Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems
and Applications, Monterey, CA, December 2000

Using History to Improve Mobile Application Adaptation

Dushyanth Narayanan, Jason Flinn, and M. Satyanarayanan
School of Computer Science
Carnegie Mellon University
{bumba,jflinn,satya}@cs.cmu.edu

Abstract

Prior work has shown the value of changing application fidelity to adapt to varying resource levels in a mobile environment. Choosing the right fidelity requires us to predict its effect on resource consumption. In this paper, we describe a history-based mechanism for such predictions. Our approach generates predictors that are specialized to the hardware on which the application runs, and to the specific input data on which it operates. We are able to predict the CPU consumption of a complex graphics application to within 20% and the energy consumption of fetching and rendering web images to within 15%.

1. Introduction

A key strategy in mobile computing is adapting application behavior to resource availability and user goals. Changing application *fidelity* — the quality of results presented to the user — has been shown to be effective in adapting application resource consumption to varying resource availability [7, 11, 12]. Fidelity is an application-specific notion of the "goodness" of a computed result or data object: for example, the JPEG Quality Factor of a lossily compressed image, or the precision bound of a floating point computation. Naturally, there is a tradeoff between fidelity and resource consumption: a lower fidelity results in a lower resource consumption, but at the cost of presenting a more degraded result to the user. Fidelity is not always a single real number: there could be multiple fidelity metrics, each of which could be discrete or continuous.

The ultimate goal of fidelity adaptation is to improve a mobile user's computing experience by delivering results quickly, with low battery drain and little distraction of the user. Consider a graphics computation that operates on a 3-D model in a mobile augmented reality application. The latency of the computation depends both on the CPU consumed by the computation and the CPU demands of other applications. The CPU consumption depends on the fidelity — the resolution of the model. If we could predict the CPU consumption as a function of fidelity, we could combine this with CPU load information to predict latency. This lets us characterize the tradeoff between fidelity and latency, and to pick good operating points: for good interactive response, we might always pick the highest fidelity that keeps the latency below 200 ms.

In this paper we show how *history-based prediction* enables the system to learn an application's behavior and predict its resource consumption. We have augmented *Odyssey* [11], an operating system platform for adaptation, with a history-based prediction system that monitors, logs, and predicts application resource consumption as a function of the fidelity. Our initial experience suggests that history-based prediction is feasible. We can predict to within 20% the CPU consumption of a 3-D graphics computation — typical of those found at the heart of augmented reality applications. We can also predict the energy consumption of fetching lossily compressed web images to within 15%. Our current prototype has a CPU overhead of 0.22% for a typical application; we expect the overhead to be even lower in a production version of the code.

2. Design Rationale

Our approach to predicting resource consumption as a function of fidelity is an *empirical* one: we sample the fidelity space by running the application at different fidelities; then we record the resource consumption at each sample point; finally we use machine learning algorithms to make predictions based on the set of samples.

One could imagine an *analytic* approach to the same problem. Algorithmic complexity analysis [4] gives CPU consumption as an asymptotic function of the input parameters. In the real world, however, constants matter, and these

This research was supported by the National Science Foundation (NSF) under contract CCR-9901696, the Defense Advanced Projects Research Agency (DARPA) and the U.S. Navy (USN) under contract N660019928918, and the IBM Corporation. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the NSF, DARPA, USN, IBM, or the U.S. government.

constants vary from one hardware platform to another. We could attempt a more detailed analysis, based on processor spec sheets. With modern processors, this is virtually impossible: we would need to account for super-scalar execution, branch misprediction, TLB misses, and other complicating factors. Further, this will only give us CPU consumption, and not memory, network bandwidth, or battery energy consumption.

In our system, we use algorithmic complexity as a starting point: to provide hints that guide the learning algorithms that process history logs. This allows us to specialize a general asymptotic functional form to the specific hardware on which the application runs. We can also specialize our predictions to the specific input data on which the application operates, instead of always predicting a worst-case or average-case scenario.

By building and evaluating a logging and learning infrastructure, we hope to answer two questions:

- Is the overhead of logging and prediction acceptable?
- What is the accuracy of prediction?

Our approach is based on two assumptions. We expect the application programmer or domain expert to specify all the fidelity metrics on which application resource consumption depends. We believe this is a reasonable assumption: at the heart of most resource-hungry computations is a well-understood algorithmic core, with a small number of parameters that affect its resource consumption.

We also expect the resource consumption to vary smoothly with the fidelity. This is because we sample the fidelity space uniformly, and try to learn the fidelity-resource function from these samples. This assumes that the function is well-behaved between any two nearby samples. To relax this assumption, we would need more sophisticated learning techniques that increase the density of sampling wherever the local behavior of the function is anomalous or highly variable.

