Thesis Proposal: Operating System Support for Mobile, Interactive, Multi-Fidelity Applications

Dushyanth Narayanan

December 2, 1999

Abstract

This proposal introduces and defines the notion of *multi-fidelity computation*, and makes the case that it is a viable approach to meeting the needs of interactive applications on mobile computers. These applications are constrained by the need for good interactive response, and efficient use of scarce resources such as user attention, battery power and wireless network.

To substantiate this claim, I plan to implement support for multi-fidelity computation in the Odyssey system. The proposal outlines a design for providing such support, and a plan for evaluating the support with a small number of applications.

1 Introduction

During the last decade, we have seen *mobile computing* come into its own as a field of study. Mobility introduces significant new challenges as well as opportunities that do not exist in the world of desktop computing. The primary challenge in mobile computing is the *scarcity* of resources. Mobile computers are resource-poor compared to static hosts of comparable cost; further, they are constrained by battery lifetimes, a consideration that is not relevant to static hosts with ready access to wall power. In some cases resource availability is also more *unpredictable* than in the static case: a wireless network environment is much more turbulent than a wired one.

As mobile machines become smaller, cheaper, and more wearable, it becomes clear that one of their main uses will be to interact with the user in order to provide them with some functionality; i.e., *interactive* applications are of great importance for this type of computer. Augmented reality, is, perhaps, the most compelling example of an interactive mobile application. Consider, for example, a technician working on aircraft maintenance. The technician wears a see-through head-mounted display, connected to a wearable computer. As he examines the aircraft, his view is augmented with useful information — components are labeled, potential faults in the structure are highlighted, and invisible detail (such as wiring) is made visible. One could imagine many such applications for augmented reality — in entertainment, on the shop-floor, or on the battlefield.

The combination of mobility and interactivity poses a new set of challenges. Mobility places us in a severely resource-constrained environment, where we must try and limit an application's usage of CPU, network bandwidth, and memory. Excessive use of these resources will affect an application's performance – i.e., increase the latency of our computations, and drain the battery at a higher rate. Further, in the case of some resources (such as wireless bandwidth or remote servers), resource usage might have an associated monetary cost. Thus, we need to adapt to scarce and highly variable resource levels to keep latency low, battery lifetimes high, and monetary expenditure low. Further this adaptation should be largely automatic and transparent: i.e., we need to keep *user distraction* at a minimum. This is especially important for a wearable computing environment, where the user may be engaged in real-world tasks in addition to interacting with mobile applications.

Interactive applications offer us one unique opportunity that other types of applications do not. In an interactive application, the final result of any computation is meant to be viewed by the user. In many cases, the user may be willing to accept a less-than-perfect result in exchange for lower latency or battery drain. If we could trade the *fidelity* of the result for resource consumption, we could dynamically find tradeoffs between fidelity, latency, battery lifetime and money, which a fixed strategy (a computation with an invariable fidelity) could not find. A computation that provides such a tradeoff between fidelity and resource consumption is a *multi-fidelity* computation.

Multi-fidelity computation is the central idea of this thesis: the following section describes more precisely the concepts of fidelity and multi-fidelity. We will see that "multi-fidelity computation" is a very broad abstraction that encompasses many algorithms and techniques developed by previous research. The rest of this document then describes how multi-fidelity computations can be used by interactive applications to conserve scarce resources on a mobile system.

2 What is a multi-fidelity computation?

We start by defining the the notion of a *multi-fidelity algorithm*, by analogy with the traditional definition of an algorithm. A multi-fidelity computation is then any computation that has a multi-fidelity algorithm at its core.

In the conventional view, an algorithm has a well-defined *output specification.* For example, any candidate for a sorting algorithm must preserve its input elements, but order them according to a given comparison function. Only algorithms that adhere to this output specification are considered acceptable, and we compare them to each other on the basis of CPU time consumed, memory used, or some other metric of resource usage or performance.

In many cases, however, a fixed output specification will not always match the user's needs. It might exceed the user's needs and thus waste resources, or fall short of them and generate unusable results. For example, suppose we wish to sort a list that the user intends to browse in sorted order. The most important thing is to produce the first few items quickly, so that the user sees a quick response (say, within 1 second). The remainder of the items can be sorted in the background, or not sorted at all — often, the user will stop after viewing the first few items, and will not request the rest of the list.

The key feature of the above scenario is that the *output specification* — the number of sorted items to compute — is not well defined. In fact, in this case we are more sure of our performance goal (we have determined, say, that most users wish to see a response within 1 second) than of the output specification. Additional constraints may be imposed by the operating system: since resources such as network and battery are shared among all applications, the system may place limits on the amount of these that a single computation can use.

In this example, the right thing to do is to present as many sorted items as we can compute within our resource constraints. We have inverted the roles of resource consumption and output specification — rather than fix the output specification and aim for the lowest resource consumption possible, we bound the resource consumption and produce the best possible output. Of course, in the general case, we might not want to fix either of these to a particular value; rather we might specify an allowable range, and let the system find the best possible tradeoffs.

Since the traditional notion of algorithm does not allow us to perform this inversion, we define a *multi-fidelity algorithm*: it is a sequence of computing steps that (in finite time) yields a result that falls within a range of acceptable output specifications, called fidelities. Upon termination, a multi-fidelity algorithm indicates the fidelity of the result.

Thus, the key feature of multi-fidelity algorithms is that they allow a range of different outputs, given the same input. The outputs are not all equivalent, however: each has a different fidelity. The essence of a multi-fidelity algorithm is that we can trade this fidelity for resource consumption — the resource consumption in its turn affects the latency, battery drain, and monetary cost of the computation.

3 Thesis statement

Multi-fidelity computation is an abstraction that captures a wide range of algorithms and techniques to trade output specification for resource consumption. Further, it is feasible to design and implement operating

system support for this, and to improve the usability of mobile, interactive applications with such system support.

I plan to validate this thesis in the following steps:

1. Define and elaborate the concepts of fidelity, multi-fidelity computation, and resource.
2. Create a programming model for application programs to use multi-fidelity computations, and a system API that fits into this model.
3. Design and implement a system that implements this API.
4. Demonstrate a small number of mobile interactive multi-fidelity applications as a proof-of-concept.
5. Evaluate the result from different aspects — how well the system implements the API; how easily the programming model allows multi-fidelity applications to be written; how much benefit each application actually derives from using multi-fidelity computations.
6. Sanity-check the usability of the multi-fidelity applications through informal user studies.

As we shall see, the class of multi-fidelity algorithms is quite large and varied. Many previously existing types of algorithms (see Section 8) can be considered specific instances of multi-fidelity algorithms. The challenge, and expected contribution of this work, is to come up with a characterization of multi-fidelity that is broad enough to encompass these various subclasses, yet concrete enough to allow effective system support for real applications.

There are several problems that I need to solve to reach this goal:

1. I need a clear definition of fidelity, to answer questions such as: how do we measure fidelity? Is it a property of a computation, or of data? Is it single- or multi-dimensional?
2. I need to develop a programming model and API that reflects this definition of fidelity. I need to identify real, interactive, mobile applications and apply this programming model to them.
3. I need to implement a system that will make fidelity decisions for applications, with a view to conserving system resources and maximizing the user's happiness or utility. In order to do this, I need to have a clear definition of what we mean by resource, and how resource consumption affects utility. The system also will need to monitor and manage the resources that I have identified as being of interest.
4. The core idea of multi-fidelity algorithms is to add fidelity as a degree of freedom, so we can find better tradeoffs by varying the fidelity. A "good" tradeoff is one that (conceptually) maximizes some notion of user happiness or utility. Thus, I need to know how utility is related to fidelity and to resource consumption, and to learn these relationships automatically by monitoring the application and the user. In the latter case, I need methods to get information from the user, while keeping user distraction to a minimum.
5. If we do not have a precisely known utility function, then we might be able to guess it. For example, if there is a sharp knee in some tradeoff curve (say between latency and fidelity), it is likely to be a point of maximum (or minimum) utility. I need techniques to automatically find these knees, or sweet spots.

