Odyssey uses *multi-computational fidelity* to dynamically tradeoff resource demand for application quality [9]. The fidelity adjustments are performed at the granularity of an *operation*, which is the smallest unit of execution that can be perceived by the user as a response to a request [20].

Since the performance of off-loading is dependent upon the underlying resource availability, Odyssey couples task demand with resource supply. That is, Odyssey makes forecasts of the underlying resource performance for CPU, memory, network bandwidth, and network latency. Odyssey couples these predictions with those from the fidelity resource functions to identify opportunities for dynamic off-loading.

Odyssey estimates CPU availability using process counts; it assumes that CPU cycles are evenly distributed across all processes and computes the percent of CPU available as $\frac{1}{P+1}$, where P is the predicted number of processes that will be executing when the new process is added. To compute P, Odyssey counts the number of processes at time t and estimates the number of processes at a future time $t+1$ using a smoothing filter computed as:

$$P_{t+1} = \alpha P_t + (1-\alpha)(n_i - p) \quad (1)$$

where n_i is the average CPU load periodically sampled from `/proc/loadavg`, p is the load consumed by the process in question (1 if running, else 0), and α is $e^{\frac{-t_p}{T}}$, where t_p is the sampling period (a 0.5 second period was used in [20]) and T is the prediction horizon – this causes more history data to be considered for longer-term predictions [20].

To predict network bandwidth and latency, Odyssey uses a different prediction model based on exponential smoothing:

$$f_{t+1} = \gamma(m_t) + (1-\gamma) f_t \quad (2)$$

The function predicts the performance of the resource at the next time step (f_{t+1}) as the sum of weighted values for the measurement at time t (m_t) and the forecast made for time t (f_t). Odyssey employs weighting (gain) factors (γ) of 0.75 for round trip time and 0.875 for throughput. This formulation combines the current performance with an aggregation of the previous prediction history to make the prediction.

An exponential smoothing predictor with a constant gain factor does not adapt dynamically to the changes in the system. To address this issue Kim et.al. [15] developed a flip-flop filter. The flip-flop filter employs two exponential smoothing predictors: one with $\gamma = 0.1$ and the other with $\gamma = 0.9$. Using a small gain factor, the former can respond to changes more quickly than the latter one. However, a small gain is more susceptible to a transient noise. As such, by using both gain factors, the filter is able to achieve the benefits from both while avoiding the limitations of each. Our prediction methodology, described in the next section, takes a similar but more general approach.

Finally, to estimate the CPU demand (execution time) of applications, Odyssey and Spectra use an online-learning predictor. The predictor maintains coefficients, specific to each program input, that it uses to model the cost behavior of the application when executed using each input. However, computing the initial coefficient values requires off-line training. The online-learning algorithm then updates coefficients using recursive least squares regression with exponential decay. Due to exponential decay characteristics, more weight is given to the recent observations.

The recursive least squares method is an efficient way to 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 is given by:

$$A_k = A_{k-1} - P_k\{x_k x_k^T A_{k-1} - x_k y_k\}$$

$$P_k = \{P_{k-1} - P_{k-1}x_k[\alpha + x_k^T P_{k-1} x_k]^{-1} x_k^T P_{k-1}\}/\alpha$$

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* [34].

Recursive least squares estimation is not specific application resource consumption (CPU demand), i.e., it is a general technique for estimation of any type of time-series data. However, it is particularly amenable to application that can be decomposed into tasks with similar execution time, e.g., scene rendering in an augmented reality application given multiple scenes being scanned by a camera. For such applications, the resource consumption behavior for the generation of a scene will be similar to that of a neighboring scene. However, if there is no correlation between the resource consumption of application tasks, this methodology may result in large prediction errors. Another limitation of this type of statistical technique is that numerical computation errors can accumulate after each recursion causing algorithm to become unstable and diverge [5].

The value of the exponential decay factor determines the agility of the method. A smaller value increases responsiveness, but decreases the amount of noise that can be filtered out. Odyssey uses a decay factor of 0.5, which makes it very agile. Our observations show that, with such a small value the method can become unstable and diverge if the observed data is noisy or moderately variable. Based on our experimentation using several different parameters on

a large dataset, we found that a decay factor of 0.8 is more appropriate and as such, use it in our experimental results. We consider the efficacy of recursive least squares for a wide range of resource types (including CPU demand) in our empirical evaluation section; we refer to this method as *LSQ* in our experimental results.

