Operating System Support for Mobile Interactive Applications

Dushyanth Narayanan

August 2002

CMU-CS-02-168

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
M. Satyanarayanan, Chair
Christos Faloutsos
Randy Pausch
John Wilkes, *Hewlett-Packard Laboratories*

This research was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Materiel Command (AFMC) under contracts F19628-93-C-0193 and F19628-96-C-0061, the Space and Naval Warfare Systems Center (SPAWAR) / U.S. Navy (USN) under contract N660019928918, the National Science Foundation (NSF) under grants CCR-9901696 and ANI-0081396, IBM Corporation, Intel Corporation, Compaq, and Nokia.

Keywords: interactive applications, mobile computing, ubiquitous computing, multi-fidelity algorithm, application-aware adaptation, predictive resource management, history-based demand prediction, augmented reality, machine learning

Abstract

Mobile interactive applications are becoming increasingly important. One such application alone — augmented reality — has enormous potential in fields ranging from entertainment to aircraft maintenance. Such applications demand good interactive response. However, their environments are resource-poor and turbulent, with frequent and dramatic changes in resource availability. To keep response times bounded, the application and system together must adapt to changing resource conditions.

In this dissertation, I present a new abstraction — *multi-fidelity computation* — and claim that it is the right abstraction for adaptation in mobile, interactive applications. I also present an API that allows a mobile interactive application to recast its core functionality as a multi-fidelity computation,

I identify one of the key problems in application adaptation: predicting application performance at any given fidelity. I solve this problem in two steps. *History-based prediction* predicts application resource demand as a function of fidelity. A *resource model* then maps application resource demand and system resource supply to performance. History-based prediction is validated through four case studies demonstrating accurate prediction of CPU, memory, network, and energy demand.

I also describe the design and implementation of runtime support for multi-fidelity computations: the overall system architecture as well as each key component. I present four application case studies: of a virtual walkthrough program, a 3-D graphics algorithm, a web browser, and a speech recognizer. In each case, I show how the application uses the multi-fidelity API; that the programming cost of using the API is small; and that the history-based prediction method accurately predicts application resource demand.

In evaluating the system prototype, I ask three questions. First, is adaptation *agile* in the face of changing load conditions? Second, is the system *accurate* in choosing the fidelity that best matches the applications' needs? Third, does the system provide substantial *benefit* compared to the non-adaptive case? I answer these questions through a series of experiments both with synthetic and real workloads. I show that adaptation is agile, accurate, and beneficial in bounding response time despite varying CPU and memory load. I also show that adaptation reduces the variability in response time, providing a more predictable and stable user experience.

Acknowledgements

While I deserve any blame for this dissertation, many people deserve a share of the credit. I could never have completed a work of this size and scope without substantial help and encouragement from colleagues, friends, and family.

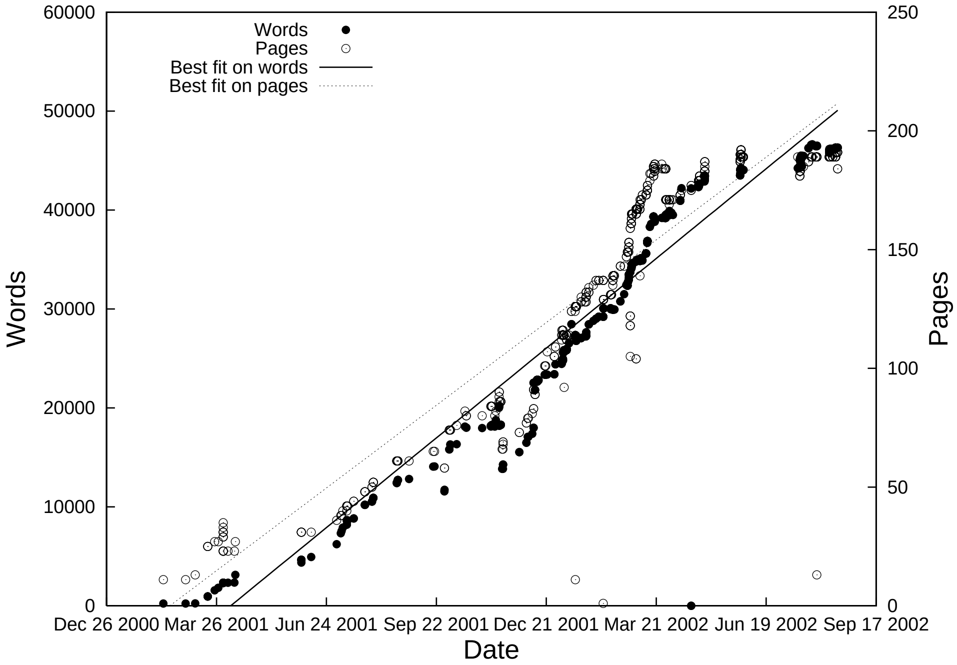
I would like to thank my advisor Satya for his expert technical guidance, for the high standards he always sets himself and his students, and for his confidence in my ability to do quality research. It has been an honour and an education to work with him. Without his ability to see the big picture at all times, I could not have stayed focussed and motivated over the last 7 years.

Large research systems are the products of many minds and man-years. I have been fortunate to work on the same system with several first-class researchers. I would particularly like to thank Brian Noble for his role as mentor, surrogate advisor, and officemate; and Jason Flinn, for an enjoyable and fruitful 5-year collaboration. Jan Harkes provided invaluable Linux expertise and a much-appreciated ability to keep my Coda volume available.

Despite their busy schedules, the members of my committee were always available for feedback and advice. I would like to thank Randy Pausch for keeping me honest about users and interactivity; John Wilkes for much perceptive and constructive criticism, and for keeping me abreast of related work; and Christos Faloutsos for stimulating discussions on a variety of topics.

I could not imagine a more friendly and supportive environment than CMU-CS. The people who make this department a wonderful place are too numerous to list, but special thanks go to Sharon Burks, Tracy Farbacher, David Petrou, Sanjay Rao, Francisco Pereira, Belinda Thom, and Marius Minea. I am also indebted to the management of the 61C Café, to their excellent coffee, and to the many friends I made there, for hugely improving the quality of my life outside computer science.

Last but certainly not least, I thank my parents. Their hard work and vision gave me the educational advantages that brought me to Carnegie Mellon. Despite occasional misgivings about my avoiding the real world, they always encouraged me to pursue my academic goals: without their faith in me and my abilities, I would not have made it this far.



The scatter plot shows the number of words and pages in this dissertation over time. The lines are best fits to the scatter plots computed using least-squares regression.

Figure 1: Size of this dissertation over time

Contents

List of Figures

Symbols used in this dissertation

d_r	instantaneous demand for resource r
s_r	instantaneous supply for resource r
$D_r(T)$	aggregate demand for resource r over a period T
$S_r(T)$	average supply for resource r over a period T
$D_{cpu}(T)$	CPU demand in cycles
$S_{cpu}(T)$	CPU supply in cycles/s
$D_{memory}(T)$	memory demand (working set) in bytes
$S_{memory}(T)$	memory supply (available physical memory) in bytes
$D_{xmit}(T)$	network transmit demand (bytes transmitted)
$S_{xmit}(T)$	network transmit supply (bandwidth in bytes/sec)
$D_{recv}(T)$	network receive demand (bytes received)
$S_{recv}(T)$	network receive supply (bandwidth in bytes/sec)
$D_{rtt}(T)$	network round-trip "demand" (number of RPCs made)
$S_{rtt}(T)$	network round-trip "supply" (inverse of round-trip time)
$D_{battery}(T)$	battery demand (in Joules consumed)
$S_{battery}(T)$	battery supply (Joules available for consumption)
$D_{filecache}(T)$	file cache demand (set of files accessed)
$S_{filecache}(T)$	file cache supply (set of valid cached files)
c_r	instantaneous cost of using resource r
$C_r(T)$	aggregate cost of using resource r over a period T
$M_r(T)$	monetary cost of using resource r
$L_r(T)$	latency cost of using resource r
$E_r(T)$	energy cost of using resource r
$L_{network}(T)$	total latency cost of using the network ($L_{xmit} + Lrecv + L_{rtt}$)
L	total operation latency
L_{opt}	optimal latency for an operation
$E_{latency}$	percentage error of observed latency from optimal/predicted value
f_{20}	bad prediction/deviation frequency: fraction of observations that deviate from optimal/predicted value by more than 20%
E_{90}	common-case error: highest deviation from optimal, or prediction error, observed after discarding the worst 10% of data

Chapter 1

Introduction

An undertaking of great advantage, but nobody to know what it is.
(prospectus of a "bubble" company. Anon., London, 1720)

Wearable computing is already common in today's world, and will be widespread in tomorrow's. Wearable computers will augment a user's environment in a variety of ways, simultaneously interacting with users and their physical environment. Thus, *mobile interactive applications* are of increasing importance. These applications must provide a good interactive experience, but also constrain their resource consumption to the available battery energy, network bandwidth, processing power and memory capacity. User requirements as well as resource conditions vary across space and time: the system and applications must *adapt* their behaviour to track this variation. Adaptation to resource variability is a useful strategy in many mobile computing environments [35, 54, 79]: I will show that, for interactive applications, it is essential.

How best can we design, implement and execute interactive applications in a mobile environment? This question is the main focus of this dissertation. It addresses the problem with a novel theoretical concept — the *multi-fidelity algorithm* — and puts forth the thesis that it is a good abstraction for adaptation in mobile interactive applications. It presents a *resource model* and an Application Programming Interface (API) which allow applications to provide high-level descriptions of their core tasks and policies. It also describes the system support necessary to translate these high-level descriptions into effective runtime adaptation.

The dissertation establishes its claim through the design and implementation of multi-fidelity adaptation in Linux, and four application case studies: virtual reality rendering, radiosity, speech recognition, and web browsing. It demonstrates that the cost of adding adaptation support to legacy code is small; that the runtime system adapts with accuracy and agility to variations in resource availability; and that such adaptation improves application interactive response.

This chapter begins with a brief scenario that exemplifies the application domain addressed by this dissertation, and the challenges presented by it. It then states the central thesis, and presents the steps required to substantiate it. Finally, it provides a road map to the remainder of the dissertation.

1.1 Scenario — 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 the placement of doors, windows, interior walls, and so on. Often, she can immediately reject the proposed modification: even with a "quick-and-dirty" representation, an aesthetic or functional limitation becomes apparent. Occasionally, a more detailed representation is required, and she is willing to wait a little longer to view the result.*
>
> *For hands-free operation, our architect uses a speech-driven interface. At some locations, the wearable computer has good wireless bandwidth to a remote compute server, and can perform speech recognition remotely. Elsewhere, bandwidth and/or server cycles are scarce, and speech recognition must be done locally; since the wearable has limited memory, the recognizer switches to a smaller vocabulary, and asks the user to use a restricted set of words.*
>
> *At some point, the system notices that the battery is running low, and that battery lifetime will fall short of the user's stated goal. It responds by reducing application fidelity and hence power consumption: augmented reality runs at a lower resolution, and when the architect browses web pages, images are compressed to reduce the energy usage of the wireless interface. The user might decide that this is unacceptable, and that she would prefer high fidelity to long battery lifetime: she uses the speech interface to set a less aggressive battery lifetime goal.*

This scenario illustrates the challenges that confront mobile interactive systems. They must continually monitor the availability of multiple system resources; predict the impact of these resource levels on applications; make adaptive decisions for each application that best satisfy user goals; and allow the user to modify these goals on the fly. In order to design and build such a system, we need to answer several questions. What is an appropriate programming model for applications? How can the system monitor and predict resources accurately and cheaply? What other system support is required to ensure that users see crisp interactive response from applications? How do we resolve conflicting goals when making adaptive decisions? What is a simple yet effective way to allow users to express their preferences?