The remainder of this document is organized as follows. Section 4 describes the portion of this work that as been completed, or largely completed — a conceptual model for thinking about fidelity and resources, and a programming model for multi-fidelity applications. Section 5 describes three proof-of-concept applications that I intend to develop, and gives a flavor for the application domain — mobile, interactive applications — that motivates this work. Section 6 is my plan for the remainder of the thesis work: it describes my approach towards the problems mentioned here — learning fidelity/resource tradeoffs, finding sweet spots on the tradeoff curves, and selecting proof-of-concept applications. Section 7 is an itemized list of work items for the proposed thesis work. Section 8 describes related work. Section 9 outlines the expected contributions of this work to the field. Section 10 provides a time-line for the thesis work. Finally, Appendix A lists the current version of the multi-fidelity API.

4 Completed work

This section describes the work already completed. Sections 4.1 through 4.5 introduce a broad conceptual framework for thinking about multi-fidelity computations, and multi-fidelity system support. The intent is to provide a basis for a system design that will support an extremely general notion of multi-fidelity computation.

Section 4.6 then describes the programming model and API that I actually intend to implement. This API has various assumptions and restrictions that map out a subspace of the general framework. The intent is that, while the implementation might support only a restricted version of multi-fidelity system support, the system design should allow any of these restrictions to be easily relaxed, and the implementation to be extended to support a more general version of multi-fidelity.

4.1 Fidelity and quality

What is fidelity? Intuitively, we can define fidelity as some measure of quality, or goodness. Odyssey [20] defines the fidelity of a data object as the extent to which it matches the perfect, or reference, object. Similarly, we can define the fidelity of a computation result: it is the extent to which the result approximates the best possible, or highest-fidelity, result. In general, there may be more than one dimension of fidelity: conceptually, the reference fidelity is the one that has the maximum possible value along each dimension.

It would seem, then, that the fidelity of a computation is simply the fidelity of its result, which is a data object. However, for most multi-fidelity computations, we can not directly specify the fidelity of the result — that will be known only after the result is computed. For example, in the case of JPEG compression, we could measure the fidelity of the compressed image by comparing it pixel-by-pixel to the uncompressed version. We control the JPEG algorithm, however, by providing it a desired compression ratio: this parameter influences, but does not exactly predict, the fidelity of the result. Some images compress better than others, and so can generate a higher fidelity result with the same input parameters.

Thus a multi-fidelity algorithm can be seen as providing a set of *knobs*, or *fidelity parameters*, as well as a set of *dials* which indicate the *fidelity* of the final result. The settings on the knobs will determine the readout on the dials; however, the knobs and dials might not correspond in a one-to-one fashion. Thus, one way to look at a multi-fidelity computations is as a mapping from the space of fidelity parameters to the space of fidelities. For each execution of the computation, we can pick one specific point in the parameter space, and get a result of the corresponding fidelity.

A fidelity parameter could either be a *knob* — a real-valued (discrete or continuous) variable — or a *switch*, an enumerated type of variable that chooses between an unordered set of options. A typical switch might choose between different algorithms that implement the same functionality. Consider *sorting* as a simple (and contrived) example. Suppose the user wanted to sort a list of items in order to examine them in sorted order. Further, the user will usually only examine the first few items. When the CPU resource is scarce, in order to have low latency, we compute a small number of sorted items with SelectionSort. If we have sufficient CPU, we could choose a larger value of m — in this case QuickSort is obviously the more efficient option. Thus we have one fidelity parameter (a "switch") to decide which algorithm to use, and another (a "knob") that decides what fraction of the list to sort. In this case, the setting on the knob determines our final fidelity; the setting on the switch affects the latency.

As we just saw, the "goodness" of such an execution is not determined solely by the fidelity of the result. The user also cares about other aspects of the execution — for example, the traditional QoS metrics of latency, throughput, and jitter. For interactive applications, the former is the key metric — throughput-based measures are often irrelevant and/or deceptive [5]. In a mobile interactive environment, we need to add three non-traditional metrics to this list: *battery lifetime*, *user distraction*, and *monetary expenditure*. These metrics, together with all the fidelity dimensions, make up a *quality* vector, which captures all the aspects of a multi-fidelity execution that a user cares about.

4.2 Quality and Utility

We can now view a multi-fidelity computation as providing a mapping from a set of fidelity parameters to a quality vector. This mapping depends on the nature of the computation itself, as well as the system state at the time of execution. Thus, the object of multi-fidelity system support should be always to set the fidelity parameters to the values that result in the "best possible" quality vector.

Since our quality vector is multi-dimensional, we need a *utility function* that maps the quality vector to a single measure of goodness, or utility. Now, the job of the multi-fidelity system is to always pick the fidelity parameters to maximize the utility.

If there are multiple, concurrent applications, then each will have its own set of fidelity parameters. We can combine these to come up with one global parameter vector, which maps onto a global quality vector, which in turn maps onto the global utility.

Thus the most general description of a multi-fidelity system is that it knows (or somehow learns) the mapping from all the fidelity parameters to the global utility, and always selects values for these parameters that maximize the utility.

Note that battery lifetime, user distraction, or monetary loss are actually global quality metrics: the user does not care about the amount of battery drained by this or that application, but about the net effect on the battery lifetime. Thus, our global quality vector needs only one dimension for each of battery drain, user distraction, and monetary loss.

4.3 Resources and system state

We have already seen that the fidelity of an execution is determined by the values of the fidelity parameters. However, the remaining quality metrics are not determined by the fidelity parameters alone: they also depend on the execution environment — i.e., the input data and the system state.

My approach is to split up this dependency into two steps. Given a set of fidelity parameters and the input data, we can determine the *resource consumption* of the computation. In general, there may be an arbitrary relationship between a computation's input data and its resource consumption. However, the *size* of the input data is the single feature that has the greatest impact on the resource consumption; moreover, it is a feature that is easily and cheaply measured[1]. I plan to summarize the effect of the input data by a single feature: its size. Thus the *resource consumption function* of any computation provides a mapping from the fidelity parameters and the input data size to the resource consumption.

The resource consumption of a computation, and the system state (the *resource availability*) together determine all the quality metrics other than fidelity: latency, battery lifetime, etc. Battery drain (which is the reduction in battery lifetime), for example, depends on the number of bytes transferred and received over the network, the number of CPU cycles used, etc[2].

Table 1 describes the various resources consumed by a multi-fidelity computation, and Table 2 lists the dimensions of resource availability.

The resource availability for a particular application depends on the global resource availability, and also on the *allocation* to that application of shared resources such as CPU, memory, and network bandwidth. Thus, in reality,

[1]The field of algorithmic complexity can, in fact, be viewed as the study of resource consumption as a function of input data size.

[2]We could have classified battery power as a dimension of resource consumption; in this case, the battery drain would be a trivial function of the battery power consumption. However, in my view, battery drain is not a function of the computation alone — it depends on system parameters such as the power used by transmission/reception on the wireless interface(s).

Resource consumed	Unit
CPU	MIPS-sec (million instructions)
Network xmit	bytes
Network rcv	bytes
Memory	bytes
Disk space	bytes
Disk xfer	bytes
Cache state	*set of objects accessed*

Table 1: Resources consumed by a multi-fidelity computation

Resource	Unit
CPU	MIPS
Memory	bytes
Network xmit b/w	bytes/sec
Network rcv b/w	bytes/sec
Disk b/w	bytes/sec
Disk space	bytes
Energy cost of CPU	joules/MIPS-sec
Energy cost of n/w xmit	joules/byte
Energy cost of n/w rcv	joules/byte
Energy cost of disk xfer	joules/byte
Monetary cost of n/w xfer	$/byte
Cache state	*set of cached objects*

Table 2: Resources available on a mobile computer.

the system has an additional set of knobs to tweak: in addition to setting fidelity parameters for each application, it can adjust the resource allocations to maximize the global utility.