3.2 Remote Processing Framework (RPF)

The remote processing framework (RPF) uses a different execution model than that of Spectra and other remote execution systems [23]. RPF models a single metric – power consumption – of the executing task. That is, it collects history data on the power consumption of previous tasks and uses it to predict the consumption of future tasks. The comparison between local and remote task execution is performed on the device itself. All code and data are shared and kept synchronized between the device and server. The server simply processes the tasks transmitted by the device and returns the results.

The device and server interact via a simple handshake protocol. During this protocol, the device verifies that the server has enough resources and the appropriate software packages. Following this, the device predicts whether off-loading will reduce power consumption. The RPF system uses a *smoothing filter* to make its forecasts given prior history. In particular, the RPF cost model is the *parameterized* function:

$$f_{t+1} = (1-\alpha)\frac{\sum_{i=n-k}^{n} v_i}{k} + \alpha f_t \tag{3}$$

where f is the forecasted value (at time t and $t+1$), and v_i is the measured value at measurement index i.

RPF is parameterized with two parameters, k and α which determine how conservative the forecaster is: A small k combined with a large α will result in higher responsiveness to recent changes. Unfortunately, the authors in [23] do not specify default values for these parameters, nor do they discuss the values that they used. To enable our evaluation and comparison to this technique, we empirically evaluated several different combinations of parameters for a large set of data. We then selected best-performing set across all experiments. For the data we detail in this study, the best overall parameterization is $k = 20$ and $\alpha = 0.0$. With this parameterization, RPF becomes a "sliding window average" over a fixed window (size k) of previous data history. The data for all experiments (including that for our different parameterizations) can be found in the technical report version of this paper [14].

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

4. NWSLITE

Odyssey and RPF employ parameterized prediction methodologies to forecast the cost of local and remote execution. The parameters are identified through empirical evaluation and are specific not only to the executing application but also to the individual tasks. In addition, different types of resource data (task execution time and characteristics, and network and CPU performance) require different predictors or different parameterization of the same predictors to be effective. Moreover, prediction of task execution behavior in Odyssey requires extensive off-line training. Our approach to the problem of remote execution cost forecasting takes a different approach. Specifically, it is one that is *non-parametric, automatic, and completely general*.

To enable this, we developed NWSLite, an accurate and efficient prediction utility for mobile devices. NWSLite is an extension of the Network Weather Service [32], a freely available toolkit [22], originally developed for Computational Grid computing [11, 3]. The Computational Grid is a computing paradigm for the development of software systems that enables dynamic acquisition of resources from a heterogeneous and non-dedicated resource pool. 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, applies a set of statistical forecasting techniques to individual performance histories, and generates forecast reports for the resources being monitored, which it disseminates via a number of different APIs in near-real-time [33]. 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 of 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 implementation of NWS that we extended uses the 24 prediction models shown in Table 1.

This mixture-of experts method achieves its accuracy by employing wide range of statistical models, each of which may be most appropriate at a given time, for a given resource. This method 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. 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.

Because the NWS was originally designed to support 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 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, prediction error can be defined using the square of the errors:

$$E = \sum_{i=1}^{n} (f_i - v_i)^2 \tag{4}$$

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

	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

Table 1: NWS Forecasters and the Approximate Costs of Each. We show cost in column three as the number of floating point operations performed.

$s_{i,j}$ to denote the winning forecaster:

$$s_{i,j} = \left\{ \begin{array}{ll} 1 & \text{If model } j \text{ is used to predict} \\ & \text{measurement } i \\ 0 & \text{Otherwise} \end{array} \right\} \quad (5)$$

Specifically, if $s_{i,j}$ is 1, the i^{th} forecast is made using predictor j. If $s_{i,j}$ is 0, the predictor is not the winner for the i^{th} forecast. If we set k to be the number of models in NWS, using (4), we can formulate prediction error of NWS as:

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

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

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

Theoretically, it is possible to optimize NWS by running it with 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 used a heuristic that evaluates how much the total computation cost and error would change if a forecaster u is substituted with another forecaster v throughout the series.

Formally, this process can be expressed as:

$$s'_{i,j} = \left\{ \begin{array}{ll} 1 & \text{If model } j \text{ is winner forecaster} \\ & \text{for measurement } i \text{ and } j \neq u \\ 1 & \text{if model } j \text{ is not winner forecaster} \\ & \text{for measurement } i \text{ and } j = v \\ 0 & \text{Otherwise} \end{array} \right\} \quad (8)$$

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

We computed $E_{u,v}$ and $C_{u,v}$ for every pair of u and v on a set of six representative traces and represented it as matrix with u as rows and v as 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 would change if v had been used instead of u. For example, $E_{2,1}$ shows the new error if *last value* is used 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 *running mean* outperforms the original NWS. Even though, it is theoretically possible, we did not come across an example of such a case.