1.2 The Thesis

The thesis directly answers the questions raised above:

> **Mobile interactive applications are best expressed as *multi-fidelity algorithms*. Given system support for resource monitoring, *history-based prediction* of resource consumption, and adaptive decision making, such algorithms can simultaneously optimize across multiple user metrics of fidelity, performance, and resource consumption. It is feasible to implement such system support through a user-level, gray-box approach with no OS kernel modifications.**

This dissertation establishes the thesis in the following steps:

- It argues that mobile interactive applications present special challenges and opportunities, and require a new approach to designing both the applications and the system.
- It introduces the *multi-fidelity algorithm* and shows why it is the right abstraction for mobile interactive applications.
- It provides a resource model and an API for how multi-fidelity algorithms interact with the operating system. This allows applications to express high-level *application metrics* such as performance and fidelity, rather than low-level abstractions such as resources. In four case studies, it shows that the costs of porting legacy applications to the API are modest.
- Applications are concerned with fidelity and performance; operating systems are equipped to manage resources. To bridge this gap, we must be able to relate fidelity to resource consumption and performance. The dissertation shows how *history-based prediction* can map application metrics to system resources in an automatic, accurate, and hardware-independent way.
- It presents the design and implementation of multi-fidelity support in Linux. This support is built on Odyssey, an existing Linux-based platform for mobile computing.
- It presents experimental results that demonstrate that multi-fidelity adaptation is agile, accurate, and leads to better interactive response for applications.

The rest of this dissertation is organized as follows. Chapter 2 introduces a *resource model*: a systematic way to characterize application and system behaviour in terms of resource *supply* and *demand*. In addition to traditional resources such as CPU and network, the model also includes *file cache state* and *battery energy*, which are of great importance when mobile.

Chapter 3 defines *fidelity*, and extends traditional notions of data fidelity to include various adaptive "knobs" and "switches" that are common in interactive applications. It then derives the notion of a *multi-fidelity computation*, and argues that it is a natural fit to the needs of mobile interactive applications. Multi-fidelity computation, along with the resource model, is the formal basis of this dissertation.

Chapter 4 presents a programming model and API for multi-fidelity computations. In designing this API, I was concerned to minimize the amount of modification to legacy application code: this consideration influenced several of my design choices. The chapter locates the API in the design space, and highlights the guiding principles that influenced my design.

Chapter 5 describes the runtime system components that support the multi-fidelity API. Here, one of my key design motivations was *deployability* of the system support across platforms. The chapter argues that *passive resource monitoring* can support effective adaptation; further, that it makes fewer demands on the underlying system than an allocation/enforcement approach. It then describes the overall system design, and each major component in detail: the resource predictors, the solver, the logger, and the remote execution engine. It concludes with an evaluation of the overhead imposed by each of these components.

Multi-fidelity adaptation requires us to regulate an application's resource demand by changing its fidelity. This requires us to know the relationship between fidelity and resource demand, for each hardware platform that the application might execute on. For the application programmer to completely specify this relationship is a heavy burden. My multi-fidelity prototype uses *history-based prediction* to learn this relationship automatically, by logging application behaviour and applying statistical machine learning techniques to the logged data. Chapter 6 shows how application resource demand predictors are built, and describes the key learning techniques used in my application case studies: *least-squares regression*, *data-specific prediction*, and *binning*.

Chapter 7 describes case studies of four mobile interactive applications — radiosity, virtual walkthrough, speech recognition, and web browsing. These are representative of my target domain, i.e. resource-intensive, mobile interactive applications (all four applications would be involved in the motivating scenario described earlier in this chapter). Each case study shows how the application uses the multi-fidelity API; evaluates the cost of porting the application; describes the history-based resource demand predictors for the application; and evaluates the accuracy of these predictors.

In previous chapters, various components of the system were individually evaluated. Chapter 8 is a holistic evaluation of system performance. It answers two questions: does the system adapt effectively; and does the adaptation improve application performance? I evaluate the system with synthetic workloads and synthetic background loads; with real applications and synthetic background loads; and with real applications running concurrently.

Chapter 9 summarizes related work. Chapter 10 concludes the dissertation by identifying and summarizing its key contributions. It then provides an overview of the system's current capabilities and describes ongoing research in multi-fidelity adaptation. Finally, it explores some future research directions opened up by this work.

Chapter 2

A resource model

> *That everything should be of the same weight and measure throughout the Realm —*
> *(except the Common People).*
>
> *(W.C. Sellar and R.J. Yeatman,* 1066 and All That, *ch. XIX)*

This thesis aims to provide a framework for application adaptation in mobile interactive systems. Its focus is adaptation to *resource variation*: varying network bandwidth, CPU speed, and battery charge levels. Designing such a framework requires clear definitions of "resource" and "performance". What is a resource? How is it measured? How is performance measured, and how is it related to resources? Are there different types of resources; what are they? What are the resources relevant to a given system? This chapter answers these questions with a *resource model*.

Most people have an intuitive understanding of what "resource" means: there is the set of things commonly considered computer system resources, and the metrics commonly used to measure them. Unfortunately, this intuitive understanding is inadequate: in this chapter I develop a more rigorous definition. I derive from first principles a resource model that

- describes application-system interaction in terms of resource *supply* and *demand*.
- represents performance as *cost functions* of supply and demand.
- identifies three broad classes of resource and their characteristic supply and demand metrics.
- shows how traditional computer system resources fit this classification.
- provides a criterion for whether a particular resource is *relevant* to a given system.

Though the model often matches the intuitive notion of resource, it does present some surprising results. Some resources have different units for supply and demand; sometimes, supply and demand are not even scalar or numeric values. I will also show how an unusual resource — *file cache state* — fits into the resource taxonomy, and argue that it is a relevant resource for adaptive mobile applications.

I start with a general model that is applicable to a variety of systems — Internet servers, massively parallel supercomputers, and desktop workstations, as well as mobile systems. I identify three resource classes — time-shared, space-shared, and exhaustible — and their characteristic supply, demand, and cost metrics. At various points, I show how to restrict the model, trading generality for simplicity by making assumptions appropriate to mobile interactive systems. I illustrate the restricted model with some examples. Finally, I define a criterion for deciding if a resource is relevant, and list the resources that are relevant to mobile interactive systems.

2.1 What is a resource?

A resource r is a measurable entity that is both *scarce* and *useful*. I.e., there is a finite supply and non-zero demand. A resource r has, associated with it:

- an *instantaneous demand* $d_r(t)$, not always zero: the maximum amount of resource that the consumer can use in the next time interval $[t, t + \delta t]$
- an *instantaneous supply* $s_r(t)$, always finite: the maximum amount that the supplier can provide in $[t, t + \delta t]$
- a set of *cost functions* $c_r(t)$ that compute the instantaneous cost of resource consumption as a function of supply and demand. Each cost function corresponds to a performance metric of interest: e.g., battery drain or monetary cost. We will see that *latency* or wall-clock time can also be a useful *aggregate* cost function, although its instantaneous cost function is trivial (i.e. each period δt corresponds to a latency cost of δt.)

How do we apply this model to application adaptation? Here, the system is the supplier of resources, and the application is the consumer. Given some level of supply, the application adapts its demand to maintain some performance goals. A key assumption here is that instantaneous supply and demand are *independent*: that we can alter demand without changing supply. Of course, present demand can affect *future* supply: e.g., a greater energy drain (demand) leads to a lower charge level (supply). If supply and demand are instantaneously independent, we can *invert* the cost functions: i.e., given the supply and the desired cost (performance), we can compute the appropriate level of demand.

How does this definition of resource help us to describe application adaptation? Applications adapt by changing their behaviour in response to changes in system state. I abstract out those aspects of application behaviour that affect performance, and represent them as application demand for one or more resources. Similarly, aspects of system state that affect performance are represented as system resource supply. This *independence* of supply and demand is crucial to my model: if supply and demand do not affect each other, we can measure or predict them individually, and then apply a cost function to derive the desired performance metric. Independence is a key property of the supply and demand metrics derived in this chapter.

In this chapter I derive independent supply and demand metrics for CPU, network bandwidth, memory, disk bandwidth, disk space, battery energy, and cache state. I also derive cost functions for three of the four key performance metrics in mobile interactive applications: interactive response time or latency, monetary cost, and battery drain. I explain why the fourth metric — user distraction — is outside the scope of the resource model.

2.2 Aggregating resource measurements

So far, we have considered supply, demand and cost as *instantaneous* values. In practice, we need aggregate rather than instantaneous metrics: e.g. battery drain measured over some time interval. What is the appropriate time interval over which to aggregate supply and demand? How can we convert the instantaneous metrics into aggregate ones? This section introduces the notion of *operation*: a natural unit of aggregation, especially for interactive applications. I will argue that time-series are not a good way to do aggregation; in the following section, I will show how to derive aggregate metrics without using time-series.

2.2.1 Operations and interactive applications

The unit of aggregation is the time interval over which we wish to characterize supply, demand, and performance. Often these time intervals correspond to some subtask or computation executed by the application. I call these subtasks *operations*. The notion of operation is natural to interactive applications. A typical interactive application is idle, and consumes no resources, until it receives some user input. It then executes for some time, produces a response, and becomes idle again. This execution is an operation. Interactive response time is simply operation latency, which can be computed by a cost function. More complex applications allow a second user request while the first is being processed, i.e., operations can overlap and run concurrently.

Interactive response times smaller than 100 ms are invisible to most users [12, ch. 2]; response times of several minutes are unacceptable in most applications. Thus, this thesis assumes operation lengths ranging from tenths to tens of seconds. I denote the aggregate resource demand for an operation (or time interval) T by $D_r(T)$; the aggregate supply by $S_r(T)$; and its aggregate cost for some cost metric C by $C_r(T)$.

Figure 2.1 shows a simplified model of an interactive application — a web browser — and an associated operation. The application is normally idle, waiting for user input. When the user clicks on a link, it initiates a "show document" operation. This consists of a network-bound fetch phase and a CPU-bound render phase (I count as "CPU-bound" not only processing cycles, but also time spent on main memory accesses). The supply

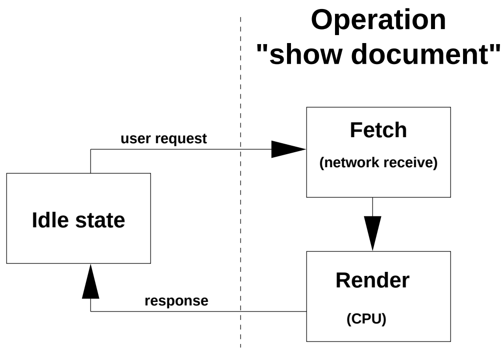


Figure 2.1: A simple interactive application and operation

and demand for CPU and network determine the latency cost of the operation, i.e. the interactive response time.

Many interactive applications are easily described in terms of operations. A speech recognizer might have a "recognize utterance" operation, triggered perhaps when the user pauses between sentences. In an augmented reality application, the user generates a request by moving or turning their head. The application responds by executing a "render new view" operation. The application might have multiple operation types: "add object to scene" and "delete object" in addition to "render new view".

2.2.2 Why not time-series?

When we aggregate, should we retain information about the exact time sequence of resource consumption? We could preserve this information by representing aggregate supply and demand as *time-series*: i.e., infinite vectors of supply and demand values, one for each instant in time. In practice, we would sample at discrete time points, to keep the vector lengths finite.

The time-series approach has a major drawback: it destroys the independence of supply and demand. Consider an application that, at time t, has a small amount Δt of processing time remaining, after which it will block. I.e., its demand for the CPU resource at time t is non-zero. If the supply is zero, i.e. the application is not scheduled at time t, then the demand will still be non-zero at time $t + \Delta t$. On the other hand, if the application is scheduled at time t, the demand at $t + \Delta t$ will be zero. Thus the demand at time $t + \Delta t$ depends on the supply at time t. Further, when the application blocks, it may generate demand for some other resource, say disk. Now the demand for disk at time $t + \Delta t$ depends on the supply of CPU at time t.