Our current prototype has three distinct phases:

- A *logging/training phase*, where we repeatedly run the application at various fidelities and log the results in a history log.
- A *learning phase*, where we feed the history log to offline learning algorithms. These algorithms use application-specific hints to convert the history log into *predictors* that compactly represent the mapping between fidelity and resource consumption.
- An *online phase*, where we run the application and use the fidelity-resource functions that we have learned, to guide adaptation. Odyssey uses the predictors generated by the learning phase to pick fidelities that will better match user latency requirements, desired battery life, and other resource constraints.

Ideally, we would combine all three phases, so that there is no need for a separate logging/training phase: the system learns as the user runs the application. However, it is difficult to explore the fidelity space completely without annoying the user. During actual use, we cannot simply choose fidelities to provide us with more history — the fidelity values must also match the user's latency, battery life, and other resource constraints.

Hence our current prototype requires some amount of logging and learning to be done offline. In fact, logging and learning also continue during the online phase. The offline learning provides the system with a good starting point, and the online learning modifies this starting point to track the dynamic behavior of the application. In this paper we only focus on the offline logging and learning mechanisms.

3. Design and Implementation

This section describes how we create a history log of application behavior, and how we use it to generate predictors of resource consumption. The history log is a collection of log entries: each entry associates a set of fidelity values with a set of resource consumption values. We feed these log entries to learning algorithms that learn the relationship between the fidelity metrics and the resource consumption.

3.1. Application-specific logging

We have implemented a single generic mechanism for logging application fidelity and resource consumption. However, each application has its own notion of what fidelity is, and how many dimensions it has. To bridge the gap between application-specific fidelity metrics and a generic logging mechanism, we use *application-specific configuration files* or ACFs. An ACF captures the salient features of an application with respect to resource consumption. Specifically, the ACF lists the *fidelity metrics* and *input parameters* for the application. An input parameter is a feature of the input data that affects the resource consumption — the size of the input data is frequently a useful input parameter. Both fidelity metrics and input parameters affect resource consumption — the difference is that we can adjust the fidelity, whereas we have no control over input parameters. The fidelity metrics and input parameters together form the input to a resource predictor function, whose output is the expected resource consumption.

Once we have generated a resource predictor using offline analysis, we encode this in the ACF as a *resource hint function*. During the online phase, Odyssey uses this hint as an initial guess, and updates it as fresh log entries are generated.

Resource	Units of consumption
Local CPU	millions of instructions executed
Energy	Joules
Latency	seconds
Network I/O	bytes transmitted/received
Remote CPU	millions of instructions executed
Physical memory	bytes
Disk I/O	bytes read/written

Figure 1. Resources consumed by multi-fidelity operations

3.1.1 Multi-fidelity operations

All resource consumption is measured with respect to a *multi-fidelity operation* [12]. A multi-fidelity operation, or just "operation" for brevity, is the unit of computation for which we can define fidelity metrics, input parameters, and resource constraints. It is an application-specific notion — for an interactive application, it is the computation done between a user request and the response. For a web browser, an operation is fetching and rendering a single page.

At the beginning of each operation, the application makes an Odyssey system call ($begin\_fidelity\_op$), and passes in the values of the input parameters. Odyssey computes and returns the appropriate fidelity values to use during the operation. This step uses the predictive ability of Odyssey to map fidelity values to the expected resource consumption.

When the operation completes, the application signals this to Odyssey by making another system call ($end\_fidelity\_op$). Odyssey then logs the fidelities, input parameters, and resource consumption of that operation. This data is also used to update the predictor functions and improve future predictions.

3.1.2 Data-specific logging

Sometimes, it is not possible to capture all the relevant features of an input data object — there may be effects that are too complex for us to express or even to understand. Hence we require the application to provide a unique label for the input data object, as an argument to $begin\_fidelity\_op$. This label could be the name of the file containing the input data. By logging this unique label along with the input parameters, we can make a more accurate, *data-specific* prediction when we see the same object again.

3.1.3 Resource monitors

The task of measuring resource consumption is done by a set of *resource monitors* in Odyssey. Each monitor is responsible for measuring a particular resource, and computing the amount consumed by each multi-fidelity operation.

Our current prototype monitors CPU and energy consumption. Figure 1 lists the complete set of resources that we envision supporting. To measure CPU consumption, we use the Linux */proc* file system, which reports the amount of CPU time consumed by each process. We scale the CPU time by the speed of the processor [1]. This scaling makes the measurement somewhat independent of the specific CPU on which we take the measurements, though of course we can never have a single number that exactly represents the CPU consumption across diverse processors. In this paper, all measurements were done on a single machine, and so we report CPU consumption directly in seconds.