4.4 Cache state as a resource

The list of resources described in Tables 1 and 2 corresponds fairly well to the traditional set of system resources, except for the addition of "cache state". In a mobile environment, where wireless connectivity is intermittent and unreliable, caching of data is a crucially important strategy. The latency and battery consumption of a computation depend on whether the objects it accesses need to be fetched over the wireless network. Just as a request to transmit some data can be viewed as making a demand on the network bandwidth resource, an object access can be viewed as a "demand" on the "cache state resource". From this point of view, it makes sense to consider cache state as a resource, even though it does not have many of the properties of traditional resources: it is not consumed, and cannot be "allocated", in the normal way.

The cache state resource is related to, but different from, the *disk space* resource. The former represents the set of cached objects; the latter the amount of space available to store these objects. In the same amount of cache space, we could store different sets of objects (or the same objects with different degrees of staleness); each of these would correspond to a different "value" of the cache state resource availability.

4.5 User distraction as a quality metric

One of the main drawbacks of utility-function based models is that, in practice, it is very hard to know these utility functions *a priori*. For example, if the user wants to do a document search, would they prefer a faster search, a more accurate search, or a more comprehensive (i.e. over a larger set of documents) search? The answer depends on the user, and even on the particular task that the user might be engaged in.

The application programmer could provide some indication of the utility function; however it seems clear that ultimately we need to learn this function from this user. In the extreme case, we could treat the user as an "oracle", and directly query them every time we needed to evaluate the utility function. Obviously this is not practical; while we may in theory make better tradeoffs, we have detracted from the user's experience by distracting them. Thus, it is useful to consider user distraction explicitly as a quality metric — this means that asking the user to make all the tradeoff decisions does not always produce the best tradeoff, since our quality suffers along the dimension of user distraction.

4.6 Programming model

What is the best way for multi-fidelity applications and the system to communicate with each other? How best can I implement the abstract model described in the preceding sections to meet the specific needs of mobile interactive multi-fidelity applications, at the same time ensuring that the system is easily extensible to support a broader range of applications? Attempting to answer these questions leads to a programming model, and an API, for multi-fidelity, which I describe here. This API should be considered a preliminary version: I expect that, as I implement various applications, the API will evolve to meet their needs. The API is described in this section, and listed in Appendix A.

One of the main features of a good programming model is that it minimizes the programming effort required to write a multi-fidelity application, or to modify an existing application to incorporate multi-fidelity computations. In my model, the application's primary responsibility is to identify the multi-fidelity computation, and to define the fidelity parameters, the fidelity dimensions, and the mapping from the former to the latter. This information is described to the system by a single `register_fidelity` call made by the application during initialization.

In the actual implementation, I plan to support a restricted version of this API, where the fidelity parameters (the knobs) are constrained to be identical with the fidelity dimensions (the dials). The programmer can also specify fidelity "switches", which can take on one of a set of discrete values. The switches are assumed to be orthogonal to the fidelity; they only affect the resource consumption. In practice, I expect that the switches will be used to choose between different algorithms that perform the same task (such as SelectionSort and QuickSort): since they have different fidelity/resource tradeoffs, the system can dynamically pick whichever one best satisfies its goals.

My claim is that this restricted API reduces the burden on the application programmer (and on the system implementor), without sacrificing too much generality. In fact, in many cases, the application's programmer's best guess for the fidelity of a result is just the value of the fidelity parameters used to create it. For example, in Odyssey, the data fidelity of a JPEG compressed image is measured by the lossiness parameter provided to the compression algorithm.

At the beginning of each multi-fidelity computation, the application queries the system with a `begin_fidelity_op` call, which signals the start of a computation and provides the input data size as a parameter. The system responds with the fidelity parameter values that it has determined are most appropriate. At the end of the computation, the application makes an `end_fidelity_op` call, which informs the system about the fidelity of the result. By monitoring the application between these two calls, the system can measure the resource consumption of the multi-fidelity computation. By doing this repeatedly, the system can build up a set of data points from which it can learn the resource consumption function of the computation. (It could also try to learn the relationship between the fidelity parameter vector and the fidelity vector; in my implementation, however, I will assume that the two are identical).

In learning the resource consumption function of a computation, we would like to use any knowledge of this function that the application programmer has. For example, the programmer might know that the CPU consumption of some particular computation varies quadratically with one of the fidelity parameters. Alternatively, this function might be based on empirical data, by doing a series of controlled user tests. We allow the programmer to express such knowledge by providing the system with *hints*. A hint is a function that maps the fidelity parameters and input data size to the consumption of some resource The system will then use these hints as a starting point in learning the resource consumption function.

The system also needs to learn the utility function that maps application quality metrics to "user happiness". The best way to learn these functions is to query the user; however, this approach would suffer along the quality metric of user distraction. Again, we would like to make use of the application programmer's knowledge in the form of hints. Realistically, I expect that application programmers will be able to hint a restricted class of utility functions, by registering a *quality constraint* on the most important dimension of quality. For example, the programmer of a augmented reality application might decide that it was crucial for all rendering operations to take 200ms or less; this would correspond to a utility function that had very high utility for all latency values below 200 ms, and very low utility for all latencies above 200 ms.

Section 2 described multi-fidelity computation as an inversion of the traditional role of output specification and resource consumption, which fixes the output specification and allows resource consumption to vary. In other words, traditional algorithms fixed the fidelity parameters and allowed the other quality metrics (which depend on resource consumption) to vary. By allowing constraints on other quality dimenisons, we are in fact implementing exactly this inversion — we fix some other dimension of quality and allow the fidelity to vary.

The `hint_constraint` call provides a way for application programmers to specify thresholds (either maximum or minimum, depending on the metric) on any quality metric: the system will then try to vary the other quality metrics in order to satisfy this constraint. A variant of this call will be made directly visible to the user, to allow them to set global constraints — minimum acceptable battery lifetime, maximum acceptable dollar expenditure.

Finally, to arbitrate between multiple applications, we need to know the global utility function, which maps the combined quality metrics of all applications to the utility. In theory, this could be any arbitrary function of the quality metrics. In practice, I will assume that the global utility function can be computed in two steps: each application has a utility that is a function of its quality metrics alone, and the global utility is a function (probably a weighted average) of the application utilities.

4.6.1 Incremental fidelity metrics

Consider a computation that continually improves the fidelity of its result, and can be interrupted at any time to return the best result computed so far. We call this an incremental-fidelity computation, or an anytime algorithm [4]. One could fit such a computation into the standard multi-fidelity model — we supply it with fidelity parameters that specify a target value for each fidelity dimension, and the algorithm stops when the current fidelity meets all the targets.

This naive approach ignores the special property of incremental-fidelity operations: that they can be interrupted at any time. Thus, in theory, we could watch the utility of the computation over time, and interrupt it when we think we have reached a maximum. This requires the system continuously monitor the fidelity of the computation, probably by having the application make periodic calls to the system.

I think this approach places an unnecessary burden on the application programmer. I propose to implement a simpler addition to the programming model — that of *constraint callbacks*. We have already discussed how the application programmer can register simple utility functions in the form of quality constraints. In the case of quality metrics monitored by the system (latency, battery drain, etc.), the application can additionally register a callback, to be invoked if the computation exceeds this constraint. Constraints on fidelity metrics can, of course, be checked by the application itself.

While this does not exploit all the power of incremental-fidelity algorithms, it is a reasonable compromise. In effect, I expect that it will provide a "sanity-check" on any fidelity decisions made by the system — if we happen to make a bad choice of fidelity parameters, we can still recover by interrupting the computation before the latency, battery drain, user distraction, or monetary expenditure become unreasonably high. Further, the "abort" decision on such computations will usually be made by the user; thus, the aim should be to present the user with a (continually improving) indicator of fidelity, and a way for them to stop the computation whenever they are satisfied.