Our methodology is similar to off-line, profile-based optimization research in which a set of representative program inputs are used to collect profile information that is used to guide optimization [18, 26, 17]. Empirical experiments then use a different set of inputs to evaluate the efficacy of the technique. Here, we used six traces to identify NWS forecasters that enable high accuracy at low cost. We then evaluated NWSLite using over 300 different traces. Given the evaluation matrix, we used a set of empirical rules with which we eliminated forecasters. We removed any model

- that had more than 1% error rate across all traces,
- for which there is another model with significantly lower cost that can replace it, with a slight but acceptable (less than 5%) increase in error rate, and
- for which there is a combination of other models that enable a similar error rate.

We ruled out many models directly using the first criteria. For example, replacing *30% trimmed median window 31* with *running mean* generated an increase in error rate of at most 0.2%. On the other hand, for *median window 31* the *running mean* was only 0.2% higher in 5 of the 6 traces, except the last for which it had an error rate that was 32% higher. In the remaining trace, *median window 5* had almost the same error rate. As such, we included *median window 5* and omitted *30% trimmed median window 31*. Eventually, we identified five predictors (shown in bold in Table 1) that trade off cost and prediction error most effectively.

5. EVALUATION

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 5.2, and present our results using these metrics in Section 5.3.

5.1 Experimental Methodology

As mentioned previously, a remote execution system must *predict* when to off-load work from a battery-powered, mobile device to a remote server. To determine this, the system must estimate the cost of performing the computation remotely and locally and then compare the two results. Each component of the cost model used for this comparison requires forecasts of what the underlying resource performance *will be* when the task is executed, e.g., CPU availability for execution time, network performance for communication time, etc.

To empirically compare the resource forecasting system that we present in this paper, NWSLite, to extant approaches to resource performance prediction for remote execution systems, we collected

Name	Trace Size	Description
Application	20 traces 17870 predictions	Interactive, 3-D rendering application CPU demand. Measurements are CPU time from user request to program response.
Network Bandwidth [22]	132 traces 750476 predictions	Observations of 64Kb-1Mbyte TCP data transfers. 3 configurations: UIUC LAN (inter-cluster), UIUC campus-wide network (intra-cluster), and cross-country Internet (UIUC-UCSD)
CPU load [22]	59 traces 6000697 predictions	Fraction of CPU occupancy time a standard user process can obtain Observations are in 10 seconds intervals.
Network Latency [22]	134 traces 750305 predictions	Round trip time of TCP. Transferring 4 bytes and measuring acknowledge time. Granularity levels same as network bandwidth.
Wireless Bandwidth [25]	1 trace 3028 predictions	4 access points on same subnet. Traces include 195 users, 300000 flows and 4.6 GB of network traffic. Bandwidth measured in 1 minute intervals

Table 2: Datasets Used for Evaluation

	Applications	
Input Scene	GLVU	Radiosity
castle	Yes	
cessna	Yes	Yes
chevy	Yes	
cloister	Yes	
cup		Yes
dragon		Yes
ground-table-land	Yes	Yes
ground-riverain-valley		Yes
shuttle	Yes	Yes
venus		Yes

Table 3: Applications and Inputs Used for Evaluation. We collected 10 trace files per application (3-D 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 10) using different navigation paths.

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 7 million predictions. All of the traces, with the exception of application execution times, were made freely available to us via web-sites of research groups around the country [22, 1, 13]. We provide the details on the different datasets in Table 2 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 [20, 19]. Such applications implement rendering technologies that are a key component of augmented reality applications. Such applications are highly suitable for remote execution: Independent tasks used in 3-D scene rendering and image fidelity adjustment are computationally-intensive and easily divided into components for offloading. The applications and inputs that we considered are shown in Table 3.

GLVU [12] allows navigating inside a 3-D scene by rendering the scene from any viewpoint of user. From an augmented reality view, Radiator [31] complements GLVU by computing the lighting effects for a given scene. Both applications can easily be divided into *operations* [20], which are a suitable unit for remote execution and 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 10 scenes which produced a total of 17870 operations. We employed all of the inputs shown in Table 3; 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 [22]. 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 KByte 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 one, five and fifteen minutes. The sensor uses the average load over the one minute 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 [25]. The conference building was covered with four 802.11b access points. The traces span a 3 day period capturing 300000 flows generated by 195 users consuming a total of 4.6 GB of bandwidth.