To measure energy consumption, we use PowerScope [6]. PowerScope allows us to sample the power consumption of a laptop and attribute it to one of the many processes running on the machine. We extended PowerScope to include a timestamp with each sample. In post-processing, we use these timestamps to correlate power samples with the operations logged by Odyssey. We compute the total energy consumed during an operation, subtract out the known background power consumption, and attribute the remaining energy consumption to that operation.

Our current prototype maps each application to a single operation at a time: we do not yet support multiple concurrent operations by the same application. We also map each application to a unique set of processes. If there are multiple applications that use a shared service (such as the X server), we do not yet compute what fraction of its resource consumption should be attributed to each application.

3.1.4 Training mode

In order to acquire data about an application's behavior over the entire range of operating parameters, we run Odyssey in a special *training mode*. Normally, Odyssey would pick the fidelity for each operation to satisfy latency, battery life, or other constraints. In training mode, we disregard these constraints, and choose fidelities randomly in order to sample the entire fidelity space. By running the multi-fidelity operation many times, we acquire sample points all over the fidelity space. In order to explore the input parameter space as well, we conduct experiments with multiple input data objects.

3.2. Linear-fit predictors

For our initial prototype, we wished to build a prediction mechanism that was easy to understand, easy to implement, and computationally cheap. The simplest such predictor is a

[1] We use the "bogomips" value provided by Linux in */proc/cpuinfo*

linear one; given a set of n inputs and 1 output, we can run a linear regression on all our samples to predict the output as some linear combination of the inputs. Currently, choosing the inputs to the linear regression is left to the application programmer. For example, if the application programmer suspects that the CPU consumption of her algorithm is of the form $c_0 + c_1 pr\log(pr) + c_2 p^2 r^2$, where p and r are fidelities or input parameters, then she would specify the inputs $pr\log(pr)$ and p^2r^2.

The coefficients $c_0, c_1, \ldots$ computed during the learning phase are maintained as application-specific state during the online phase. Every time we wish to make a prediction, the application-specific predictor computes the function represented by these coefficients. Every time we get a new log entry, Odyssey updates the coefficients using incremental gradient descent [10]. Thus the system improves its prediction accuracy as more operations are performed, while keeping the computational expense of each update relatively small.

3.3. The solver

Once we have a prediction mechanism, we need to use it to make fidelity decisions. Given a predictor for CPU consumption and a CPU consumption constraint, we need to pick the values of fidelity for which we will satisfy the constraint while maximizing the fidelity. We have implemented — but not yet evaluated — a simple gradient-descent *solver* which does this optimization. If there are multiple fidelity metrics, then the solver maximizes an application-specific *utility function* that maps a multi-dimensional fidelity to a single number representing user satisfaction.

The predictors generated by offline learning are provided to the solver as *resource hint functions* in the ACF. The ACF also contains the utility function and an *update function*. The update function is called every time we log a new operation, and can update the internal state of the hint function. In our prototype these functions are implemented as entry points into an object file that is dynamically loaded into Odyssey when the application is started.

3.4. Applications

This section describes the applications that we have modified so far to use the Odyssey API extensions.

3.4.1 Radiosity

A *radiosity* [3] computation colors and shades a 3-D scene according to the light sources present in the scene. A 3-D scene or model is a collection of 3-D objects, each represented as a set of polygons which make up the surface of the object. Every time we edit the model, we need to run a radiosity computation in order to capture the lighting effects that we would see in the real world.

Radiosity and other 3-D graphics algorithms are key to building realistic augmented reality environments. Imagine an architect who is commissioned to renovate an old building, and wishes to show her proposed design to the client. With a mobile computer, a heads-up display, and augmented reality software, a client can walk around the building, and interactively view and edit the proposed renovations. To provide a realistic experience of this environment, we need sophisticated (and resource-hungry) algorithms such as radiosity.

Two of the most commonly used radiosity algorithms are *hierarchical* and *progressive* radiosity. Both of these are computationally quite expensive. The computational requirement grows with the number of polygons n in the input data — as $O(n\log n)$ for hierarchical and as $O(n^2)$ for progressive radiosity. Thus, it often makes sense to simplify the model before running the algorithm. This reduces the number of polygons in the model at the cost of losing some detail — we get a cheaper and quicker radiosity result at a lower fidelity. Thus, before running radiosity on any scene, we need to choose an *algorithm* — either progressive or hierarchical — and a *resolution* — a real number between 0 and 1, which specifies what fraction of the input polygons to retain. Figure 2 shows the ACF for the radiosity application.