5 Motivating scenarios

The ultimate aim of multi-fidelity system support is to enable real applications to run more effectively. This section presents three scenarios from my chosen application domain — mobile, interactive applications — that illustrate how they could use multi-fidelity.

5.1 Rendering for Augmented Reality

An architect is designing the renovation of an old warehouse for use as a museum. Using a wearable computer with a head-mounted display, she walks through the warehouse trying out many design alternatives pertaining to placement of doors and windows, placement of interior walls, and so on. For each alternative, the augmented reality software on her wearable computer superimposes the proposed design change on the architect's view of the surroundings. In many cases, an aesthetic or functional limitation immediately becomes apparent and she rejects that alternative. In the few cases that survive this stage of scrutiny, the architect requests a more accurate visualization as well as standardized tests to ensure that the design is structurally sound and meets building codes.

In this application, rendering of 3-D objects is performed frequently. Before an object can be rendered, it needs to be appropriately colored or shaded, according to the light sources present, and the shadowing and reflected light from other objects. A well-known way to do this shading is with a *radiosity computation* [3]. This is highly compute-intensive, but our application requires low latency for good interactive response. Our architect wishes to do "quick-and-dirty" validations of her ideas, and is willing to sacrifice some fidelity in order to be able to try out many ideas interactively.

There are many ways in which the application can control the fidelity of rendering. It can choose between different algorithms such as progressive radiosity and hierarchical radiosity. It can also control the number of polygons used to represent 3-D objects. A representation with fewer polygons can be shaded faster, but will have a lower fidelity compared to the original.

Using a multi-fidelity approach, the application collaborates with Odyssey to choose between these alternatives. Odyssey's guidance is based on the current CPU availability, the latency constraints of the application, and knowledge about the CPU consumption of the application at various fidelities. If the data for rendering the 3-D objects reside on a remote server, then Odyssey must also decide on how much computation to perform at the server and how much at the client, based on CPU and network availability. The current cache state of the wearable computer and the residual energy in its battery are likely to be important influences on this decision.

5.2 Scientific Visualization

A visualization program presents earthquake simulation data as an animation. The transformation is highly compute intensive and proceeds in three pipelined stages: sampling the data onto a regular grid, computing isosurfaces, and rendering a 2-D view of the 3-D grid. The user specifies a region of interest, and a desired frame rate. The application then queries the system to find the appropriate configuration and fidelity level. As the animation continues, conditions in the system change; whenever the application needs to adapt to the new state of the system, it receives a callback and is supplied with new fidelity parameters.

In this application, fidelity can be reduced by downsampling the input grid of data. In a distributed implementation of the application, the split of functionality between client and server is important. The optimal split depends on the current availability of client and server CPU as well as network bandwidth. Thus the system has to decide what the best split is, and what fidelity is sustainable while maintaining the desired frame rate.

Although scientific visualization is not closely related to mobile computing, it is an important application domain that can benefit from a multi-fidelity approach. Through visualization, large data sets such as MRI scans and astronomical measurements can be grasped more effectively by users. The Quake visualizer [1] is an instance of this kind of application.

5.3 On-Site Engineering Calculations

An unexpected contingency has arisen at a bridge construction site: excavation for a pier has revealed a different soil type than planned for in the design. A civil engineer is at the site, exploring design modifications to the bridge to cope with this problem. To assist him in his work, he uses a hand-held computer running a spreadsheet-like software package. He explores a number of alternatives, examining the impact of each on the strength of the bridge, on the manpower and material costs, and on the delays it will cause to the schedule. During the initial part of his exploration, speed is more important than accuracy of results — results are therefore displayed to him in a font and color indicating low fidelity. As he converges on a few promising alternatives, he requests higher fidelity results: the computations now take much longer, and the presentation of the results indicates the higher fidelity. Once he has selected what appears to be the best choice, this design modification is shipped over a wireless link to his company's supercomputer for full structural analysis and supervisory approval. Once he has received this approval, the engineer gives the modified design to the local personnel. Work can proceed without further delay.

In this example, the multi-fidelity algorithms are all numerical in nature. Depending on the specific problem, there are many ways in which different levels of fidelity can be obtained. For example, a successive approximation algorithm may terminate after just a few iterations to yield a low-fidelity result. A simulated annealing algorithm may change the coarseness of its mesh as well as the number of iterations to yield different fidelities. The number of terms in a series expansion can be varied to achieve a desired fidelity. Indeed, the concept of fidelity is probably most easily applied to numerical algorithms.

6 Proposed work

Section 4 described the work already done in designing system support for multi-fidelity computations — a conceptual framework for thinking about multi-fidelity computations, and an initial draft for a multi-fidelity API. This section provides a road map for the remainder of the thesis work. Section 6.1 describes some general design principles that I intend to follow. Sections 6.2 through 6.14 describe the remaining design and implementation tasks. Section 6.15 describes my plan for evaluating the system. A unified list of work items is presented in Section 7.

6.1 Design philosophy

My intention is to integrate system support for multi-fidelity into Odyssey [20]. Odyssey currently provides the ability for applications to adapt to changes in network bandwidth. The nature of the adaptation is left up to the application; typically it consists of changing the *data fidelity* of objects transferred over the wireless network.

I propose to extend Odyssey to support the multi-fidelity API, and also to monitor the full set of resources described in Section 4.3. The current Odyssey prototype monitors wireless network bandwidth, and is being extended to monitor battery power as well [8]. In adding functionality to Odyssey, I plan to leverage as much of the existing Odyssey code as possible, and ensure that "legacy" Odyssey applications can run concurrently with new multi-fidelity applications.

The ultimate aim of any system support is to enable *real* applications to function more effectively. There are three broad approaches to creating such application programs: run unmodified applications on top of (transparent) system support, run modified applications, or write the applications from scratch.

The essence of multi-fidelity is to *alter* the behavior of the application, and even the results presented to the user, in order to make more efficient use of resources. Thus, the first approach — running unmodified applications — is not a good fit. This is in line with the Odyssey philosophy, which emphasizes *application-aware* as opposed to *application-transparent* adaptation.

The approach of writing applications from scratch is made unattractive by the amount of time and effort required to build a realistic application. Further, the notion of multi- fidelity is applicable to a very wide range of applications; requiring applications to be written from scratch would greatly reduce the number of applications that could benefit from multi-fidelity support.

In this thesis, I plan to take the second approach — to modify existing applications to use multi-fidelity support. Thus, one of my goals while creating the programming model described in Section 4.6 was that it be easy to modify an existing application's source to use this method.

One might modify an application in various ways — in this thesis, I have chosen the obvious and most simple way: to modify the source code. In principle, other modification techniques — code interposition, binary rewriting, etc. — could be used to modify an application for which source code is unavailable.

6.2 Resource monitoring

In order to make good fidelity decisions, the system needs to be continuously aware of the availability of resources: CPU, network bandwidth, memory, battery power, etc. I plan to extend the existing Odyssey monitoring of network bandwidth to include CPU (both local and on a remote computation server), memory and battery power as well. The real aim of resource monitoring is *resource prediction*: knowing how much resource will be available to an application in the near future. I plan to develop simple methods for resource prediction.

Section 4.3 describes a broader view of resource availability (or system state), than I have just described here. In this view, resource availability includes things such as the energy cost of CPU/network/disk access, or the dollar cost of network access. I plan to develop simple techniques to measure these aspects of resource availability, as well.