Time-series capture the maximum amount of information, but they make supply and demand interdependent. Can we make simplifying assumptions that result in independent metrics, yet retain enough information to be useful? The resource taxonomy in Section 2.3

shows, for each resource class, how to derive aggregate supply and demand metrics that preserve independence.

2.3 A taxonomy of resources

I classify system resources into three basic categories — *time-shared*, *space-shared*, and *exhaustible* — each with a characteristic form for its supply and demand metrics. This classification is not exhaustive; however, it is sufficient for the resources of interest in this thesis. In this section, I describe each class, its characteristic supply and demand metrics, and common resources that belong to it. For each class, I also give an example of a cost metric as a function of supply and demand. My intention in all cases is to use the simplest supply and demand metric that allow a good approximation of performance.

2.3.1 Time-shared resources

Network transmit (and receive) bandwidth and CPU are time-shared or scheduled resources. There is some system-wide rate of supply $R_r(t)$; at any given instant one application receives the entire supply, and others receive none. (I assume that the overhead of switching between applications is negligible.) Thus, for a single application

$$s_r(t) = \begin{cases} R_r(t) & \text{if the application is scheduled to use the resource} \\ 0 & \text{otherwise} \end{cases}$$

When an application wishes to transmit some data, or use the CPU, it can do so at whatever rate the system will support; i.e., its demand is effectively infinite:

$$d_r(t) = \begin{cases} \infty & \text{if the application wishes to use the resource} \\ 0 & \text{otherwise} \end{cases}$$

The instantaneous resource consumption is $\min(s_r(t), d_r(t))$.

When the time-sharing is sufficiently fine-grained, and the cost of switching small, we can approximate resource supply with a General Processor Share (GPS) [73] model: each application gets a steady supply $s_r(t)$, where

$$\sum_{all\ apps} s_r(t) = R_r(t)$$

I use the following aggregate supply and demand metrics for an operation T that lasts for a time $|T|$:

$$S_r(T) = \frac{1}{|T|} \int_T s_r(t) dt$$

$$D_r(T) = \int_T \min(s_r(t), d_r(t))dt$$

I.e., $S_r(T)$ is the average rate of supply to operation T, and $D_r(T)$ is the total amount used by the operation: number of CPU cycles consumed, bytes transmitted, etc.

Note that aggregate supply and demand now have different units: aggregate supply is the *average* rate of supply, whereas aggregate demand is the *integral* over time of demand. My model is that each computation consumes a fixed amount of resource, rather than taking a fixed amount of time. For supply, it is more logical to assume that the system provides the resource to the application at a certain *rate*, rather than a fixed quantum per operation. With these metrics, we can now compute the latency cost of using resource r

$$L_r(T) = \frac{D_r(T)}{S_r(T)}$$

If the operation only uses resource r, and is never idle, then this is the latency of the operation, i.e. $L_r(T) = |T|$. Thus, expressing supply as a rate makes latency a function of supply and not vice versa.

Clearly, GPS is an idealization. However it is an acceptable one in practice, if the time sharing is fine-grained compared to the length of an operation. In this thesis, I treat network bandwidth and CPU as time-shared, GPS resources. For completeness, I also include disk I/O as a GPS resource. However, my current prototype does not support this resource; further, correctly accounting for positional latencies would require a more complex treatment for disk I/O than the simple GPS model.

2.3.2 Space-shared resources

Disk space is the best example of a space-shared resource. There is a fixed total amount of space available: some is used to store data and the remainder is free space. There are three kinds of objects that this space contains:

- objects permanently stored on disk: these are never reclaimed and can be excluded from any computation of supply and demand.
- cached copies of objects, whose originals are elsewhere (e.g. on a remote server) or can be recomputed. I treat the space used for caching as a separate resource: "cache resources" are dealt with later in this section.
- temporary objects, created by the operation and deleted before it completes: these are the only objects that I consider to be part of disk-space demand.

The instantaneous supply is the amount of free space available for temporary objects. The demand is the total space required by the application for the current set of temporary objects.

$$s_r(t) = free_r(t)$$

$$d_r(t) = \sum_{X \in temp(t)} size(X)$$

where $temp(t)$ is the set of temporary objects owned by the application at time t.

The latency cost is negligible if there is sufficient free space for all the objects: I do not model the performance impact of disk space utilization or layout. If there is insufficient space, the application cannot run: then the cost is infinite.

$$c_r(t) = \begin{cases} \infty & \text{if } d(t) > s(t) \\ 0 & \text{otherwise} \end{cases}$$

The aggregate demand is the maximum amount of temporary object space used by an operation; the aggregate supply is the amount of free space that is always available during the operation. Thus, we can define *conservative* aggregate supply and demand metrics:

$$D_r(T) = \max_T d_r(t)$$

$$S_r(T) = \min_T s_r(t)$$

$$C_r(T) = \begin{cases} \infty & \text{if } D_r(T) > S_r(T) \\ 0 & \text{otherwise} \end{cases}$$

2.3.3 Cache resources

Memory is a space-shared resource, but differs from disk space in that it is *virtualizable*. An application's virtual memory might not be completely resident in physical memory: the remainder is on backing store, usually disk. In other words, physical memory acts as a *cache* for virtual memory. I call this a *cache resource*: such resources merit a different treatment from ordinary space-shared resources. A shortage of free space is no longer fatal: there is a finite cost associated with eviction and replacement.

A typical computer system has a large number of cache resources: every level of a memory hierarchy, except the bottommost, is a cache resource. In this thesis I am concerned with two cache resources that have a large impact on mobile interactive applications: physical memory and client file cache. I expect that most general-purpose mobile computing devices will support a file system as well as virtual memory backed by a disk drive, low-power flash-based storage [26], or, in the future, MEMS-based storage [84].

Memory

When the application accesses a non-resident page, it causes a page fault. If the fault is on a new page, it is satisfied cheaply from a pool of free pages; if the page is on backing store, it requires a costly disk access. Thus, it seems insufficient only to measure the amount of memory available to an application: we must consider the current *contents* of memory as well. The supply metric is no longer a numeric scalar value — it must capture the set of resident pages, as well as the number of free pages available to the application.

$$s_{memory}(t) = (Resident(t), freepages(t))$$

Here $Resident(t)$ is a set, while $freepages(t)$ is a scalar.

$$d_{memory}(t) = \begin{cases} \{p(t)\} & \text{if the application is accessing page } p(t) \\ \{\} & \text{otherwise} \end{cases}$$

The cost in terms of disk faults generated is:

$$c_{memory}(t) = \begin{cases} 0 & \text{if } p(t) \in Resident(t) \text{, or } p(t) \text{ is a new page and } freepages(t) > 0 \\ 1 & \text{otherwise} \end{cases}$$

How do we aggregate memory demand and supply over an operation T? We could record the exact time sequence of page accesses (for the demand), as well as that of page evictions (for the supply). This is unwieldy and results in interdependent supply and demand: by making a simplifying assumption, we can reduce both supply and demand to independent, scalar, numeric values.

There are two reasons for paging activity: *cold misses* and *capacity misses*. The former occur on pages that have never been read in from disk; the latter on pages that have been evicted due to memory pressure. In this work, I model the cost of paging only in the pathological case where there are a very large number of capacity misses: i.e., when the system *thrashes* by repeatedly evicting and reloading the same pages.

Thrashing occurs when the working set of an application exceeds the memory available to it. Thus, the memory demand of an operation is the size of its working set. The supply is the number of pages available to it. I approximate the supply by the number of pages available at operation start: the application's resident pages and the free pages available to it. I neglect the paging cost when there is no thrashing; the cost of thrashing is assumed to be infinite.

$$D_{memory}(T) = |Working(T)|$$

$$S_{memory}(T) = |Resident(t_0)| + freepages(t_0)$$

$$C_{memory}(T) = \begin{cases} \infty & \text{if } D_{memory}(T) > S_{memory}(T) \\ 0 & \text{otherwise} \end{cases}$$

Like the GPS model, the working set model is an idealization. It is not valid in situations where cold misses have a major performance impact. It does not capture the benefits of *prefetching*, since the aim of prefetching is to avoid cold misses. For the purposes of this dissertation, these limitations of the working set model are acceptable, and I choose it over a more general but more complex model.

Client file cache

Many applications access data from remote servers; in mobile environments, the client machine often caches this data on local disk. Thus client file cache is also a cache resource, at a different level of the caching hierarchy. This section shows how to model a read-only client cache in terms of supply and demand.

We can model client cache as we do memory; however, the working set model is no longer valid. Cold misses are common, and cannot be ignored: even if an object has been previously cached, it might have been invalidated by the server. On the other hand, thrashing is rare on the time scale of a typical interactive operation.

Thus, for client file caches in mobile interactive environments, we concentrate on cold misses and ignore capacity misses. The demand of an operation is the set of objects accessed; the supply is the set of valid cache entries.

$$D_{filecache}(T) = \{X | X \text{ is accessed by } T\}$$

$$S_{filecache}(T) = \{X | X \text{ is cached and its cache entry is valid}\}$$

Objects that are in the former set but not in the later cause cache misses. Note that it is vital to represent supply and demand as sets of objects, rather than just as the aggregate size of the objects: it is the set representation that allows us to determine which objects will trigger cache misses.

I assume that cache misses are serviced by fetching data objects over a wireless link, and storing them on the local disk. The primary cost of such misses is thus the *network receive cost*, i.e. the total number of bytes fetched:

$$C_{filecache}(T) = \sum_{X \in D_{filecache}(T) \wedge X \notin S_{filecache}(T)} bytes(X)$$

From this, we can easily compute the latency cost as a function of bytes fetched and network receive bandwidth.

This model does not capture the cost of eviction, since I ignore capacity misses. It also ignores staleness: if the system knows, or suspect, that a server has a later version of an object, it would ideally balance the cost of revalidation and refetch against that of using stale data. The model can be extended to support staleness by storing some additional data with each cache entry — its version number, the time it was last validated, and the most recent version number for this object that the client is aware of. Given these, we could derive a probabilistic estimate of staleness, and a *staleness cost metric* based on this estimate. In some cases, staleness can be determined from the object meta-data: e.g., many cacheable web pages come with an “expiry date”.

2.3.4 Exhaustible resources

The time-shared and space-shared resources considered above are renewable: time-shared resources are renewed periodically, and space-shared resources are reclaimed when applications release them. Resources such as battery energy do not fit this pattern: there is a fixed amount of resource available, which is depleted by usage and must be augmented by *recharging*. To characterize the supply of such a resource, we need to know the current resource level $E_r(t)$. When aggregating supply over a time interval, we must also consider the recharging activity during that interval: thus, the recharge rate $R_r(t)$ and the maximum possible resource level $M_r(t)$ are also relevant. $M_r(t)$ is often a constant (e.g. battery capacity): for generality, I assume it varies dynamically. Finally, the resource might not support arbitrarily high rates of resource drain: there might be a maximum allowable depletion rate $A_r(t)$. Thus, the instantaneous supply is the vector

$$s_r(t) = (E_r(t), R_r(t), M_r(t), A_r(t))$$

The instantaneous demand $d_r(t)$ is the rate at which the application wishes to deplete the resource level. The actual depletion rate is limited by both supply and demand.I define the *instantaneous depletion cost* measured in Joules/s:

$$c_r(t) = \min(d_r(t), A_r(t))$$

In the case of battery energy, the supply metric must also capture an important aspect of energy-related state: wall power. I assume that when on wall power, application behaviour has no effect on battery charge level. Then

$$s_{battery}(t) = (E_{battery}(t), R_{battery}(t), M_{battery}(t), A_{battery}(t), Wall(t))$$

($Wall(t)$ is a boolean, true when on wall power and false when not; note that $R_{battery}(t) > 0 \iff Wall(t)$.)

$$c_{battery}(t) = \begin{cases} 0 & \text{if } Wall(t) \text{ is true} \\ \min(d_r(t), A_r(t)) & \text{otherwise} \end{cases}$$

The aggregate demand of an operation T is the total amount of resource consumed by it:

$$D_r(T) = \int_T \min(d_r(t), A_r(t))dt$$

The aggregate supply is the amount of resource that is available for the application: this depends on the current resource level, the recharge rate, and the system policy for rationing the resource. In the case of battery energy, I assume that the operation can run until it completes or the battery is completely drained. In this case, the aggregate supply is the entire available resource, including the recharging that happens during the operation:

$$S_r(T) = E_r(t_0) + \int_T R_r(t)dt$$

The aggregate depletion cost is the amount of resource consumed:

$$C_r(T) = D_r(T)$$

For battery energy, taking wall power into account,

$$C_{battery}(T) = \begin{cases} 0 & \text{if on wall power} \\ D_{battery}(T) & \text{otherwise} \end{cases}$$

For simplicity, I assume that wall-power status does not change during an operation.