Radiator is an implementation of several common radiosity algorithms with a GUI front end. It allows us to load a 3-D scene containing one or more 3-D objects and light sources, select a radiosity algorithm and a resolution, and run the algorithm. We have modified radiator to call Odyssey before each radiosity computation, passing in the number of polygons in the input data. Odyssey selects and returns the algorithm and resolution to be used for that computation.

3.4.2 Web browser

Our second application is a web browser that degrades the fidelity of GIF images fetched over the web by converting them to lossily compressed JPEG [13]. Previous research has shown that such degradation is effective in reducing the consumption of network bandwidth [7, 11] and energy [5]. In this paper we focus on energy: we predict the energy consumed to fetch an image over a wireless network and render it.

Web images have one feature or input parameter — the size of the original image — and one fidelity metric — the JPEG Quality Factor [2, 13], which represents the quality of the compressed image. The JPEG Quality Factor can take an integer value from 0–100; in our experiments we use only the range 5–80 since the compression algorithm

```
description radiator:radiosity  # <application>:<computation>
logfile /usr/odyssey/etc/radiator.radiosity.log
mode training                   # sample fidelity space
param polygons 0-infinity       # number of polygons in scene
fidelity resolution 0-1         # how much to scale down the scene complexity
fidelity algorithm progressive hierarchical # choice of algorithm
constraint lcpu 27721.8         # no more than 60 CPU seconds on a TP560X
hintfile /usr/odyssey/lib/rad_hints.so
hint lcpu rad_lcpu_hint         # hint function
utility rad_utility             # utility function
update rad_update               # update function
```

The ACF for the radiosity computation. The computation has one input parameter — the number of polygons in the input data — and two fidelity metrics — the choice of radiosity algorithm, and the resolution. The number of polygons and resolution are ordered and real-valued. The "algorithm" fidelity is unordered, and can take one of the two values "progressive" and "hierarchical". rad_lcpu_hint, $rad\_utility$, and rad_update are the names of entry points into the module "rad_hints.so".

Figure 2. The configuration file (ACF) for the radiosity application.

behaves unreliably outside this range.

Our web browser application is made up of an unmodified Netscape browser and an HTTP proxy running on the same machine. The proxy intercepts all web requests and transforms them into Odyssey system calls. Odyssey then fetches a degraded version of the image from a distilling proxy located on the other side of the wireless link.

4. Validation

To validate our prototype, there are two questions that we need to answer:

- Is the overhead of logging and prediction acceptable?
- What is the accuracy of prediction?

This section describes a set of experiments that answer these questions. All our experiments were run on an IBM ThinkPad 560X with a 233 MHz Mobile Pentium processor and 96 MB of RAM, running a Linux 2.2 kernel. The machine was equipped with a 2 Mbps, 2.4 GHz Lucent WaveLAN wireless interface.

4.1. Overhead of logging

In order to measure the overhead of logging application behavior, we measured the performance of a null operation — a call to *begin_fidelity_op* followed immediately by a call to *end_fidelity_op*.

The CPU overhead of Odyssey is 2.0 ms for each pair of calls. The increase in application latency is 2.2 ms per pair of calls. This overhead is higher than we would like, but we are confident that it can be substantially lower in a production version of Odyssey. Even a 2.2 ms overhead is often acceptable for an interactive operation such as the web image fetch — for a fetch that takes 1 sec, the increase in latency is only 0.22%. The 2.2 ms latency can be attributed to:

- 0.2 ms for logging, including the cost (averaged over many calls) of asynchronously flushing log entries to disk.
- 1.4 ms to measure application CPU consumption by reading and parsing files from */proc*. We could reduce this substantially by adding a more efficient interface to read CPU statistics from the Linux kernel.
- 0.5 ms for two calls to the Odyssey user-space implementation. A kernel implementation of Odyssey would have a much smaller overhead.
- 0.1 ms of other overhead.

4.2. Accuracy of prediction

4.2.1 CPU usage of radiosity

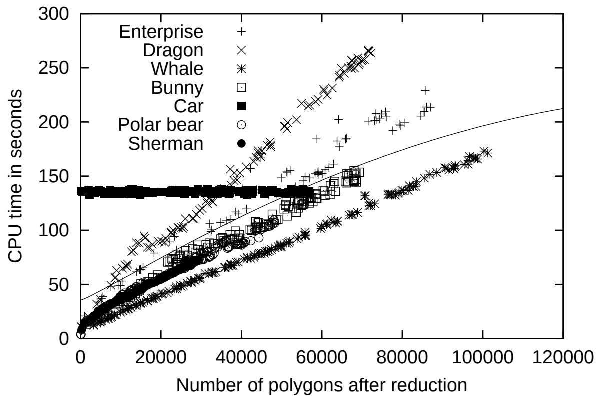
To measure the accuracy of history-based prediction of CPU consumption, we used the *radiosity* application (Section 3.4.1). We ran it on 7 different data objects, ranging roughly in size from 30,000 polygons to 200,000 polygons. We collected a total of 1578 samples in training mode. There were 3556 training runs; 1978 of them failed because they exceeded our resource limits. To prevent experiments that ran forever, we set a CPU limit of 300 sec on each radiosity computation. To avoid paging, which would distort our measurements, we limited the application to 64 MB of the available 96 MB of RAM.