6.3 Using cache state

In Section 4.4, we talked about treating cache state as a resource. Currently Odyssey supports caching on a per-warden basis: a *warden* is a module that is responsible for a particular type of data (images, video streams, etc) and is responsible for all operations (including caching) on objects of this type. I plan to extend this mechanism, to allow Odyssey to cache untyped data objects, and for applications to access these objects through Odyssey. Since Odyssey manages this cache, it will be aware of the cache state at all times. Suppose we have a multi-fidelity algorithm (like a search) whose fidelity depends on the set of objects actually searched. If we knew the relationship between fidelity and the set of accessed objects, we could decide which uncached objects were worth fetching over the network, and which cached objects we can evict.

I plan to extend the multi-fidelity API so that applications can express to the system the set of data objects accessed, and the fidelity tradeoffs involved in adding or removing objects from this set.

6.4 Getting user feedback on utility functions

I need to develop ways of getting user feedback that help us make better tradeoffs, without being extremely intrusive. My current plan is to augment the programming model, so that each application can add a user-interface component that receives the user feedback. For example, this UI component might have buttons for each quality metric: by clicking on a button, the user would indicate that they wished the application to go faster, or have a higher resolution, or that they wanted a longer battery lifetime. Alternatively, these buttons could be replaced by slider bars that the user could drag to a desired value. There could be a range of user interfaces, of which the system would pick the one that satisfied our “user distraction” requirements.

6.5 User distraction as a quality metric

Though I have identified user distraction as a key metric of quality, I still do not know how it can be measured, how applications might place constraints on it (as they would for other quality metrics), and how we could predict it. A major part of the conceptual work remaining in this thesis is to come up with a model of user distraction that allows us to incorporate it as a first-class quality metric.

6.6 Learning to predict quality

Section 4.3 describes how we split up the mapping of fidelity parameters to quality into three steps: mapping fidelity parameters to fidelity; mapping fidelity parameters and input size to resource consumption; and mapping resource consumption and system state to the quality metrics other than fidelity.

My implementation will assume that the first mapping is a trivial one; that the fidelity parameters correspond exactly to the resulting fidelity. The second mapping introduces a substantial learning problem; each resource might be an arbitrary function of the fidelity parameters. I plan to investigate simple online learning techniques to find these mappings. The object is not to do research into sophisticated learning techniques, but to demonstrate that a machine learning approach is a reasonable one. I shall probably rely heavily on application hints about resource consumption (Section 4.6); for my proof-of-concept applications I shall generate these hints by running an offline learning algorithm on a controlled set of executions.

The third mapping (resource consumption to quality) I plan to implement using simple functions — for example, given the energy cost of CPU, network, and disk access, we simply multiply by the amount of each resource consumed and add, to get the reduction in remaining battery lifetime. Similarly, monetary cost and latency are linear functions of those resource usages that cost money and take time, respectively. In reality, different applications may have different levels of concurrency, and so the latency may not be exactly the sum of CPU-time, network-time, etc. However, I expect that we can still learn latency as a linear function of CPU, network and disk access; the coefficients may vary slightly from one application to another. My planned implementation will only support these four quality metrics: latency, battery lifetime, money, and user distraction.

One resource that it is hard to reason about is *physical memory*. As long as there is sufficient physical memory for all applications, memory usage has little effect on the latency or battery usage of a computation. However, as soon as there is memory contention, we incur the costs of paging. These costs are hard to measure and predict, and even harder to charge to a specific application. In this work, I make the simplifying assumption that it is always a bad idea to have memory contention. In effect, I add an implicit constraint that total memory usage not exceed memory availability — in other words, we assume that our utility will become extremely low if we start paging.

6.7 Learning application utility functions

My approach to learning application utility functions will be based primarily on constraints (Section 4.6) provided by the application programmer. Constraints allow us to rephrase the goal "maximize the utility function" as "find the best fidelity that does not violate our constraints". The main drawback of application-specified utility functions is that they do not account for different users having different preferences, or for a single user's preferences changing over time.

I plan to find simple methods to get user feedback on application utility functions. The primary goal of these methods will be to extract information with minimal distraction of the user. Ideally such a technique would be completely non-intrusive, and passively monitor the user's behavior. Realistically, I expect that I shall have to actively solicit some user feedback; for example, each application might have an associated button for each quality metric — the user could click to reduce the latency, or increase the resolution, etc. I plan to use similar methods to get user constraints on "global" quality metrics: the user sees slider bars to set the desired battery lifetime, and the budget for monetary expenditure.

6.8 Learning system-wide utility functions

When we have multiple concurrent applications, we are trying (in theory, at least) to maximize a "global" (rather than per-application) utility function. The main problem is that the global utility function is not known, and is very hard to learn.

My approach to the global utility function is to assume that it is a simple weighted average of application utilities; the weights reflect the priorities of the application. Initially all applications start at a default priority; we use user feedback (for example, the user clicked on one of the buttons mentioned above) to increase the relative priority of one application over the others.

6.9 Resource allocation and enforcement

When we have multiple applications competing for resources, the system has to allocate global resources among them. The allocation is influenced by the resource requirements of each application, which again is determined by the fidelity that the application is computing at. Thus, by setting the fidelity parameters to maximize the global utility, we are in effect making a resource allocation decision.

In any system that does resource allocation, we need to handle the issue of enforcement. Do we rely on applications to be well-behaved, or do we regulate them in order that they do not exceed their allocations? Adding enforcement to the system would make it more robust; however, it would be a substantial implementation effort with little reward in the form of original research. In this work, I plan to make the simplifying assumption that all applications on the system are well-behaved; the intent is to provide a "best-effort" system that works well in the absence of malicious applications.

If we do not do enforcement, we cannot control the load on the system, but only try to predict it. Even if all Odyssey applications are well-behaved, there might be other resource-consuming applications that are unaware of

Odyssey and its resource allocations; such applications have to be treated as a variable "background load" from Odyssey's viewpoint. Further, some resources such as wireless bandwidth and remote CPU are variable in and of themselves, independently of the load. Thus the system cannot offer any hard guarantees on resource constraints. Supporting applications that require such guarantees (e.g. real-time applications with hard latency constraints) is outside the scope of this work.

6.10 Finding sweet spots

Section 4.6 describes how we can allow applications to describe utility functions in a restricted way, by specifying constraints on various quality metrics. In theory, such a constraint implies that we have a high utility everywhere below the constraint threshold, and a low utility everywhere above it. Of course, most application utility functions will not have exactly this behavior; in fact, the sharp change in utility (if any) might not always occur exactly at the threshold value.

Consider a search operation on a set of documents, some of which are cached locally. Our measure of fidelity is the number of documents searched.If we search only cached documents, our latency is low (say 1 sec); accessing even a single uncached document, however, causes a sharp increase in latency. If the application had specified a latency constraint of 1.2 sec, then we would spend the extra 0.2 sec fetching a small number of documents over a low-bandwidth network. Rather than incur the additional latency for a small increase in fidelity, we should probably restrict the search to the set of cached documents.

What we have here is a "knee", or sweet-spot in the latency-fidelity tradeoff curve. Although we do not know the utility function exactly, we can make a good guess that this corresponds to a point of maximum utility; thus, the right thing to do would be to relax our latency constraint slightly to allow us to find "knees" in the neighbourhood of 1 sec.

I plan to investigate methods to automatically find these knees, where possible, and to use them in making fidelity decisions.

6.11 Object-specific prediction

In general, it is impossible to know what properties of the input data (other than the size) might affect the resource consumption, and how. However, suppose we have already observed the resource consumption in the case of some particular input data, and the application wishes to repeat the computation on the *same* data object. For example, the application might have produced a quick-and-dirty result, and now, at the user's request, wishes to compute a slower but better result. In this case, we already have some very specific information on the resource requirements for that particular object, which already captures the data-dependent effects. Thus, we should probably give more weight to these observations than to those made on computations with different data.

This requires that, for each multi-fidelity computation, the application provide not just the data size, but some sort of unique name for the input data object. I plan to extend the multi-fidelity API to allow this, and the fidelity-resource prediction to make use of this information.