In my system, I use Flinn's goal-directed adaptation mechanism [34] to adapt energy consumption. Goal-directed adaptation uses a a variant of the depletion cost metric. The cost of an operation depends on its energy consumption, weighted by the "criticality" of the battery resource. This weight reflects the extent to which the system is meeting the user's goal for desired battery lifetime: it is zero when on wall power, small when battery reserves are plentiful, and increases as these reserves shrink relative to the desired lifetime. The battery lifetime goal would be set by the user based on the amount of work they need to do, or the amount of time they will be away from wall power.

2.4 Combining resources

So far, we have looked at supply, demand, and cost for operations that consume a single resource. How do we characterize operations that use multiple resources? Consider an operation T with a network-bound phase T_{recv}, during which it receives data, and a CPU-bound phase T_{cpu}.

We can still measure the total resource demand as before. The CPU demand $D_{cpu}(T) = D_{cpu}(T_{cpu})$ is the number of cycles used by T_{cpu}, and similarly the network receive demand is the number of bytes received by T_{recv}. We can approximate supply by assuming that the average supply rate is the same for both phases:

$$S_{cpu}(T_{cpu}) = S_{cpu}(T)$$

$$S_{recv}(T_{recv}) = S_{recv}(T)$$

Since the phases are sequential, the total latency

$$\begin{aligned} L(T) &= L(T_{cpu}) + L(T_{recv}) \\ &= L_{cpu}(T_{cpu}) + L_{recv}(T_{recv}) \\ &= \frac{D_{cpu}(T_{cpu})}{S_{cpu}(T_{cpu})} + \frac{D_{recv}(T_{recv})}{S_{recv}(T_{recv})} \\ &= \frac{D_{cpu}(T)}{S_{cpu}(T)} + \frac{D_{recv}(T)}{S_{recv}(T)} \end{aligned}$$

2.4.1 Linear and additive cost metrics

Latency in the above example is an example of a linear, additive cost metric. A linear cost metric for some resource r and operation T satisfies

$$C_r(T) = X_r \cdot D_r(T)$$

where X_r is a constant for the period of the operation. Latency is a linear cost metric with respect to GPS resources: $X_r = \frac{1}{S_r(T)}$ where $S_r(T)$ is the average supply rate. Battery depletion is a linear cost with respect to CPU and network if we assume that each cycle of processing, and each byte transmitted or received, consumes a fixed amount of energy.

An additive cost metric for an operation T that uses resources $r_1, r_2, \ldots$ satisfies

$$C(T) = C_{r_1}(T) + C_{r_2}(T) + \ldots$$

Latency is an additive cost metric if all the resources are accessed sequentially. Battery depletion is always additive assuming an ideal battery: the total energy consumption of the CPU, network, disk, etc. is the sum of their individual consumptions. Monetary cost is always additive.

A cost metric that is both linear and additive satisfies:

$$C(T) = X_{r_1} D_{r_1}(T) + X_{r_2} D_{r_2}(T) + \ldots$$

E.g. an operation that only uses GPS resources, and uses them sequentially, will have a linear, additive latency cost metric. If they are used concurrently, then it is linear with respect to each resource, but not additive. If we consider the effect of memory using the working set model, then it is additive but not linear: the memory effect adds ∞ to the latency if thrashing, and 0 if not.

2.4.2 Concurrent use of resources

If an operation uses resources concurrently, latency is not additive. If all resources are used simultaneously, then

$$L(T) = \max\{L(T_{cpu}), L(T_{xmit}), \ldots\} = \max\{\frac{D_{cpu}(T)}{S_{cpu}(T)}, \frac{D_{xmit}(T)}{S_{xmit}(T)}, \ldots\}$$

More complex concurrency patterns will result in correspondingly complex latency metrics.

2.5 Resource model for mobile interactive systems

In this chapter so far, I presented a general resource model based on the notion of supply, demand and cost metrics. I showed how to describe application behaviour as resource demand; system state as resource supply; and performance as resource cost. I derived supply, demand, and cost metrics for common computer system resources and performance measures. In some cases, I showed how to simplify these metrics by making assumptions specific to mobile interactive systems.

This section further restricts the resource model to the domain of this thesis: application adaptation in mobile interactive systems. We start with some examples of such adaptation, and apply the resource model to them. I then define the resources and cost metrics that are relevant to this thesis; I explain why money is not a relevant resource, and user distraction not a cost metric. Finally, I show how resource dependencies — demand for one resource leading to demand for another — can be captured by appropriate cost metrics.

2.5.1 Example: remote execution

Consider a speech recognizer running on a hand-held device. Due to local resource poverty, the actual computation is done on a powerful remote server. The application records a user utterance and ships it to the server, which does the recognition and returns the result. I am assuming a "request-response" mode of execution, where the request (send) phase, the computation at the server, and the response (receive) phase are all non-overlapping. We are interested in the response time or latency of the entire "recognize" operation, which

- transmits D_{xmit} bytes of digitized speech data to the server,
- performs $D_{remote\ cpu}$ cycles worth of computation on the server,
- receives D_{recv} bytes of recognized text from the server.

The three steps are sequential and the operation latency is additive:

$$L_{recognize} = L_{xmit} + L_{server} + L_{recv}$$

Each step uses exactly one resource: network transmit bandwidth, server CPU, and network receive bandwidth respectively. We already know how to compute latency cost metrics for these resources:

$$L_{xmit} = \frac{D_{xmit}}{S_{xmit}}$$

$$L_{server} = L_{remote\ cpu} = \frac{D_{remote\ cpu}}{S_{remote\ cpu}}$$

$$L_{recv} = \frac{D_{recv}}{S_{recv}}$$

Here S_{xmit}, S_{recv}, and $S_{remote\ cpu}$ are the bandwidth to and from the server, and the processing rate available at the server. Thus, we have the linear, additive latency cost metric:

$$L_{recognize} = \frac{D_{xmit}}{S_{xmit}} + \frac{D_{remote\ cpu}}{S_{remote\ cpu}} + \frac{D_{recv}}{S_{recv}}$$

The remote computation also requires sufficient memory at the server. If we use the simple "working set" latency cost metric (Section 2.3.3), then memory has no effect on latency when there is no thrashing, and it dominates the latency when thrashing occurs. In this case, latency is not linear any more, but it is still additive:

$$L_{remote\ memory} = \begin{cases} \infty & \text{if } D_{remote\ memory} > S_{remote\ memory} \\ 0 & \text{otherwise} \end{cases}$$

$$L_{recognize} = L_{xmit} + L_{remote\ cpu} + L_{remote\ memory} + L_{recv}$$

2.5.2 Example: web browser

Figure 2.1 shows the state transitions of a typical, if simplistic, mobile interactive application: a web browser. The user requests a document; the browser fetches it over the network and renders it to the display. The performance metrics of interest are latency and battery drain. Let us assume the fetch and render are not sequential but perfectly pipelined. In this case, the latency is dominated by the slower of two phases, i.e. we have a non-additive latency cost metric (Section 2.4.1)

$$L_{show\ document} = \max(L_{fetch}, L_{render})$$

Suppose we have a document of size X bytes, which requires P cycles of processing to render. If the available network receive bandwidth is B, and the available processing rate is R, then

$$L_{fetch} = \frac{X}{B} = \frac{D_{recv}}{S_{recv}} = L_{recv}$$

$$L_{render} = \frac{P}{R} = \frac{D_{cpu}}{S_{cpu}} = L_{cpu}$$

Hence

$$L_{show\ document} = \max(\frac{D_{recv}}{S_{recv}}, \frac{D_{cpu}}{S_{cpu}})$$

Battery drain or energy use is still linear and additive: if the CPU consumes P_{cpu} Joules for every cycle of processing, and the wireless interface P_{recv} Joules per byte of data received, then:

$$E_{showdocument} = E_{cpu} + E_{recv} = P_{cpu} \cdot D_{cpu} + P_{recv} \cdot D_{recv}$$

2.5.3 Which resources are relevant?

The resource definition and taxonomy developed here fit a wide variety of resources. Even the three traditional resources of economics fit into the taxonomy: Land as space-shared,

Resource	Type	Demand units	Supply units
CPU	Time-shared	cycles	cycles/s
Energy	Exhaustible	Joules	Joules
Network bandwidth (xmit)	Time-shared	bytes	bytes/s
Network bandwidth (recv)	Time-shared	bytes	bytes/s
Physical memory	Space-shared	bytes	bytes
File cache state	Space-shared	(set of objects)	(set of objects)
Disk I/O (read)	Time-shared	bytes	bytes/s
Disk I/O (write)	Time-shared	bytes	bytes/s
Disk space	Space-shared	bytes	bytes

Note that CPU demand is a measurement of processor time, not of the actual number of instruction cycles used by the application. I simply scale processor time by processor clock speed, to obtain cycles from second. This scaling compensates for differences in measured CPU time due to differences in processor clock speed. However, it does not wholly eliminate the effect of variation in processor architecture: the same computation might still require different numbers of cycles on different processors.

Figure 2.2: Resources relevant to mobile interactive systems

Labour as time-shared, and Capital as an exhaustible resource. Many such resources are not relevant to mobile interactive systems or even to computer systems in general. Gasoline is used in many systems containing automobiles, but is usually irrelevant to a hand-held computer.

What resources and cost metrics are relevant to this thesis, i.e. to mobile interactive systems? I identify four key cost metrics: *latency* or interactive response time, *battery lifetime*, *monetary cost*, and *user distraction*. I then define relevant resources as those that:

- fit into my resource taxonomy.
- are involved in the interaction between a mobile interactive application and the underlying system: there is non-zero application demand and finite system supply.
- have a significant impact on at least one of the cost metrics mentioned above.

On the basis of this definition, we can exclude gasoline from the list of relevant resources. We can also exclude *money*. Application-system interaction might involve monetary cost (e.g. if data is transmitted over a commercial wireless network); but money is not directly involved in the interaction itself. The supplier of money is not the system, but is external to it.

Figure 2.2 shows the resources relevant to a mobile interactive system. Of these, my current prototype supports CPU, memory, energy, network bandwidth, and file cache state on the local machine, as well as CPU, memory and file cache state on remote execution servers.

2.5.4 User distraction: not a resource cost metric

User distraction is an important aspect of an interactive application, especially in a mobile environment. A mobile user is probably engaged in one or more real-world tasks while they are using the computer. It is critical to provide the information or computation required by the user with minimal distraction.

User distraction is a factor of many things, including:

- the input/output devices used. E.g. requiring keyboard use might be distracting
- the number and type of user interactions required. E.g. pop-up dialog windows
- the real-world task that the user is engaged in
- the mode in which results are presented (text, graphical, video, ...)
- violation of user expectation. E.g. a low-quality result where high quality was expected

To express user distraction as a cost metric in my resource model, we would have to represent all these aspects of application behaviour and system state as "resources". Rather than strive for an impossibly general resource model, I chose to treat user distraction explicitly as a metric that is outside the resource model. In Chapter 3 we will see yet another user metric of great relevance: *output quality*. Although we can often trade output quality for resource usage, the former is not directly a result of the latter: rather, both result from adaptive choices made by the application. For this reason, I treat output quality independently, outside the resource model.