Given our application-specific hints (Section 3.4.1), we ran linear regression with the inputs $pr \log pr$ and p^2r^2 —

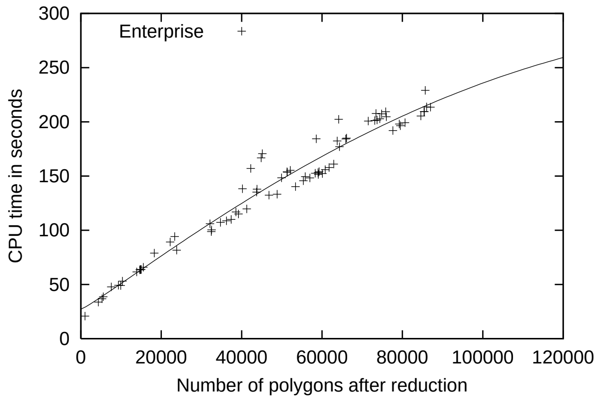
Data object	Polygons	Hierarchical radiosity			Progressive radiosity		
		Samples	% error	RMS error	Samples	% error	RMS error
Enterprise	203880	75	10%	10.7 sec	17	5%	0.3 sec
Dragon	108590	106	17%	6.1 sec	47	11%	0.5 sec
Whale	101814	159	4%	1.8 sec	68	5%	0.2 sec
Bunny	69543	153	11%	4.3 sec	87	6%	0.3 sec
Car	56972	164	1%	1.2 sec	0	n.a.	n.a.
Polar bear	48963	147	8%	3.1 sec	152	17%	0.7 sec
Sherman	29450	170	7%	1.4 sec	233	14%	0.4 sec
All objects		974	80%	36.7 sec	604	31%	1.8 sec

'RMS error" measures the absolute prediction error in CPU seconds. "% error" is the 90th percentile of relative error — a % error of 4% means that 90% of the samples had a relative error of less than 4%. When running progressive radiosity on the "car" object, all 295 training runs exceeded our CPU and/or memory limits and had to be discarded.

Figure 3. CPU prediction error for radiosity



(a) Fit on all objects



(b) Fit on the "Enterprise" object

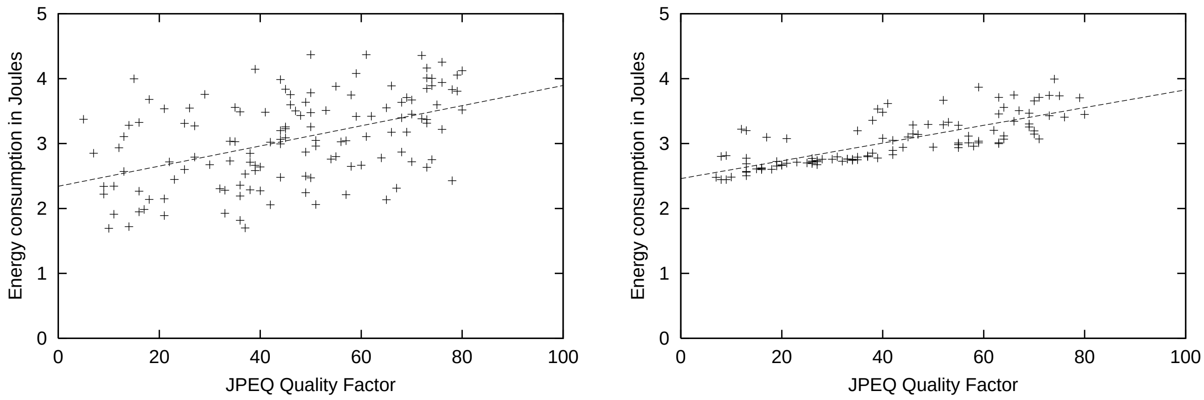
The x-axis is the reduced number of polygons (pr). The y-axis is the CPU consumption in seconds. The points represent measured samples; the line is a best fit on these points of the form $c_0 + c_1 pr \log pr + c_2 p^2 r^2$. The left-hand graph shows the fit when samples from all 7 data objects are combined; the right-hand graph shows the fit for the object "Enterprise" alone.