6.12 Applications

I intend to demonstrate a small number of applications using the multi-fidelity API. The intent is to provide a proof-of-concept of the system — to demonstrate the benefits of multi-fidelity computation and to gain experience with the API. We want the following properties for such applications:

- They are interactive, and allow for a fidelity/performance tradeoffs — i.e., the computations that directly affect interactive response can be implemented by multi-fidelity algorithms.

- They place a significant amount of stress on system resources.

- They are interesting from a mobile computing point of view; i.e. they are well-suited to mobile computing or at least reasonably likely to be used in a mobile context.

Section 5 describes three application scenarios — augmented reality, scientific visualization, and on-site engineering calculations — that show multi-fidelity computation at work. I plan to implement and evaluate these three as multi-fidelity applications in my system. I also plan to implement a search application that will use knowledge of cache state to make fidelity decisions.

6.13 Remote execution

Mobile devices, for reasons of cost, weight, and energy-efficiency, are always less computationally powerful than their static counterparts. Thus, it often makes sense for a mobile device to use compute servers accessible over the wireless network, to execute part or all of an application. This gives the system an additional degree of freedom — it can select the fidelity of an algorithm, and it can also select the partitioning of functionality between client and server. I plan to support this behavior by marking each possible partition (in practice, there are likely to be a small number) as a *fidelity mode* of the application. This captures the fact that each of these partitions will have different fidelity/resource tradeoffs, which have to be learned by the system.

Remote execution introduces an additional resource: remote CPU. I plan to implement support for remote CPU monitoring, and also a simple mechanism for remote execution — a generic RPC call. Rather than implement a full-fledged mobile code system, I plan to have statically configured servers that run the necessary code; the client then invokes various parts of this code by making RPC calls. The RPC calls will be directed through Odyssey through the generic RPC mechanism: the system will monitor the network usage, but not interpret the content of the RPCs in any way. This allows us to add new functionality to the application server without requiring modifications to the system.

6.14 Throughput and jitter

Latency is the only "traditional" QoS metric that I have discussed so far in this proposal. For applications that are interactive but not real-time, this is a much more important measure than throughput or jitter. In an interactive application, we are concerned mainly with the speed of interactive *response*: i.e., the latency between a user request and a visible result. The requests might be explicit (user clicks a button to start a search) or implicit (user of an augmented reality program turns their head). These requests are likely to be unpredictable, and extremely unlikely to be periodic.

Throughput and jitter are more suited to the predictable, periodic computations performed by continuous-media, (hard or soft) real-time applications, which have been studied extensively by the QoS community. Most of these applications adapt to changing network bandwidth by changing their *data fidelity*, and support for such adaptation already exists in Odyssey today. In view of this, I do not expect that adding support for throughput and jitter will result in a substantial research contribution, nor in more effective system support for my chosen application domain.

There are some applications, such as scientific visualization (Section 5.2) where network bandwidth is not the only resource consumed — there are significant amounts of computation, with associated fidelity parameters. Further, when the user wishes to visualize the data as an animation, these computations are predictable and periodic. My current plan is to invert the throughput requirements of these applications, and express them as a latency requirement (instead of 10 frames/sec, we would require 0.1 sec/frame).

If time allows, I will extend the implementation to support throughput and jitter as first-class quality metrics; however, I suspect that the implementation effort will be better spent elsewhere.

6.15 Evaluation

There are two main goals that I have in mind in building a multi-fidelity system: it must provide abstractions that allow applications to derive benefit from multi-fidelity, and it must implement these abstractions efficiently. The abstraction allows applications to describe their fidelity metrics and the associated utility functions, i.e., their their adaptive *policy*. The implementation then provides the *mechanism* to implement a wide range of policies.

6.15.1 Evaluating the multi-fidelity abstraction

The purpose of the multi-fidelity abstraction is to make applications more usable. The obvious way to evaluate it is to compare an unmodified application with the modified, multi-fidelity version. If we assume a "perfect" implementation (one that exactly predicts the fidelity-resource relationship, and always finds the best tradeoff points), then we are really comparing the effect of two application policies: the naive, or non-adaptive policy against the adaptive, multi-fidelity policy. In my design the application "policy" is represented by the application's fidelity metric, together with any quality constraints the application might have.

In order to state with complete confidence that one version of an application is more useful than another, we would need time-consuming human-factors studies. I do not intend such a complete evaluation: rather, I intend to sanity-check the improved usability of the application through informal user studies. I expect that with a well-chosen fidelity metric, *under carefully controlled experimental conditions*, the benefit of modifying the application will be apparent.

There is another aspect to evaluating the multi-fidelity abstraction: the *programming cost* of modifying existing applications. I plan to measure programming cost in two ways: a quantitative measure of the amount of source code modification required (in KLOC), and a qualitative evaluation based on feedback from application developers. This feedback should also help refine the API, and to point out limitations in the API design — classes of applications that cannot effectively use the API.

6.15.2 Evaluating the multi-fidelity support

To measure the multi-fidelity system support, I need to answer the question: how well does the mechanism implement policies of various kinds? In other words, how much does the observed behavior deviate from the desired behavior specified by the policy? I expect this aspect of the evaluation to be mostly quantitative. In the augmented reality application, if our policy was always to take 2 sec for every rendering operation, we could measure the deviation from 2 sec of the actual times taken by rendering operations.

In order to understand why the system behaves as it does, we will need another set of evaluation metrics: the behavior of individual components of the system, as measured by micro-benchmarks. I intend to measure the following components:

- The overhead of multi-fidelity system calls.
- The cost of the computation that goes into making fidelity decisions.
- The effectiveness (i.e., the agility and the accuracy) of the performance monitoring and prediction. In order to create repeatable scenarios of resource variation, I plan to use techniques analogous to network trace modulation [21] to vary the availability of battery power, CPU, and memory.
- The agility and accuracy with which the system learns and predicts the fidelity/resource relationships for each application, perhaps compared with some off-line analysis of "what was the best we could have done"?
- Given some fidelity-resource function, how well the system finds sweet spots, or knees.

7 Work items

This section describes the completed and remaining work in this thesis, as an itemized list. The items followed by a single star are considered necessary for a minimum acceptable thesis; those followed by two stars are part of the expected thesis. Three stars mark the bonus items. Work already done is marked by a $+$, and items that I plan to leave to future researchers by $-$.

- API and programming model
  - § Initial draft of API $+$
  - § Extend programming model to handle cache state as a resource $\star$
  - § API support for multiple operation types per app $\star\star$
  - § API extensions for long-running operations (throughput) $\star\star\star$
  - § Multi-dimensional fidelity metrics $-$
- Implementation / Integration with Odyssey
  - § Add multi-fidelity calls to Odyssey. $\star$
  - § Share network prediction across traditional Odyssey apps and multi-fidelity computations. $\star$
  - § Add remote-execution facility (generic RPC) to Odyssey. $\star$
  - § Extend Odyssey cache management interface to allow generic (untyped, wardenless) cache objects. $\star$
- Resource/quality monitoring
  - § Network monitoring/prediction (use existing Odyssey code) $+$
  - § Latency $\star$
  - § Cache state $\star$
  - § Battery power (use Jason Flinn's extensions to Odyssey) $\star$
  - § local, remote CPU monitoring $\star\star$
  - § Memory $\star\star$
  - § Disk bandwidth $\star\star\star$
  - § Disk space $\star\star\star$
- Predicting application behavior
  - § Learning resource consumption functions $\star$
  - § Learning input data specific behavior $\star\star\star$
  - § Using user feedback to learn utility functions $\star\star$
- Making fidelity decisions
  - § Automatically finding sweet spots $\star$
- Applications
  - § Radiosity
    - · Get the vanilla app running $+$