2.6 Resource dependencies

In Section 2.5.2, we saw that using wireless network bandwidth has an energy cost. In general, a cost metric for one resource might be in terms of demand for another. E.g. memory paging results in disk access; CPU usage consumes energy; file cache misses cause fetches over the network. We can view these relationships as a dependency graph: there is a dependency from a resource to its cost metrics, which might themselves be resources. Figure 2.3 shows the dependencies between the resources and cost metrics in a mobile interactive system.

Elements that are not supported in my current prototype are shown in italics and dashed lines. The prototype does not support the monetary cost metric, or the disk space and disk bandwidth resources. It "skips over" the dependencies on disk bandwidth; it uses the working set model to directly estimate latency cost as a function of memory supply and demand; and it assumes that the only cost of file cache access is the network fetch in the case of a miss. It also ignores the dependency of energy consumption on CPU, network, and disk bandwidth. Instead, it measures energy consumption directly as a property of application behaviour.

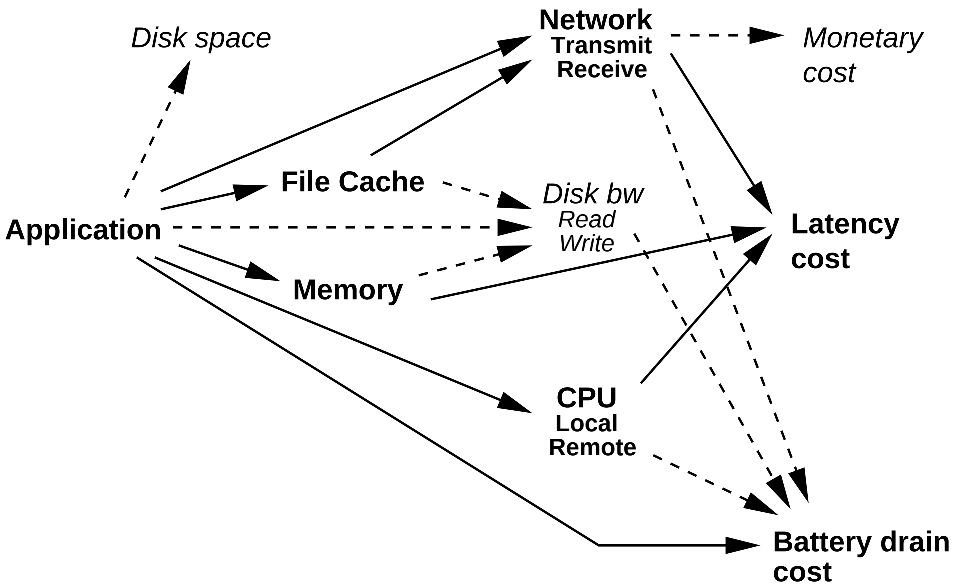


Figure 2.3: Dependencies between resources

2.7 Summary

In this chapter, I presented a resource model based on supply, demand, and cost metrics. I started with an extremely general model, and then simplified the model by making reasonable assumptions about the target domain: mobile interactive systems. I characterized traditional system resources in terms of supply and demand, and also a novel resource: cache state. I showed how to derive cost metrics — latency, battery drain, and monetary cost — as functions of supply and demand. Money and user distraction are outside the resource model; disk I/O and disk space are covered by the model but not supported by the current prototype; CPU, energy, network, memory, and file cache are supported by the model and the prototype.

Chapter 3

Multi-fidelity computation

The reasonable man adapts himself to the world ...

(G.B. Shaw, Reason*)*

Mobile interactive users require low response times and long battery life. In other words, applications must bound the latency and battery drain of interactive operations. As we saw in Chapter 2, these costs depend partly on resource supply, which is highly variable, especially in mobile environments. Wireless network bandwidth varies due to contention, noise, and radio shadows. Time-shared resources see fluctuations in offered load, both on the local host and on remote servers. Hardware energy-saving measures — scaling CPU frequency, reducing network transmit power, or selectively disabling memory banks — are yet another source of variability. The client OS cannot control all of these factors: hence there will always be some variability imposed by the environment.

If we cannot guarantee supply, we must regulate demand: applications must adapt demand to match supply in order to maintain performance. In this chapter, I introduce the *multi-fidelity computation* as a simple yet effective abstraction for adaptation in mobile interactive applications. Multi-fidelity computation generalizes and subsumes many previous approaches to application adaptation, as also several theoretical extensions to the notion of algorithm. This generalized definition of multi-fidelity computation is one of the contributions of this thesis, and the basis of the API and system design described in Chapters 4 and 5.

I start by defining fidelity, and show that the notion of *data fidelity* does not capture the entire range of fidelity adaptation. I complement data fidelity with a new concept: *computational fidelity*, and define multi-fidelity algorithms based on this broader definition of fidelity. I then extend the definition of fidelity further, to include *performance-tuning parameters*. I briefly discuss the role of *nontunable parameters* in fidelity adaptation, and outline the differences between fidelity and *output quality*. Finally, I discuss previous constructs that are similar to multi-fidelity algorithms: approximation algorithms, anytime algorithms, and imprecise computations.

3.1 Fidelity

Previous work has shown the value of adapting data fidelity: degrading data objects to reduce network bandwidth usage [37, 71] or battery drain [31]. Fox et al. [37] have shown that degrading digital images can reduce fetch times for web documents. Noble et al. [71] describe a system that supports multiple concurrent adaptive applications, each with its own metric of data fidelity. Data fidelity is generally defined as the extent to which the degraded version of a data object matches the perfect or reference version.

Data fidelity, while valuable, is not sufficient to describe the entire range of fidelity adaptation. Consider a search over an image database. We could run an approximate version of the query, which runs faster but might be more inaccurate: e.g., we can reduce the *harvest* [36] — the amount of data examined — as in *online aggregation* [46] or *congressional sampling* [4]. Another example of approximate querying is *approximate medians* [60], where accuracy is traded for main memory usage. In all these cases, we degrade fidelity by providing a less trustworthy result: however, this fidelity is not associated with any specific input data item, but with the computation itself. Similarly, we could imagine performing a numerical computation to different degrees of accuracy; again, the accuracy is not a property of the input data, but of the computation.

I define *computational fidelity* as a runtime parameter of a computation that can change the quality of its output without changing its input. Both data and computational fidelity share a key characteristic: they allow us to trade output quality (resolution of an image, or accuracy of a search) for resource consumption. I combine the two into a generalized definition of fidelity:

- A fidelity metric is a runtime parameter to a computation that affects its output quality, either by changing the input data or by changing the computation itself.
- A single computation can have several fidelity metrics: i.e., fidelity can be multi-dimensional.
- Each fidelity metric has a specific range of values, either discrete or continuous. E.g., the lossiness of JPEG compression an image is usually represented as a number between 0, and 100: this JPEG level is a continuous fidelity metric with a value range $[0, 100]$. If the algorithm only allows specific JPEG levels (say 25, 50, 75, and 100), then we have a discrete fidelity metric. Discrete fidelity metrics can also be non-numeric: e.g., a speech recognition computation can choose between a "large" and a "small" vocabulary.
- Fidelity values for each metric are ordered: each value results in a higher output quality than the last.

3.2 Multi-fidelity algorithms

Traditionally, an algorithm has a well-defined *output specification*. E.g., any candidate for a sorting algorithm must preserve all its input elements and order them according to a given criterion. Only algorithms that adhere to this output specification are considered acceptable, and we compare them based on CPU time consumed, memory used, or other metrics of resource usage and performance.

In an interactive application, a fixed output specification will not always match the user's needs. It might exceed the user's needs and waste resources, or fall short of them and generate unusable results. Consider a list of items that the user wishes to browse in sorted order. It is important to produce the first few items quickly (say within 1 s), for good interactive response. The remaining items can be sorted while the user is viewing the first few. Often, the user will terminate the operation after browsing these first few items, resulting in resource savings; if we had sorted the entire list beforehand, we would have wasted resources.

The key feature of the above scenario is that the *output specification* — the number of sorted items to compute — is not known. In fact, in this case we are surer of our *latency constraint* than of the output specification. The latency constraint implies constraints on resource demand: we must not use more CPU cycles or transmit more network bytes than we can within 1 s, at the current level of resource supply. Similarly, battery lifetime constraints imply constraints on the computation's energy demand.

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: we might specify an allowable range, and let the system find the best possible tradeoff.

Since the traditional notion of algorithm does not allow us to perform this inversion, I define a *multi-fidelity algorithm*:

> A multi-fidelity algorithm is a sequence of computing steps that (in finite time) yields a result that falls within a range of acceptable output specifications or fidelity levels. Upon termination, a multi-fidelity algorithm indicates the fidelity level used to generate the result.

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.

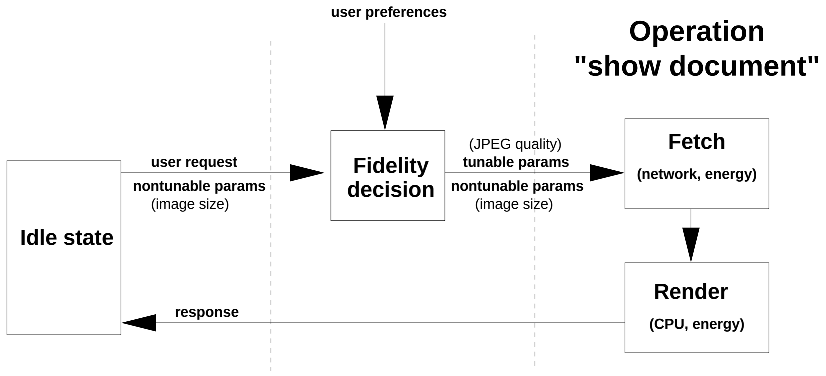


Figure 3.1: An example of an interactive, multi-fidelity operation

The operations discussed in Section 3.1 can all be cast as multi-fidelity algorithms: my definition covers both data fidelity and computation fidelity. By extending any computation with a pre-processing step that degrades its input and a post-processing step that degrades its output, we can create a multi-fidelity computation that supports any data fidelity metrics defined on the input or output data, in addition to any computational fidelity metrics.

3.3 Mobile, interactive, multi-fidelity applications

Multi-fidelity algorithms are an excellent match for my target domain: mobile interactive applications. Mobility results in variable resource supply, necessitating adaptation. Interactivity means that the final result of a computation is viewed by a user, not fed into another computation. As we saw in the sorting example, users are imperfect sources of input and tolerant sinks of output. They rarely specify the exact fidelity that they desire from an operation; often they will accept a lower fidelity output in exchange for lower latency or longer battery life; if they are dissatisfied with the results of adaptation, they are able to take corrective action.

Clearly, there are some interactive applications where fidelity adaptation is undesirable: e.g. a medical imaging application where accuracy is of paramount importance. However, there is a large class of interactive applications where fidelity can profitably be traded for performance — these are the target domain of this dissertation.

We can model an adaptive, interactive application as a collection of operations, each of which could be a multi-fidelity computation. Every time we begin such a *multi-fidelity operation* [83], we have an adaptive decision point: we can choose fidelity values to match current resource levels and latency (or energy) constraints.

Recall the "fetch and display document" operation of Chapter 2. To save network bandwidth, we can JPEG-compress images at a proxy server before fetching them. The lossiness of compression depends on the JPEG quality: a runtime parameter to the compression algorithm. Here the JPEG quality is a fidelity metric: if we reduce fidelity, we reduce output quality as well as network demand. Figure 3.1 shows the multi-fidelity version of the fetch-and-display operation. It is similar to the non-adaptive version, with one additional piece of functionality: choosing the fidelity and acting on that choice. Chapters 4 and 5 describe the design and implementation of the system that provides this functionality.

3.4 Performance-tuning parameters

Some applications have useful adaptations that do not affect output quality at all. E.g., remote execution — running part or all of a computation on a more powerful server machine — does not change its output. However, dynamically choosing the split of computation between local and remote hosts is still extremely valuable in a mobile environment [33]: it allows us to trade local resources for network bandwidth and server resources. Similarly, parametric query optimization [50] tailors the execution plan of a database query to the runtime buffer availability.