Figure 4. CPU prediction for hierarchical radiosity

Image	Size	Netscape			xv		
	(bytes)	Samples	% error	RMS error	Samples	% error	RMS error
nsh	1394081	118	10%	0.8 J	103	3%	0.8 J
apple	174650	124	22%	0.7 J	94	9%	0.3 J
radio	114816	130	25%	0.7 J	105	8%	0.3 J
castle	58223	130	38%	0.6 J	105	11%	0.2 J
circuit	19685	124	73%	0.6 J	107	8%	0.1 J
laserdt	8802	120	71%	0.6 J	104	10%	0.1 J
artban	971	130	93%	0.6 J	100	14%	0.1 J
redgem	110	127	94%	0.6 J	101	9%	0.1 J
All objects		1003	115%	1.1 J	819	63%	1.0 J

‘RMS error” measures the absolute prediction error in Joules. “% error” is the 90th percentile of relative error — a % error of 50% means that 90% of the samples had a relative error of less than 50%.

Figure 5. Energy prediction error for web image fetch



(a) Using Netscape **(b) Using xv**

The points are measured samples that correlate the JPEG quality factor f (x-axis) with the energy consumption in Joules (y-axis). The lines are best fits on these points of the form $E = c_0 + c_1 S + c_2 f S$. For this image, the size S is 58223 bytes.

Figure 6. Energy prediction for fetching image “castle”

for each algorithm, we computed a best fit of the form $c_0 + c_1 pr \log pr + c_2 p^2 r^2$. Figure 3 shows the prediction errors of this linear-fit approach. For individual objects, we find a good fit. On the other hand, when we try to fit a single function to all the data objects, we have a bad fit. Figures 4(a) and 4(b) are a visual illustration of the difference between data-specific and data-independent prediction. The curve does not fit the points very well in Figure 4(a), which includes all the data objects. Figure 4(b) shows the fit for the "Enterprise" object alone, which has the highest root-mean-square prediction error of any single object. Even in this worst case, we see that we have a good fit. This illustrates the importance of learning from recent history, and specifically of data-specific prediction from history.

4.2.2 Energy usage of the web browser

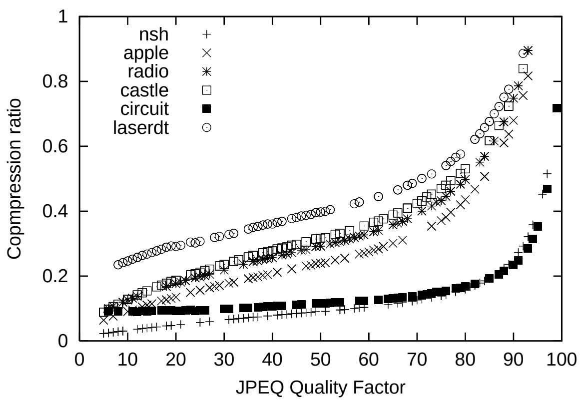
We measured the accuracy of predicting the energy consumption of fetching and rendering web images over a wireless network (Section 3.4.2). We performed 935 trial runs, each of which consisted of one operation — fetching and rendering a single image. The 7 images we used ranged in size from 110 bytes to 1.4 MB, and for each operation we picked a random fidelity (JPEG Quality Factor) in the range 5–80. The images were fetched over the WaveLAN wireless link from an IBM ThinkPad 570 with a 366 MHz Mobile Pentium II processor and 128 MB of RAM.

We measured the energy consumption of each operation by sampling the power consumption during the operation and subtracting out the background or baseline power consumption. This baseline is the power consumption when the CPU is idle, the screen is backlit, and the wireless interface is up but not in use. Our test machine had a baseline power consumption of 7.94 Watts.

We expect the energy cost of fetching an image to be proportional to its compressed size, and the cost of rendering it to be proportional to the uncompressed size. Thus we expect the energy consumption to be of the form $c_0 + c_1 S + c_2 S'$, where S is the uncompressed size and S' is the compressed size. If we further assume that the compression ratio $r = \frac{S'}{S}$ is linearly related to the JPEG quality factor f, then we get a function for energy of the form $c'_0 + c'_1 S + c'_2 f S$.

The first 5 columns of Figure 5 shows the prediction error of fitting such a functional form to the energy consumption of Netscape. We see extremely large prediction errors, especially for smaller objects. We found that this was caused by a large amount of noise in the energy consumption of the Netscape process. We suspect that scheduling effects in Netscape's threading package cause the amount of CPU consumption (and hence energy) to be non-deterministic.