        - modify to use multi-fidelity API, with latency constraint $+$
        - augment UI for multi-fidelity. $+$
        - add support for remote execution. $\star$
        - use resource callbacks to interrupt computation $\star\star$
        - add memory constraints $\star\star$
    - § Quake visualizer
        - Get a vanilla version running in a Linux environment $\star\star$
        - Find a fidelity parameter and a way to tweak it $\star\star$
        - Modify visualizer to use Odyssey API $\star\star$
        - Add multiple fidelity modes corresponding to client/server split $\star\star$
    - § Augmented reality
        - Find augmented reality application (with source code). $\star$
        - Modify application to run on a laptop with a Head-Mounted Display. $\star$
        - Find a fidelity parameter and modify application to use multi-fidelity $\star$
    - § Search
        - Identify a mobile search application that uses cached state. $\star$
        - Extend it to use cache state information to make fidelity decisions. $\star$

- Evaluation
    - § Overhead of system calls $\star$
    - § Overhead of computing fidelity decisions $\star$
    - § Overhead of resource estimators $\star$
    - § Accuracy/agility of resource estimators $\star$
    - § Accuracy/agility of application behavior prediction $\star$
    - § Performance benefits of multi-fidelity to applications $\star$
    - § Tools to control/vary resource availability for experiments $\star$
    - § Efficiency of system in tracking application policy. $\star\star$
    - § Behavior of system with multiple multi-fidelity applications $\star\star$
    - § Programming cost measurements $\star$
    - § Lessons learned about API and its limitations $\star$
    - § Accuracy of sweet spot finder $\star$
    - § Informal user studies with multi-fidelity applications $\star\star$

8 Related work

The importance of *adaptation* in mobile computing systems is now well-recognized [9, 16, 22]. There is much work in various application domains that can be viewed as adapting *data fidelity* to varying network availability [10, 20]. The Odyssey system [20], in particular, is closely related to this research. In many ways — in concept, in design, and in implementation — this work builds on previous work done in Odyssey.

The abstract model of multi-fidelity described in this document is very similar to *Quality of Service (QoS)* models. Of particular interest is the work by Lee et al. [17] that provides a framework to support arbitrary QoS metrics that depend on multiple resources. The Darwin system [2] allows for “service brokers”, which translate application-level resource requirements (quality metrics) to the system level. Rather than have explicit per-application brokers, I propose to have the system learn to be a broker, by monitoring the application. The Rialto operating system [15] has, in its design, the notion of using user preferences to guide resource allocations between applications.

Imprecise computations [6, 7, 14], support graceful degradation of real-time systems under overload conditions. Each computation is modeled as a mandatory part followed by an optional anytime part that improves the precision of the result. The amount of time spent in the optional part can be viewed as a measure of fidelity; the system’s aim is to ensure all mandatory portions meet their deadlines, and to use the remaining resources to improve the fidelity.

The main differences between this work and the QoS work mentioned above are:

- I intend to support a set of non-traditional resources and quality metrics — user attention, cache state, battery — that are important in the mobile domain.
- The applications I intend to support are not continuous-media applications. The latter have periodic behaviours and more or less (at least in the short term) predictable resource requirements. The interactive applications that I focus on perform computations in response to user activity — they are not periodic and their resource requirements may vary with the input data.
- I do not intend to provide real-time guarantees (i.e. strict latency bounds). The assumption will be “better late than never” (unless the user decides to abort the computation), but “better low-fidelity than late”.

There is a large number of algorithms and techniques that can all be brought under the broad heading of “multi-fidelity computations”. The main contribution of this work is in showing that they can all be captured by the same abstraction, and that this abstraction can be effectively used to provide operating system support for real, mobile interactive applications.

Approximation algorithms [11], or more precisely, polynomial time approximation schemes, are algorithms that produce results which are provably within some bound of the true result. The tightness of the bound (i.e., the allowable error) is a tuning parameter analogous to fidelity. *Randomized algorithms* [18] (or probabilistic algorithms) have a non-zero probability of providing an incorrect answer; by running the algorithm many times, one can reduce this probability to an arbitrarily low value. The probability that the answer is correct can be viewed as a “fidelity” metric.

Anytime algorithms [4] can be interrupted at any point during their execution to yield a result — a longer period before interruption yields a better result. *Any-dimension algorithms* [19] are similar, except that they allow more general termination criteria. Since a range of outcomes is acceptable, these of algorithms can be viewed as multi-fidelity algorithms — specifically, as multi-fidelity algorithms that provide an incremental fidelity metric.

There are many techniques used in the real world today that can be viewed as multi-fidelity computation. For example, Hellerstein et al. [12] have proposed various techniques for online aggregation — computation of aggregate statistics over a database. As the size of the sample increases, so does the precision of the result. The Rivl programming language [23] provides resolution-independent operations on multimedia object, which the system could implement at varying resolutions depending on user requirements and system resources.

The LookOut system [13] addresses user-interaction issues that are relevant to this work. It models the cost of demanding the user’s attention, as well as the uncertainty about a user’s goals or preferences. The application of LookOut is to decide whether or not to invoke certain automated services; I am hopeful that similar techniques can be used to learn a user’s preferences for tradeoffs between quality metrics.

9 Expected contributions

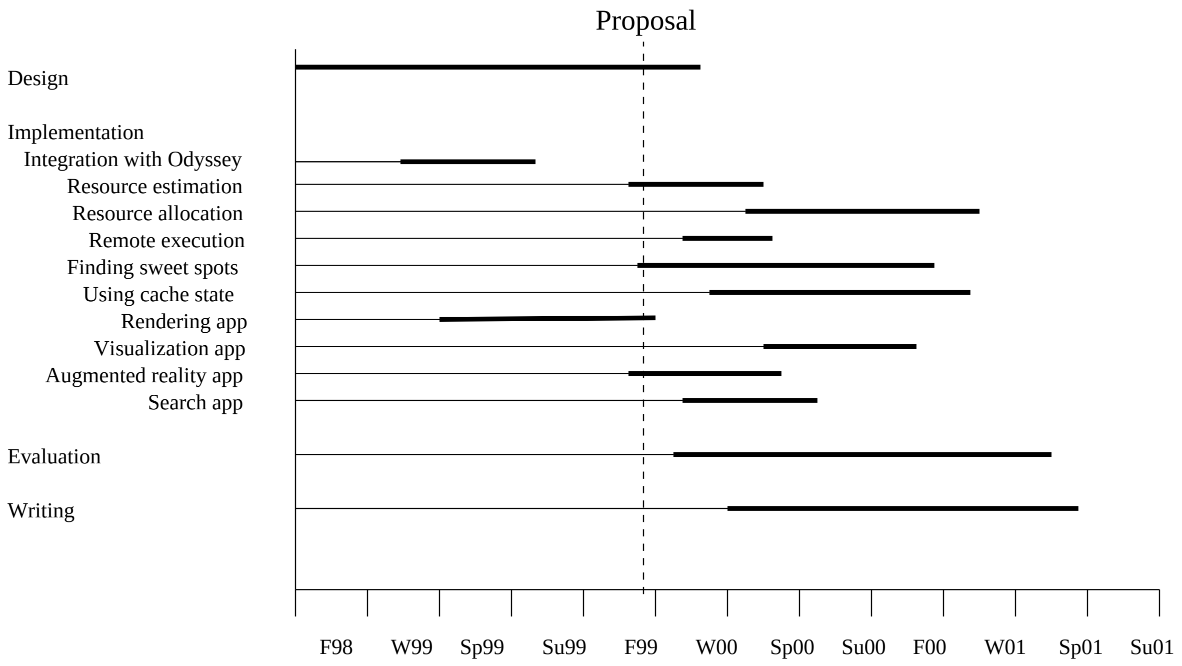
I expect this thesis to make contributions at several levels: at the theoretical and conceptual level, at the systems design and implementation level, and in experimental methodology.