Such decisions — how to split a computation, which execution plan to use, or whether to use a CPU-intensive rather than a memory-intensive algorithm — are runtime parameters of the computation. Changing these *performance tuning parameters* affects resource consumption, but not output quality. I view performance tuning parameters as degenerate cases of fidelity metrics: those that have zero impact on output quality. I extend my previous definition of fidelity and of multi-fidelity computation:

> A fidelity metric is a runtime parameter that affects the output quality and/or resource consumption of a computation. A multi-fidelity computation is a computation with one or more fidelity metrics.

3.5 Nontunable parameters

Fidelity metrics are *tunable parameters*: we can adjust their values to regulate resource demand. A computation may have other parameters that affect its resource consumption, but are not tunable. They may have been fixed by the application or the user, or they may be properties of the input data. E.g., the size of the input data often plays an important role in determining resource consumption: the computational complexity [19] of an algorithm is the relationship between an algorithm's input data size and its resource consumption. In Figure 3.1, there is one nontunable parameter: the original (uncompressed) image size. This affects both the compressed image size (and hence the network usage) as well as the CPU cost of rendering the image.

Nontunable parameters are not candidates for adaptation: changes in their values are outside the adaptive system's control. However, their values do impact resource consumption and hence performance. Thus any adaptive system must be aware of nontunable parameters and their effect on resource consumption. Adaptive decision-making will be improved by taking all known nontunable parameters into account: we can view them as the input to the decision process, while the tunable parameters are the output.

3.6 Output quality

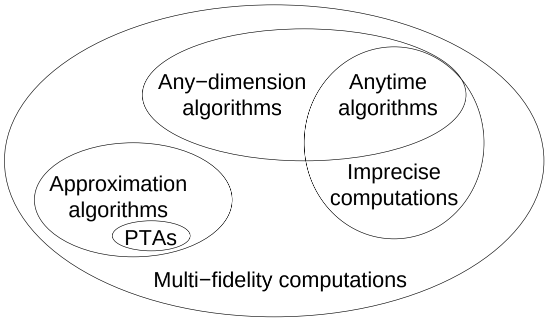
By output quality I mean the user-perceived quality of the results of a computation (independent of its other aspects, such as its performance or monetary cost). Clearly, output quality depends on fidelity; it may also depend on external factors relating to the user and their environment. E.g. the perceived quality of an image displayed by a web browser depends on the display resolution, the ambient light, the user's eyesight, etc. By fidelity, on the other hand, I mean objective, situation-independent measures of how a computation is degraded. Subjective measurements of output quality are outside the scope of this thesis: I assume throughout that output quality is identical to fidelity, or that that there is a fixed, well-known relationship between the two.

3.7 Related Concepts

In addition to the examples of Section 3.1, I am aware of three previous theoretical extensions to the concept of an algorithm that are related to this work. I describe these approaches here, and show that they can be viewed as special cases of multi-fidelity algorithms (Figure 3.2).

Approximation algorithms produce results that are provably within some bound of the true result. A subset of this class — *polynomial time approximation schemes* [39] — have a tuning parameter analogous to fidelity: the error bound ϵ. Lower values of ϵ lead to longer running times, corresponding to higher fidelity requiring a higher resource consumption. Approximation algorithms are typically of interest for intractable (NP-hard) problems, and concentrate on reducing asymptotic complexity. A recent exception is the work by Frieze et al. on approximate methods to do Latent Semantic Indexing [38], a problem for which polynomial time solutions already existed. Similar to approximation algorithms are *probabilistic* and *randomized algorithms* [64], also used to generate polynomial-time solutions for intractable problems. Here the "fidelity" metric is the probability of computing a correct answer, rather than the error bound on the answer.

In contrast, multi-fidelity algorithms are applicable in many situations where low-order polynomial solutions are available. Sorting, for instance, is $O(N \log N)$ in complexity and yet the example given earlier showed why one might use a multi-fidelity algorithm for



The figure shows how approximation algorithms, anytime algorithms, and imprecise computations can be viewed as special cases of multi-fidelity computations. PTAs (polynomial time approximation schemes) are a special case of approximation algorithms; any-dimension algorithms and imprecise computations are two different generalizations of anytime algorithms.

Figure 3.2: Concepts related to multi-fidelity algorithms

this purpose. In the real world, even a reduction of time by a constant factor is extremely valuable. Further, classical complexity measures typically only measure CPU usage, and occasionally resources such as memory and network. They do not consider energy: a critical resource in mobile computing.

A multi-fidelity algorithm may be composed of diverse, unrelated algorithms, one of which is dynamically selected based on runtime tradeoffs. Thus approximation algorithms are properly viewed as a special case of multi-fidelity algorithms.

Anytime algorithms [21] and their generalization, *any-dimension algorithms* [66], are a second important extension to the concept of an algorithm. An anytime algorithm can be interrupted at any point during its execution to yield a result — a longer period before interruption yields a better result. Any-dimension algorithms are similar, except that they allow more general termination criteria. Since a range of outcomes is acceptable, these classes of algorithms can be viewed as multi-fidelity algorithms. However, their generality is restricted by the requirement that valid results be available not only on completion but at all times. Hence, anytime and any-dimension algorithms are also subsets of the more general class of multi-fidelity algorithms.

The third extension, *imprecise computations* [29, 30, 48], supports 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 real-time scheduler ensures that the mandatory portion of every task meets its deadline, and that the overall error is minimized. Since they allow multiple outcomes, imprecise computations are clearly instances of multi-fidelity algorithms. However, their restricted structure and real-time focus makes them a subset of the latter class.

Chapter 4

A multi-fidelity API

Had I been present at the Creation, I would have given some useful hints for the better ordering of the universe.

(Alfonso the Wise)

Chapter 3 claimed that a mobile interactive application should be viewed as a collection of multi-fidelity computations, for the purposes of adaptation. How are these *multi-fidelity applications* written? What system support do they need, and what API should they use to access it?

This chapter describes the programming interface that I have designed for multi-fidelity applications; Chapter 5 describes the runtime system behind the interface. I start with some background: the principles that guided my design decisions, and the previous systems which influenced them. I show that there is a mismatch between application-level and system-level abstractions, and introduce *application-specific predictors* as a way to bridge this gap. I then describe the three components of the programming interface: *library calls*, *application configuration files*, and *hint modules*. I conclude the chapter with a simple example using the API, and outline a 5-step process for porting applications to use the multi-fidelity interface.

4.1 Background

4.1.1 Guiding principles

My primary design goal was a *low barrier to deployment*: to make it easy to create and run multi-fidelity applications. Writing an application from scratch is often prohibitively expensive: thus, I was concerned to minimize the cost of *porting existing applications*. To ease deployment, I wanted the applications and the runtime support to run on off-the-shelf platforms. These considerations led to the following design principles:

- *high-level API*: provide abstractions that are easy for applications to use, rather than those that are natural for the OS to provide.
- *small, minimally invasive API*: the fewer modifications made to application source code, the better.
- *configuration files*: where possible, use text configuration files to communicate information to the system, rather than API calls embedded in the application. These files are easier to write, maintain, and modify than application source code.
- *minimalism*: whenever possible, use existing OS features rather than invent new ones. I use Linux as a base system, and my system runs on two popular kernel versions — 2.2 and 2.4 — and two hardware architectures — x86-based PC and StrongARM-based Itsy [8].

4.1.2 Assumptions

My focus is to provide effective support for adaptation: I assume that any application using this support has some inherent capability to adapt. In other words, the application must have some tunable parameters. Further, I assume access to these tunable parameters, either through source code or some code interpositioning technique. E.g., Puppeteer [20] manipulates the behaviour of Windows application components through well-defined interfaces such as DCOM. Of the four case studies in this dissertation, three are of source-available applications. The fourth is an unmodified web browser augmented with an HTTP proxy: the proxy, for which I had access to source code, transparently adapts image quality on the browser's behalf.

4.1.3 Historical context: Odyssey

This work grew out of previous work in Odyssey [71], a system that supported data fidelity adaptation to changes in network bandwidth. The key abstraction in Odyssey was the *resource callback*: applications registered a tolerance window with Odyssey, and received a callback when resource levels strayed outside this tolerance window. The callback typically triggered an adaptation of data fidelity and a new tolerance window registration.

My implementation is based on Odyssey: it extends the Odyssey API and infrastructure to support multi-fidelity computation. It uses existing features of Odyssey, such as the network bandwidth monitor. Some of my design principles — minimalism and user-level implementation — are inherited from Odyssey. Others — the high-level API and the accent on minimal source code modification — are based on past experience with porting applications to Odyssey.

4.2 The importance of prediction

There is a basic mismatch, or semantic gap, between adaptive applications, the OS, and users:

- application programmers deal in application-specific parameters: fidelities and non-tunable parameters
- the OS manages system state and resources
- users care about user metrics: output quality and performance

Which of these sets of abstractions should the API support? If the goal is to minimize application programmer burden, then the API should talk about fidelities, rather than resources. Similarly, user-specified policies and constraints should be in terms of user metrics.

How will the system translate between fidelities, resources, and user metrics? From the resource model (Chapter 2), performance is a function of resource supply and demand; supply is a feature of system state; and demand depends on application behaviour (i.e., on tunable and nontunable parameters). My design encapsulates these dependencies into components called *predictors*:

- *Resource demand predictors* predict application resource demand as a function of fidelity and nontunable parameters.
- *Resource supply predictors* predict the resource supply available to an application from observations of system state.
- *Performance predictors* predict performance metrics — latency battery lifetime, monetary cost — as a function of resource supply and demand.

In addition to latency, battery lifetime, and monetary cost, mobile interactive users are concerned with *output quality* and *user distraction*. The latter are not performance metrics, and cannot be expressed as resource cost functions. This thesis does not address the construction of predictors for output quality or user distraction: however, the system design allows for such predictors to be easily plugged in, if available.

Supply predictors are generic system components; performance predictors are application-specific, but a single predictor will often serve a large class of applications. Chapter 5 describes supply predictors for CPU, memory, network bandwidth, and energy, as well as the generic "latency" and "battery depletion cost" predictors used by all the applications studied in this thesis. My current prototype does not contain a monetary cost predictor.

Demand predictors are clearly application-specific: however, I wished to automate, to the extent possible, the task of constructing such predictors. I outline my general approach here: Chapter 7 describes and evaluates the specific predictors that I or others built as part of my application case studies.

4.2.1 Demand prediction through logging and learning

My approach to predicting resource demand is empirical. We *sample* the fidelity space by running the application at different fidelities; we *log* the resource demand at each sample point; and we use *machine learning* techniques to build predictors based on the samples. At runtime, the system continues to observe application resource demand: this information is used by the predictors to update themselves using *online learning* techniques.

One could imagine an *analytic* approach to the same problem. Algorithmic complexity analysis [19] can give us CPU consumption as an asymptotic function of input parameters. In the real world, however, constants matter, and these constants vary from one hardware platform to another. We could attempt a more detailed analysis, based on processor specification 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.

I propose that algorithmic complexity be used as a starting point: to guide the learning algorithms that process log data. 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.

4.3 Multi-fidelity API

Based on the design goals explained here, I have designed an API that is

- *small*: the base API has 3 C library calls.
- *high-level*: fidelities are the basic abstractions, and the application does not deal directly with resources.
- uses *application configuration files*: these describe the application's fidelities and nontunable parameters to the system.
- *synchronous*: applications request a fidelity decision at each decision point, rather than wait for the system to suggest a change. A synchronous API is easier to program to: the performance advantages of asynchrony can be obtained by a library layer that exports the synchronous API, but uses callbacks and caching to interact with the underlying runtime system.
- supports *application-specific predictors*, in the form of hint modules.

The remainder of this section describes in detail the C library calls, the application configuration file syntax, and the hint module interface.