How accurate would our predictions be if we had a well-behaved browser? To answer this question, we constructed a simple browser that sends HTTP requests, reads image data, and displays the image using *xv* [1], a freely available image editor program for X. We repeated the experiments using this browser instead of Netscape. The last three columns of Figure 5 show that we can actually predict energy consumption quite well (in the worst-case, our error is 14%). Figures 6(a) and 6(b) visually depict the difference between using Netscape and using xv, for the "castle" object.



For each image, the compression ratio r (y-axis) is a linear function of the JPEG quality factor f in the range 5–80. The graph does not show the images "artban" and "redgem": these images are so small that JPEG compression increases their size. In practice, we would never compress these images, but always use the original.

Figure 7. JPEG Compression ratio as a function of fidelity

Figure 7 shows us that the compression ratio for each data object — and hence the energy usage — is a linear function of fidelity in the range 5–80. This is why we have accurate per-object prediction: however, there is no single function that will fit all the data objects. Han et al. [8] have shown that the input byte size is not a good predictor of output bytes, but that that the output bytes seem to be a linear function of the number of input pixels. Even in this case, there is a lot of noise and prediction error across images: again, this illustrates the importance of data-specific prediction.

4.3. Overhead of learning

Since we currently do learning offline, the overhead of this phase is not critical. Our current implementation in Perl took less than 10 seconds (on a 233 MHz Pentium) to process 16 different data sets from the radiosity application. We expect that with an optimized C implementation, the overhead will be even lower.

5. Hybrid learning algorithm

The results in the preceding sections clearly indicate the value of data-specific prediction. However, we cannot anticipate every possible data object that an application might see. This suggests a hybrid learning approach. We use offline learning to learn a generic function that serves as a starting point. In the online phase, whenever we see a new data object, we adjust the coefficients to match the behavior of the new object. Thus at the cost of a few erroneous predictions during this calibration, we can accurately predict the resource consumption of the new data object.

We envision using this hybrid approach in the following way. When we have few samples for the input data object, we pick fidelities conservatively. In most cases, this will result in a "quick-and-dirty" version — the fidelity is lower than the user wants, and the resource consumption less than she was willing to spend. In such cases, the user simply repeats the computation. This time, we have acquired one more sample point, and can afford to be less conservative. By being conservative initially, we have acquired sample points cheaply, and improved our predictive capability at a small cost in resource consumption.

We have not yet evaluated this hybrid approach, but we expect the overhead of each update to be extremely low: our incremental gradient descent code does about 6 floating point operations per input on each update. Of course, there is also the memory overhead of keeping per-object state. If there is a large set of data objects, we might have to use caching mechanisms that discard information on long-unused objects, or save it to secondary storage. Alternatively, we could store the digested per-object information in the object itself, as an extension to the file format. An Odyssey application would be able to read this extension, and we would add a system call for the application to provide this information as a hint to Odyssey.

6. Related work

Adaptation and history-based prediction are well-known concepts; there are many examples of systems that use one or both techniques. To the best of our knowledge, this is the first piece of work that learns and predicts application resource consumption *as a function of fidelity* in order to improve adaptation in mobile applications. We see our predictive mechanism as a service to be used by higher-level adaptive systems.

We are aware of one other piece of work that tries to learn resource consumption functions: PUNCH [9] is a system for learning the CPU requirements of an application as a function of the input parameters. The objective of PUNCH is to use predictions of CPU usage to decide how and where to execute the application in a distributed computing environment.

The Odyssey predictor, on the other hand, predicts resource consumption as a function of both fidelity and input parameters. We use it in combination with the solver to pick the best possible values of fidelity for that computation. Odyssey is intended to be used with interactive applications in a mobile environment, where a "quick and dirty" answer is often more valuable to the user than a high-fidelity result that wastes time, battery energy, network bandwidth, and other resources.

7. Future Work

There are several directions in which we plan to extend this work. Our immediate task will be to expand the set of applications that use our API extensions. This should provide valuable experience with using the API and indicate how it can be extended or refined. We also intend to test our adaptive applications under realistic scenarios, and measure the benefit to the user of prediction-based adaptation. This would also allow us to evaluate the hybrid online learning mechanism described in Section 5.

We are working on expanding the number of resources supported by our prototype, and especially on adding latency (user wait time) and network I/O. User wait time is a critical resource for any interactive application, since it directly impacts user satisfaction. Network I/O is important since it affects energy consumption as well as latency. In fact we would expect energy consumption to be a function of CPU, network, and disk activity, because these affect the power consumption of the CPU, network interface, and disk respectively. Similarly, latency depends on CPU, network and disk consumption. We are designing a prediction mechanism that incorporates such "resource dependencies", where predictions for one resource (CPU) could be used by predictors for a higher-level resource (latency). We also intend to extend our system to allow multiple threads in an application, which could be performing multi-fidelity operations simultaneously.