The major conceptual contribution of this work will to define multi-fidelity computation as an abstraction, and to show how this abstraction can be used to provide operating system support for interactive mobile applications. Additionally, this work will provide a programming model and an API for multi-fidelity applications.

The design and implementation contributions will be in the artifacts I build into Odyssey. These will include methods to monitor multiple resources, to learn and predict application behavior, and to automatically find good tradeoffs (sweet spots). Taken together, the artifacts will provide a platform to develop more mobile, interactive, multi-fidelity applications and to study the proof-of-concept applications that I develop.

Finally, the evaluation of the system will make contributions to the field. Informal user studies should provide valuable (if anecdotal) lessons to programmers of multi-fidelity applications on how to choose fidelity metrics and utility functions. Another contribution will be in clearly separating, and measuring, the effects of adaptive policy and mechanism. The adaptive policy of a application determines the maximum benefit that an application could derive from fidelity adaptation. The efficiency of the adaptive mechanism determines how closely observed behavior matches the application's policy goals. I also expect the evaluation process to produce artifacts analogous to network trace modulation [21], that can create realistic and reproducible scenarios of resource variation.

10 Timeline



A The multi-fidelity API

```
/* a fidelity dimension: a real number between 0 and 1, either quantized
   (num_values is number of distinct values) or continuous (num_values is 0)
*/

typedef struct {
int num_values;
}
fid_dim_t;
```

```
/* Value of a fidelity dimension: just a real number between 0 and 1 */
typedef double fid_dim_val_t;

/* a fidelity parameter: could be either a knob or a switch */
/* knob values all range from 0 to 1; num_values specifies the number of
   distinct values (0 if continuous range) */
/* switches are just a set of integers (0 .. num_values-1)

typedef struct {
int is_knob;
int num_values;
}
fid_param_t;

/* Value of a fidelity parameter: real number in [0...1] for knobs, integer
   in [0...(n-1)] for switches */
typedef union {
double knob_val;
int switch_val;
}
fid_param_val_t;

/* A function of n variables: currently we just represent it as a simple
   weighted sum, i.e., an array of n weights. I probably need to extend
   this to allow quadratic and higher-order terms, and to deal with
   enumerated types (like switches), for which real-number operations
   do not make sense */

typedef double *func_t;

/* register our fidelity parameters and fidelity metric */
int register_fidelity(int num_params, fid_param_t *params, int num_dims,
fid_dim_t *dimensions);

/* all the quality metrics we care about */
typedef enum {Q_LATENCY, Q_BATTERYLIFE, Q_MONEY, Q_DISTRACTION} quality_t;

/* Hint a utility function as a function of the four quality metrics */
int hint_utility(func_t utility_func);

/* Specialized form of the utility function: specify  a constraint on
   one quality metric. Optionally register a callback for when the
   constraint is violated. Constraints are upper bounds for latency,
   money, user distraction and lower bound for battery lifetime */

typedef void (*quality_callback_t)(quality_t which_quality, double val);
```

```
int hint_constraint(quality_t which_quality, double constraint,
quality_callback_t callback_fn);

/* rsrc_id enumerates the different types of resources */
typedef enum {
RU_NETBW, RU_NETBW_XMIT, RU_NETBW_RCV,
RU_LCPU, RU_RCPU,
RU_PHYSMEM,
RU_DISKBW, RU_DISKSPACE,
RU_CACHESTATE}
rsrc_t;

/* Hint our resource consumption (for everything other than cache state) as
   a function of fidelity parameters and input size */
int hint_resource(rsrc_t which_resource, func_t rsrc_func);


/* Signal beginning of an operation, and query for the values to set the
   fidelity parameters to */

int begin_fidelity_op(IN double input_size, OUT int *opid,
OUT fid_param_val_t *fidparams);


/* Signal end of an operation, and the fidelity achieved */

int end_fidelity_op(IN int opid, IN double input_size,
IN fid_param_val_t *fidparams,
IN fid_dim_val_t *fid_dims);
```

References

[1] Martin Aeschlimann, Peter Dinda, Loukas Kallivokas, Julio López, Bruce Lowekamp, and David O'Hallaron. Preliminary Report on the Design of a Framework for Distributed Visualization. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99),* Las Vegas, NV, June 1999.

[2] Prashant Chandra, Allan Fisher, Corey Kosak, and Peter Steenkiste. Network support for application-oriented quality of service. In *Sixth IEEE/IFIP International Workshop on Quality of Service*, Napa, CA, USA, May 1998.

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

[4] Thomas Dean and Mark Boddy. An Analysis of Time-Dependent Planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88),* pages 49–54, Saint Paul, Minnesota, USA, August 1988. AAAI Press/MIT Press.

[5] Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo I. Seltzer. Using Latency to Evaluate Interactive System Performance. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 185–199, Berkeley, CA, USA, October 1996. USENIX.

[6] W. Feng and Jane W. S. Liu. An Extended Imprecise Computation Model for Time-constrained Speech Processing and Generation. In *Proceedings of the IEEE Workshop on Real-Time Applications*, pages 76–80, New York, New York, May 1993.

[7] W. Feng and Jane W. S. Liu. Algorithms for Scheduling Tasks with Input Error and End-to-End Deadlines. Technical Report UIUCDCS-R-94-1888, University of Illinois at Urbana-Champaign, Sep 1994.

[8] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. To appear in the 17th ACM Symposium on Operating Systems and Principles, December 1999.

[9] G. H. Forman and J. Zahorjan. The Challenges of Mobile Computing. *IEEE Computer*, 27(4), April 1994.

[10] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 160–170, Cambridge, Massachusetts, October 1–5, 1996. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.

[11] M.R. Garey and D.S. Johnson. *Computers and Intractability.* Freeman and Co., New York, NY, 1979.

[12] Joseph M. Hellerstein, Peter J. Haas, and Helen Wang. Online aggregation. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 171–182. ACM Press, 1997.

[13] E. Horvitz. Principles of mixed-initiative user interfaces. In *Proceedings of CHI '99, ACM SIGCHI Conference on Human Factors in Computing Systems*, Pittsburgh, PA, May 1999.

[14] David Hull, W. Feng, and Jane W. S. Liu. Operating System Support for Imprecise Computation. In *Flexible Computation in Intelligent Systems: Results, Issues, and Opportunities*, Cambridge, Massachusetts, November 1996.

[15] Michael B. Jones, Paul J. Leach, Richard P. Draves, and Joseph S. Barrera III. Modular real-time resource management in the rialto operating system. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 12–17, Orcas Island, WA, USA, May 1995. IEEE Computer Society.

[16] R. H. Katz. Adaptation and Mobility in Wireless Information Systems. *IEEE Personal Communications*, 1(1), 1996.

[17] Chen Lee, John Lehoczky, Dan Siewiorek, Raj Rajkumar, and Jeff Hansen. A scalable solution to the multi-resource qos problem. To appear in the 20th IEEE Real-Time Systems Symposium, December 1999.

[18] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, Cambridge, UK, 1995.

[19] David J. Musliner, Edmund H. Durfee, and Kang G. Shin. Any-Dimension Algorithms. In *Proc. Workshop on Real-Time Operating Systems and Software,* pages 78–81, May 1992.

[20] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, J. Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems and Principles*, Saint-Malo, France, October 1997.

[21] Brian D. Noble, M. Satyanarayanan, Giao T. Nguyen, and Randy H. Katz. Trace-based mobile network emulation. In *Proceedings of the ACM SIGCOMM Conference : Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM-97)*, volume 27,4 of *Computer Communication Review*, pages 51–62, New York, September 14–18 1997. ACM Press.

[22] M. Satyanarayanan. Mobile Information Access. *IEEE Personal Communications*, 3(1), February 1996.

[23] Jonathan Swartz and Brian Smith. A resolution independent video language. In *Proceedings of ACM Multimedia '95*, November 1995.