4.3.1 C library calls

The basic API consists of 3 calls (Figure 4.1). The application calls *register_fidelity* on startup: the system then loads the specified application configuration file and hint modules (predictors). At the beginning of each multi-fidelity computation, the application calls *begin_fidelity_op*. The values of nontunable parameters are passed as inputs, and the tunable parameters are returned as outputs. The application then runs the multi-fidelity computation with the parameters suggested by the system. On completion, it calls *end_fidelity_op*: this lets the system measure the resource consumption and latency of the computation. This information is logged for future examination, and also used to update the resource demand and performance predictors.

4.3.2 Application Configuration Files

In order to choose fidelity values for an application, the system must know what fidelity metrics the application has, and what their allowable values are. These are declared in an *application configuration file* (ACF) with a simple declarative syntax: a sample ACF is shown in Figure 4.2. Each ACF corresponds to an operation type: thus the API supports multiple operation types per application, although the current system prototype does not.

The ACF declares each fidelity metric with the keyword **fidelity**, and each nontunable parameter with the keyword **parameter**. Each of these can be ordered or unordered. Ordered metrics have real-numbered values in some range, which may be continuous, or discretized with some step value. An unordered metric takes one out of a fixed set of possible values, represented as strings.

The ACF also provides a unique descriptor string that describes the type of multi-fidelity computation; a log file for observations of resource demand; and the method to be used in selecting fidelity values at runtime. The latter is indicated by the **mode** keyword, and has one of two values: **normal**, in which the system tries to pick the most appropriate fidelity values given the current resource levels, and **training**, where the system samples fidelity values randomly. The training mode is used offline to generate logs that cover the entire fidelity space, which are then used to construct predictors.

Hint modules

The ACF also points to a hint module: a binary object file that is dynamically loaded into the system on application startup. The hint module contains resource demand predictors, which are specified in the ACF as entry points into the module. Predictors are written as C functions which receive the fidelity and nontunable parameter values, and a "snapshot" of the current resource supply levels, as input (Figure 4.3). The module can also specify an application-specific latency predictor to override the generic latency predictor. The latency

```
/* register operation type with runtime system. Input is a pointer to the
   Application Configuration File for that operation type. Output is a
   unique identifier for the operation type. */
int register_fidelity(IN char *conf_file, OUT int *optype_idp);

/* query runtime for appropriate fidelity, before starting an operation.

   inputs are:
      dataname: a unique identifier for the operation's input data
      optype_id: the identifier returned by register_fidelity
      num_params: number of non-tunable parameters
      params: array of non-tunable parameter values
      num_fidelities: number of tunable parameters (fidelities).

   outputs are:
      fidelities: array of tunable parameter (fidelity) values
      opidp: unique identifier for this operation
*/
int begin_fidelity_op(IN const char *dataname, IN int optype_id,
      IN int num_params, IN fid_param_val_t *params,
      IN int num_fidelities,
      OUT fid_param_val_t *fidelities,
      OUT int *opidp);

/* report completion of an operation. Inputs are the application id,
   the operation id, and a failure code. The latter can be
       SUCCESS: successful completion
       ABORT:   If the program crashes without calling end_fidelity_op, the
                runtime will log a value of ``ABORT''; it is not meant to
                be passed in directly to this function.
       USER_ABORT: operation aborted by user
       RSRC_ABORT: operation aborted due to resource constraint violation
       FAILED:     operation failed for some other reason
*/
int end_fidelity_op(IN int optype_id, IN int opid, IN failure_code failed);
```

Figure 4.1: The multi-fidelity API

```
description radiator:radiosity
logfile /usr/odyssey/etc/radiator.radiosity.log
mode normal
constraint latency 10
param polygons ordered 0-infinity
fidelity algorithm unordered progressive hierarchical
fidelity resolution ordered 0.01-1

hintfile /usr/odyssey/lib/rad_hints.so
hint cpu radiator_radiosity_cpu_hint
hint memory radiator_radiosity_memory_hint
hint latency radiator_radiosity_latency_hint
update radiator_radiosity_update
utility radiator_radiosity_utility
```

ACF for the radiosity computation. There is one nontunable parameter — the number of polygons in the input data — and two tunable parameters — the choice of radiosity algorithm, and the resolution. The file also specifies a latency constraint of 10 seconds on every radiosity computation, and provides a hint module with CPU, memory, and latency predictors.

Figure 4.2: Example Application Configuration File

```
typedef int (*hint_func_t)(IN char *dataname,
                           IN fid_param_val_t *params,
                           IN fid_param_val_t *fidelities,
                           OUT double *val,
                           IN struct snapshot *res_snapshot,
                           IN struct server *whichserver);

typedef int (*update_func_t)(IN char *dataname,
                             IN fid_param_val_t *params,
                             IN fid_param_val_t *fidelities,
                             IN int numvals, IN struct res_value *vals,
                             IN failure_code failed);
```

Predictors and utility functions are of type *hint_func_t*. They receive as input a unique identifier for the operation's input data (*dataname*), the nontunable parameter and fidelity values, a resource supply snapshot, and the server chosen for remote execution (if any). They return a floating point value which is the predicted resource consumption. Update functions are of type *update_func_t*, and are passed the dataname, fidelity and parameter values, an array containing resource demand and latency values, and the failure code passed by the application to *end_fidelity_op*.

Figure 4.3: Signatures for hint module functions

predictor invokes the resource demand predictors, and then computes latency as a function of resource supply and demand.

The hint module can also contain an *update function*. This function, if present, will be invoked at the end of each operation and passed the fidelity, nontunable parameters, the resource consumption and performance of that operation. It can then update the internal state of the predictors using this information.

Finally, the hint module contains a *utility function*: a function that maps each adaptive choice to a number representing user satisfaction or utility. Thus the utility function encodes the user's policy for trading off between fidelity and performance. The utility function is passed the fidelity and nontunable parameters; it can then invoke the resource demand and/or performance predictors, and use these predictions to calculate the desirability of any particular fidelity choice. The utility function is expected to return a real number in the range $[0, 1]$.

Resource constraints

As we saw in Chapter 3, one of the advantages of multi-fidelity computations is the ability to *invert* the roles of fidelity and resource consumption: to produce the best fidelity, while keeping resource demand within some bound. I call these bounds *resource constraints*. These resource constraints might be obtained

- from the user: e.g. a soldier on the battlefield might want to limit radio network transmission to avoid detection.
- by observing resource supply: e.g. the supply of memory acts as a constraint on the demand, if we assume the working set model from Chapter 2.
- from higher-level performance constraints: a user-specified bound on latency results in a bound on CPU demand; the value of the CPU bound varies with the CPU supply.

Resource and latency constraints can be specified statically in the ACF; they can also be added or updated at runtime through a *hint_constraint* call. Optionally, the application can also specify a tolerance bound and a *constraint violation callback* function: if the resource usage (or latency) of an operation exceeds the tolerance bound, the callback function will be invoked. Currently, my prototype supports callbacks only for latency constraints.

4.4 Example use of API

Figure 4.4 shows an example use of the basic multi-fidelity API by a radiosity application. During initialization, the program calls *register_fidelity* and the system reads the Application Configuration File shown in Figure 4.2. The application then waits for user requests. When it receives a user request, it invokes *begin_fidelity_op*, passing in the name of the input data file and the number of polygons it contains (i.e., a nontunable parameter).

```
...
int optype_id;
register_fidelity("/usr/odyssey/etc/radiator.radiosity.conf",
                  &optype_id);
...
while (!done) {
      int opid, success;
      char *objname, *algorithm;
      double num_polygons, resolution;
      fid_param_val_t nontunables[1], tunables[2];
      ...
      get_user_request(&objname, &num_polygons);
      ...
      nontunables[0].ordered_val = num_polygons;
      begin_fidelity_op(objname, optype_id, 1, nontunables,
                        2, tunables, &opid);
      resolution = tunables[0].ordered_val;
      algorithm = tunables[1].unordered_val;
      ...
      success = do_radiosity(objname, resolution, algorithm)
      ...
      end_fidelity_op(optype_id, opid, success);
}
...
```

Figure 4.4: Sample use of multi-fidelity API

begin_fidelity_op returns the tunable parameter values: the algorithm and resolution to use. After completing the radiosity computation, the application calls *end_fidelity_op*.

4.5 Conclusion

The process of porting an application to the multi-fidelity API consists of five steps:

- *Describe*: Create an ACF that list the application's fidelity metrics and nontunable parameters.
- *Modify*: Insert calls to the multi-fidelity API into the application.
- *Measure*: Acquire a log of resource demand at different fidelity values, by using the system in "training" mode and running the application repeatedly.
- *Learn*: Use machine learning techniques to build demand and performance predictors based on the logs.
- *Hint*: Build a hint module containing the application's utility function and the predictors.

In Chapter 7 I show, through four application case studies, that the one-time *cost* of porting applications is modest; in Chapter 8 I show that the runtime *benefits* of multi-fidelity adaptation are considerable.

Chapter 5

System support for multi-fidelity computation

I must Create a System, or be enslav'd by another Man's.
(W. Blake, Jerusalem, *pl.10, 1.20)*

What system support does the multi-fidelity API require? What functionality can be supplied by generic system components rather than application-specific ones? If application-specific components are required, how can the system aid in building them?

In this chapter I describe the design and implementation of runtime support for multi-fidelity computation. I outline the principles that guided the design process; I then describe the high-level design, followed by a description of each key component. Some of these components were developed by other researchers: these are described briefly, while components developed by me as part of this thesis work are described in more detail. I conclude with a micro-benchmark based measurement of the overheads imposed by each runtime component.

5.1 Design Rationale

Deployability was the key consideration in my runtime design, just as application portability was in the API design. I wanted to deploy both applications and runtime on off-the-shelf platforms with minimal effort. This led to a design philosophy that favours

- *minimalism*: I use features of the underlying OS where possible, rather than reimplement them.
- a *gray-box* [6] approach that avoids kernel modification. My runtime is implemented as a user-level process that occasionally relies on knowledge of kernel internals, but never on modification of these internals.

What runtime support does the multi-fidelity API require? The runtime system's primary responsibility is to *make fidelity decisions* for the application at the beginning of each multi-fidelity operation. Secondarily, the system should provide monitoring and logging of application resource demand, for performance debugging purposes and to build predictors for resource demand in the future.

The focus of this dissertation is *application* adaptation: at each decision point, the system tries to pick the application's tunable parameter values to achieve the best possible tradeoff between performance and output quality. Clearly, application performance is also affected by OS behaviour: changes in resource allocation, CPU frequency or disk spindown policy will affect the application's resource supply. My approach is to *observe* and *predict* system behaviour — i.e. resource availability and allocation — but not to modify it.

A purely predictive approach [23] has several advantages:

- it preserves the assumptions of my resource model by avoiding cyclic dependencies between resource supply and demand. Rather than attempt to manage both supply and demand, my approach predicts supply and regulates only demand.
- no QoS guarantees are required from the underlying OS: I require a weaker property, that of *predictability* of resource allocation. Predictability requires only that the OS's allocation algorithms be known, and that the OS export sufficiently detailed statistics to enable us to second-guess its resource allocation decisions. However, it does not require that these allocation decisions conform to any pre-arranged notion of fair share, proportional share, or guaranteed rate. We will see that while the allocation decisions of a real OS (Linux) are not perfectly predictable — not all of the relevant state is visible at user level — we can still build predictive models that work well in practice.
- the approach works equally well for resources where guarantees are impossible: e.g., wireless network bandwidth is affected by environmental conditions that are outside the OS's control.
- legacy, non-adaptive applications are assured that their resource allocations and performance will not be affected: the OS continues to make the same allocation decisions as before.
- a prediction-only approach makes it easier to avoid OS modification using a gray-box approach: we can observe and predict system behaviour entirely at user level, perhaps with knowledge of kernel internals but without needing to modify them.