In the medium and long term, we would like to extend our linear regression method to more sophisticated learning algorithms, and evaluate these algorithms — how accurate they are, how quickly they converge, how good the initial guess must be (for online methods), and what the overheads are. We would also like to find a safer way to specify application hint functions: our current approach of dynamically loaded objects is very efficient but not safe. We need a better mechanism (possibly an interpreted language) that would strike the right balance between flexibility, safety, and performance.

Our prototype relies on the application programmer to provide the utility function that maps fidelity to user satisfaction. This is very hard to do, especially with multiple fidelity metrics and time-varying user preferences. We intend to explore ways of using user feedback to update the utility function. This is analogous to the way that feedback on resource consumption updates our resource predictors.

Currently, the solver tries to find the best utility that satisfies a set of constraints. Often, we do not want to set a hard constraint on a resource such as latency — the user might be willing to wait a small amount of additional time, but only if it resulted in a large increase in fidelity. In other words, we want the highest fidelity that we can achieve cheaply. This corresponds to finding a knee, or "sweet spot" on the tradeoff curve between fidelity and resource consumption. We would like to characterize these "sweet spots" and have the solver find them automatically.

Acquiring history logs for each hardware platform that we might ever use is burdensome. We would like to use logs acquired on one hardware platform to make predictions on another. Our CPU measurement is already scaled to CPU performance; however, a simple linear scaling usually will not capture all the differences between processors. We will need a mechanism that uses log entries acquired on other platforms, but gives them a smaller weight than those acquired on the host platform.

Ideally, we would like the system to start with little or no log information and refine its predictions as it goes along. This requires techniques that can explore the fidelity space conservatively. For each operation, we need to pick a fidelity that is not too far from the known portion of the space, to avoid egregious mispredictions. At the same time we wish to extend the known space, so that we eventually learn about new desirable operating points. It would be interesting to investigate techniques that strike a balance between these two conflicting requirements.

8. Conclusion

Fidelity adaptation is essential for applications to maintain good interactive response and low battery drain in a turbulent and resource-poor mobile environment. However, for most applications, the exact effect of fidelity on resource consumption is not known a priori: it depends on the hardware platform and even on the input data to the application.

History-based prediction offers a way to measure, log, and learn the fidelity-resource tradeoffs of any application. This allows us to implement a variety of adaptation policies to pick good operating points on these tradeoff curves. Our initial results show that we can log and predict resource consumption with acceptable overhead and good accuracy. There remain a number of issues to be addressed in making history-based prediction easy to use and truly effective in guiding adaptation.

Acknowledgements

The authors thank Andrew Willmott for providing the source code to *radiator*, and for being the domain expert for this application. We thank Peter Dinda for many useful discussions on resource prediction. Feedback from David Petrou, Sanjay Rao, Mihai-Dan Budiu, Mor Harchol-Balter and Avrim Blum contributed substantially to improving this paper.

References

[1] J. Bradley. *xv: Interactive Image Display for the X Window System*. ftp://ftp.cis.upenn.edu/pub/xv/docs/xvdocs.pdf.

[2] S. Chandra and C. S. Ellis. JPEG compression metric as a quality aware image transcoding. In *2nd Symposium on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.

[3] M. F. Cohen and J. R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, Boston, MA, 1993.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Boston, MA, 1990.

[5] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile application. In *Seventeenth ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 48–63, Kiawah Island, SC, Dec. 1999.

[6] J. Flinn and M. Satyanarayanan. PowerScope: a tool for profiling the energy usage of mobile applications. In *Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10, New Orleans, LA, Feb. 1999.

[7] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 160–170, Cambridge, MA, Oct. 1996.

[8] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Peret, and J. Rubas. Dynamic adaptation in an image transcoding proxy for mobile web browsing. *IEEE Personal Communications Magazine*, 5(6):30–44, Dec. 1998.

[9] N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive application-performance modeling in a computational grid environment. In *Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC)*, Los Angeles, CA, Aug. 1999.

[10] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[11] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Sixteenth ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 276–287, Saint Malo, France, Oct. 1997.

[12] M. Satyanarayanan and D. Narayanan. Multi-fidelity algorithms for interactive mobile applications. In *Third International Workshop on Discrete Algorithms and Methods in Mobile Computing and Communications*, Seattle, WA, Aug. 1999.

[13] G. K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, Apr. 1991.