5.2 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:

$$MSE = \frac{\sum_{i=1}^{n}(x_i - y_i)^2}{n}$$

$$\text{Error deviation} = \sqrt{MSE} \tag{9}$$

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

Description	Units	Avg	NWSLite	NWS	LSQ	RPF
APP1 - best		148845.000	5287.856	5358.179	8180.561	22013.694
APP2 - median	msecs	169753.000	135125.056	138064.335	145384.166	186430.176
APP3 - worst		9179.390	1322.139	1329.372	2385.072	5702.085
BW1 - within cluster		65.801	17.161	16.958	52.112	17.191
BW2 - cross-cluster	Mbits/sec	76.522	13.308	13.329	59.279	13.507
BW3 - cross-country		4.536	0.878	0.859	78.063	1.164
CPU1 - best		1.992	0.016	0.016	13.905	0.029
CPU2 - median	CPU	0.543	0.017	0.017	14.451	0.049
CPU3 - worst	fraction	1.391	2.672	2.684	3.113	2.661
LAT1 - within cluster		13.936	16.873	16.890	41.121	17.048
LAT2 - cross-cluster	msecs	2.345	8.309	8.319	46.829	8.337
LAT3 - cross-country		77.217	14.295	12.753	81.820	13.149
WBW	Kbytes/sec	206.674	193.782	194.498	255.254	261.744

Table 4: Error Deviation for a Set of Representative Traces. 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.

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**:

$$\text{Relative error rate} = \frac{\sqrt{MSE}}{observed\_mean} \tag{10}$$

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 2Mb/s is large in a 10Mb/s link, but may not be significant in a 100Mb/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 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:

$$\frac{\sum_{i=1}^{n} \frac{|x_i - y_i|}{|x_i|}}{n} \tag{11}$$

5.3 Results

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 [20, 9, 21] as well as our own values, resulting in 18 different forecasters. We selected the best performing parameterization for each over all of the datasets we considered.

We were unable to include the complete set of results due to space constraints; however, they can be found in the technical report version of this paper [14]. In the subsections that follow, we first present results for prediction accuracy and then evaluate the computational cost of each prediction system.

5.3.1 Predictor Accuracy

Table 4 compares the **error deviation** (Equation 9) 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: intra-cluster, inter-cluster, and inter-campus (across country). For wireless (WBW), we only have a single trace and thus show data only for it.

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 5, 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 [20]), 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 3 times better than RPF. The wireless dataset is especially challenging. All the forecasters show a high error rate.

Figure 2 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 4. 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

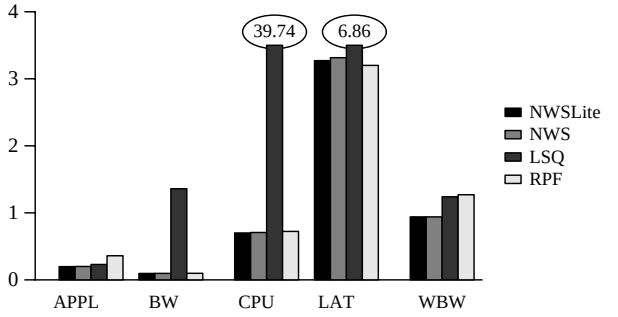

Figure 2: Relative Error Rate (Equation 10). This metric shows how severe the error is with respect to the average measured value. The LAT has the highest relative error rate among all forecasters, however, as most latency observations are very small (around 1 msecs), the absolute error is small.

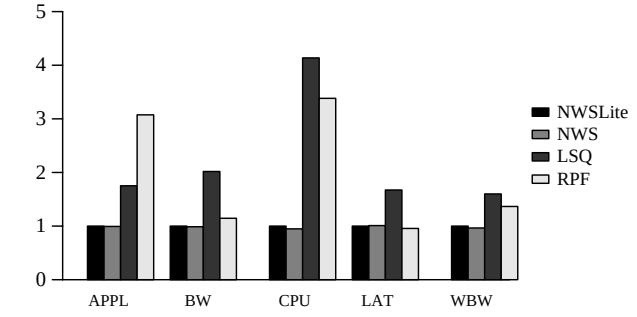

Figure 3: Predictor Predictability (Equation 11). Due to high variation among forecasters, the values are normalized to NWSLite for each group. The lower the value, the more accurate the forecaster.

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 3 shows the **predictability** (Equation 11) 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.

The predictability results support our findings in Figure 2. NWSLite 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 smoothing-filters. 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 2 and 3; even though the relative error rate of RPF is small, its predictability is not. This is due to 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.

5.3.2 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 [6] simulator. Figure 4 shows the average cost of each predictor. NWSLite uses 55 floating point instructions per forecast. Even though this is more than the cost of RPF and LSQ, which use 8 and 42, respectively, the accuracy of NWSLite exceeds both of these predictors significantly.

Prediction System	Floating Point	Total Instructions	Execution time (microsecs)
NWSLite	55	592	381.34
NWS	2626	9388	10231.31
LSQ	42	138	295.27
RPF	8	50	154.9

Table 5: Execution Cost Comparison per Prediction

As most resource-restricted devices lack a floating point co-processor, floating point instructions are very expensive. We break down the instruction counts into floating-point and non-floating-point instructions in the first two columns of Table 5.

We also executed the predictors on a real resource-restricted device: An iPAQ H3800 hand-held computer from Compaq [7]. The iPAQ has a 206 MHz Intel StrongArm CPU and runs Familiar Linux version 0.5.3. The execution times (in microseconds) 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

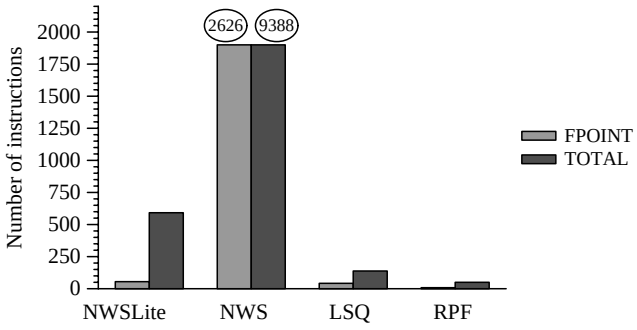

Figure 4: Forecaster Cost as Number of Instructions Executed (floating-point (FPOINT) and TOTAL) per Prediction

	APP		BW		CPU		LAT		WBW	
	E90	E95	E90	E95	E90	E95	E90	E95	E90	E95
NWSLite	3319.000	7336.000	10.271	25.699	0.019	0.043	15.772	24.566	198.130	351.090
NWS	3343.000	7459.000	9.601	25.580	0.018	0.038	15.801	24.502	202.771	358.798
LSQ	5866.552	13305.338	14.105	28.459	0.058	0.115	16.415	26.867	230.591	422.977
RPF	17147.400	38696.700	10.596	25.561	0.080	0.209	16.187	24.915	326.340	533.047
ODY-LAT	3759.839	8806.945	9.923	39.717	0.025	0.094	16.318	29.848	197.429	335.172
ODY-BW	3458.320	7894.141	7.384	42.541	0.021	0.079	16.883	31.494	192.992	354.560

Table 6: Results in Summary: Percentile Error. 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 5 datasets and prediction systems studied.

NWS but enables prediction accuracy that is nearly equivalent. Given that it requires only 381 microseconds 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.

5.4 Result Summary

We summarize the result of our findings in Table 6. To make our results comparable to previous studies [20], 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/second. The 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, 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 200Kbits/sec on a 11Mbits/sec 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 pre-determined 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 resource performance data.

The flip-flop filter extension to Odyssey [15], 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.

6. CONCLUSIONS

By off-loading tasks from the resource-restricted devices to wall-powered, high-performance servers, remote-execution can significantly extend the capability and battery life of mobile and pervasive devices. To determine when to offload, these devices must make forecasts about the efficacy of doing so. A device must estimate the cost of both remote and local execution given the highly-variable underlying resource performance as well as the characteristics of the task to be executed. As such, it must employ sophisticated prediction techniques that are both accurate and light-weight, i.e., they do not consume significant device resources.

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.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments for the final version of this paper. In addition, we would like to thank Dushyant Narayanan and Rajesh Krishna Balan from Carnegie Mellon University for providing us with source code for GLVU and Radiosity. This work was funded in part by NSF grant No. EHS-0209195, and an Intel/UCMicro external research grant.

7. REFERENCES

[1] A. Balachandran, G. Voelker, P. Bahl, and P. Rangan. Characterizing user behavior and network performance in a public wireless lan. In *ACM International Conference on Measurement and Modeling of Computer Systems*, 2002.

[2] R. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *International Conference on Mobile Systems, Applications, and Services*, 2003.

[3] F. Berman, G. Fox, and T. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. Wiley and Sons, 2003.

[4] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G Shao. Application level scheduling on distributed heterogeneous networks. In *Supercomputing 1996*, 1996.

[5] G. Bottomley and S. Alexander. A novel approach for stabilizing recursive least squares filters. *IEEE Transactions on Signal Processing*, 39, 1991.

[6] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, UW Madison Computer Sciences, June 1997.

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

[8] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-tuned remote execution for pervasive computing. In *Hot Topics in Operating Systems(HotOS-VIII)*, pages 61–66, Germany, 2001.

[9] J. Flinn, S. Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *International Conference on Distributed Computing Systems (ICDCS '02)*, pages 217–226, 2002.

[10] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles*, pages 48–63, 1999.

[11] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1998.

[12] GLVU source code and documentation, Feb 2002. `http://www.cs.unc.edu/~walk/software/glvu/`.

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

[14] S. Gurun, C. Krintz, and R. Wolski. Efficient Prediction. Technical Report 2003-34, University of California, Santa Barbara, 2003.

[15] Minkyong Kim and Brian Noble. Mobile network estimation. In *Mobile Computing and Networking*, pages 298–309, 2001.

[16] U. Kremer, J.Hicks, and J.M.Rehg. A compilation framework for power and energy management on mobile computers. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, August 2001.

[17] C. Krintz. Coupling On-Line and Off-Line Profile Information to Improve Program Performance. In *International Symposium on Code Generation and Optimization (CGO)*, March 2003.

[18] C. Krintz and B. Calder. Using Annotation to Reduce Dynamic Optimization Time. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–167, June 2001.

[19] D. Narayanan. *Operating System Support for Mobile Interactive Applications*. PhD thesis, Carnegie Mellon University CMU-CS-02-168, August 2002.

[20] Dushyanth Narayanan and M. Satyanarayanan. Predictive resource management for wearable computing. In *International Conference on Mobile Systems, Applications, and Services*, 2003.

[21] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile application-aware adaptation for mobility. In *sixteenth ACM symposium on Operating systems principles*, pages 276–287. ACM Press, 1997.

[22] The Network Weather Service Home page – `http://nws.cs.ucsb.edu`.

[23] A. Rudenko, P. Reiher, G.Popek, and G.Kuenning. The remote processing framework for portable computer power saving. In *ACM Symp. Appl. Comp.*, San Antonio,TX, February 1999.

[24] A. Rudenko, P. Reiher, G. Popek, and G. Kuenning. Saving portable computer battery power through remote process execution. *Mobile Computing and Communications Review*, 2(1):19–26, January 1998.

[25] Wireless LAN Traces from ACM SIGCOMM'01. `http://ramp.ucsd.edu/pawn/sigcomm-trace/`.

[26] M. Smith. Overcoming the challenges to feedback-directed optimization. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo00)*, January 2000.

[27] J. Sousa and D. Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *3rd Working IEEE/IFIP Conference on Software Architecture*, pages 29–43, 2002.

[28] N. Spring and R. Wolski. Application level scheduling: Gene sequence library comparison. In *Proceedings of ACM International Conference on Supercomputing*, July 1998.

[29] S. Sucu and C. Krintz. ACE: A Resource-Aware Adaptive Compression Environment. In *International Conference on Information Technology: Coding and Computing (ITCC)*, April 2003.

[30] Martin Swany and Rich Wolski. Representing Dynamic Performance Information in Grid Environments with the Network Weather Service. In *2nd IEEE International Symposium on Cluster Computing and the Grid*, May 2002.

[31] A. J. Willmott. Radiator source code and online documentation, Oct 1999. http://www.cs.cmu.edu/ ajw/software/.

[32] R. Wolski. Dynamically Forecasting Network Performance Using the Network Weather Service. *Journal of Cluster Computing*, 1:119–132, January 1998.

[33] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, 1999.

[34] P. Young. *Recursive Estimation and Time-Series Analysis*. Springer-Verlag, 1984.

[35] Z.Li, C.Wang, and R.Xu. Computation offloading to save energy on handheld devices:a partition scheme. In *Proc. of International Conference on Compilers,Architectures and Synthesis for Embedded Systems (CASES)*, pages 238–246, 2001.