The disadvantage of a prediction-only approach is a lack of integration between system-level resource management and application-level adaptation. When there are multiple applications running, the system could potentially make better decisions if it knew how each application would adapt in response to those decisions. This thesis does not address the hard problem of integrating resource allocation, hardware management, and application adaptation: the system leaves the first two to the default OS mechanisms, and passively observe and predict their effect on application resource supply.

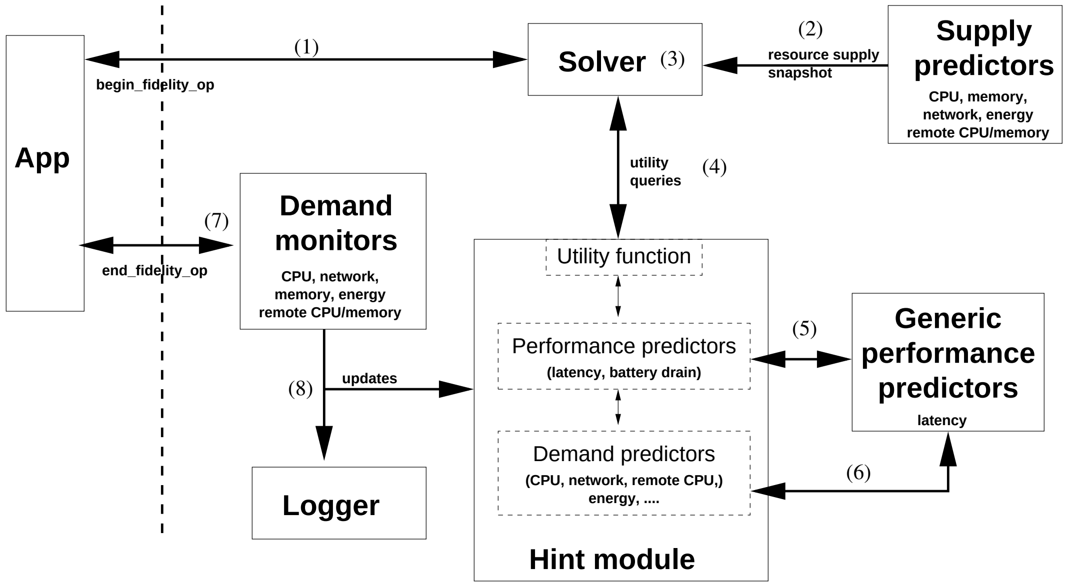


Figure 5.1: System support for the multi-fidelity API

5.2 Design

Figure 5.1 shows the high-level design of the runtime system. Its primary functionality — making fidelity decisions — is triggered by a *begin_fidelity_op* call from the application, and is implemented by the following steps:

1. The application passes in the values of the nontunable parameters.
2. A set of *supply predictors* predicts the application's resource supply for the near future. This information is captured in a "resource supply snapshot".
3. A *solver* searches the space of tunable parameters to find the best possible values. It evaluates candidate values for the tunable parameters by computing their goodness or utility.
4. Utility is computed by the application-specific *utility function* in the hint module. This function is given the tunable and nontunable parameter values as well as the resource supply snapshot.
5. The utility function invokes *performance predictors* that compute the latency and battery drain resulting from this particular choice of tunable parameters.
6. Performance predictors in their turn, call application-specific *resource demand predictors*; they then combine supply and demand to compute performance.

The second piece of functionality — monitoring and logging — is triggered by a call to *end_fidelity_op*:

7. *Resource demand monitors* compute the aggregate resource demand of the operation just concluded.
8. A *logger* records the resource demand, tunable and nontunable parameter values, and the success/failure code of the operation in a disk file.

I divide the runtime design into *generic* and *application-specific* components. The solver, supply predictors, demand monitors, and logger are generic system components. The hint module, which contains the utility function, demand predictors, and performance predictors, is application-specific. Although performance prediction is application-specific, a simple latency prediction scheme works for a large class of applications. I include among the generic system components a latency predictor that is used by all the applications studied in this dissertation.

This chapter describes the implementation of the generic system components: the solver, the supply predictors, the generic latency predictor, the demand monitors and the logger.

5.3 The solver

The solver's task is to search the space of tunable parameters and find the set of values that maximize the operation's utility. It supports two basic types of parameter:

- *discrete*: these have a *finite* set of possible values: e.g. integers from some range, or one of a set of enumerated values.
- *continuous*: these take a real-numbered value from a specified range. In other words, the set of possible values is *infinite* but *bounded*.

My search strategy is very simple: for discrete parameters, the system exhaustively searches all combinations of values. Although this approach has complexity exponential in the number of parameters, it works well when the number of parameters is small. For continuous parameters, it finds the optimal values by a gradient-descent approach. When there are both discrete and continuous parameters, I combine the two strategies: for every combination of discrete parameter values, the system runs the hill-climbing algorithm to find the optimal values of the continuous parameters. I have found that this simple solver is sufficient for the applications I studied; more sophisticated solvers can easily be plugged into the system without modifying other components.

5.3.1 Restricted vs. unrestricted utility functions

The difficulty of the solver's task depends largely on the utility function. We can broadly classify utility functions into two types, *restricted* and *unrestricted*.

Restricted utility functions are constrained to some parametrized form, e.g. polynomial or piecewise linear. Specifying a utility function consists of specifying the parameter values: e.g. the coefficients of a polynomial. With restricted utility functions, the solver can exploit its knowledge of the underlying functional form for faster, more accurate, or more robust operation. E.g. Lee [57] has shown efficient, near-optimal solutions based on integer programming for weighted-sum utility functions over discrete, ordered parameter spaces.

The disadvantage of restricted functions is that we must coerce all utility functions into the specified form. I intend utility functions to be a general expression of application policy, and it is not clear if any given form will suffice for all useful application policies. Further, utility is a function of performance, which is a function of resource demand. This means, for example, that a linear utility function is only usable by applications for which both resource demand and performance are linear functions of the tunable parameters.

I decided to use unrestricted utility functions: I only constrain the range of the function to $[0, 1]$ and require that the same inputs will always produce the same outputs (idempotency). This allowed me to

- use a procedure-call interface between the solver and the utility function. This interface is standard and independent of solver implementation.
- write utility functions in C, rather than devise a special syntax for them. These C functions are compiled and loaded into the solver's address space.

The disadvantages of this approach are:

- utility functions are Turing-complete and the solver can make no assumptions about their behaviour. A pathological utility function can cause the search to fail, to be inefficient, or to generate suboptimal results.
- loading utility functions into the solver's address space is unsafe: a buggy or malicious function can crash the system.

Unrestricted utility functions give us simplicity, generality, and efficiency at the expense of safety and robustness. In Section 10.3 I will discuss alternative representations that address these limitations.

5.3.2 Search vs. feedback-control

An alternative model to search-based adaptation is *local feedback-control*: as the system runs, it continually evaluates its current performance, and incrementally adjusts fidelity upwards or downwards as required. A purely local approach has many drawbacks. It cannot respond quickly yet accurately to dramatic changes in the environment: that requires large yet accurate changes to tunable parameter values, which are impossible if it can not predict the effect of these changes on performance. Thus, many iterations of the feedback loop will be required before we converge on the correct parameter values. Further, if there are multiple tunable parameters, we can not know which one will give us the desired improvement: we must try each in turn, making adaptation even less agile.

In my design, I use feedback-control only to *augment* prediction. E.g., battery energy is managed through *goal-directed adaptation* [34], a form of feedback-control. I also use feedback techniques when application behaviour varies in ways that the predictors did not anticipate. In such cases, I use feedback to correct the predictors, rather than directly adjusting the application's tunable parameters.

5.4 Utility functions and resource constraints

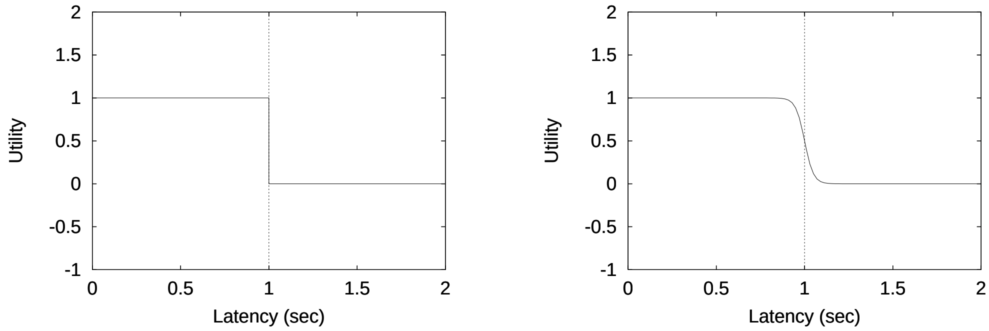
In addition to utility functions, applications or users might have per-operation *constraints* on resource demand or cost (e.g. latency). As we saw in Chapter 3, a valuable feature of multi-fidelity computation is the ability to invert the roles of output quality and resource consumption: to say "give me the best possible quality without using more than X units of resource." Resource or performance constraints might arise from the system, the application programmer, or the user:

- To satisfy battery lifetime goals, the runtime system dynamically sets constraints on per-operation energy usage, based on the current battery level and the desired battery lifetime. Similarly, to ensure that excessive memory demand does not cause thrashing, the runtime system computes a constraint on each operation's memory demand, based on the currently available memory supply.
- The application programmer can specify a latency constraint in the ACF, based on their knowledge of acceptable response times for the operation.
- the user can override the application's default constraints, or add additional ones. E.g., a soldier on the battlefield could constrain network transmission to avoid detection.

I assume that performance and resource constraints have a binary effect on user utility. If the constraint is violated, then utility is greatly reduced; if the constraint is satisfied, utility is unaffected. E.g., when a memory constraint is violated, the system thrashes, and utility is near zero. As long as we stay within the constraint, memory demand has no effect on utility. I represent this binary effect by multiplying a *step function* (Figure 5.2(a)) into the utility function. When the constraint is violated, the step function's value is 0, and the utility is forced to 0. When the constraint is satisfied, the value is 1, and utility is unchanged. The advantage of this multiplicative approach is that we can compose an arbitrary number of functions in this way, yet ensure that the range of the resulting function is always $[0, 1]$.

5.4.1 Constraints as sigmoids

The disadvantage of step functions is that they do not allow us to specify *tolerance* or "slack". E.g., the user may desire a latency constraint of 100 ms: all values below this are equally good. Above 100 ms, the user's utility degrades steadily until it is near zero at 1 s:



(a) Step function (b) Sigmoid (0.1 sec tolerance)

Figure 5.2: Utility function for a latency constraint of 1 sec

all values above this are equally bad. Thus, the region the region between 100 ms and 1 s is a *tolerance zone*, where the user's utility degrades gracefully rather than sharply.

To incorporate the notion of tolerance, I use *sigmoid* (Figure 5.2(b)) rather than step functions. A sigmoid for a constraint value μ and a tolerance σ has the form

$$\frac{1}{1+e^{(x-\mu)/\sigma}}$$

It goes from 1 at $-\infty$ to 0 at $+\infty$. The curve has three regions: the "good" part $[-\infty, \mu-\sigma]$, the tolerance zone $[\mu-\sigma, \mu+\sigma]$, and the "bad" part $[\mu+\sigma, +\infty]$. The utility at the tolerance zone boundaries is 0.88 and 0.12 respectively; in the tolerance zone, utility degrades almost linearly. This means that, even in the "bad" zone, utility can be as high as 0.12. To avoid this, I actually use the sigmoid

$$\frac{1}{1+e^{4(x-\mu)/\sigma}}$$

which has boundary values of 0.98 and 0.02: the multiplier of 4 ensures that the utility outside the tolerance zone boundaries is very close to 0 (or 1).

My runtime system provides a helper function that computes sigmoids for any given constraint and tolerance values. Application utility functions can use it to incorporate user constraints on latency or resource demand into the utility. System-determined constraints on memory and battery drain are multiplied in by the runtime system before the final utility function is provided to the solver.

5.5 System resource predictors

My current prototype has five supply predictors — for CPU, memory, network, energy, and file cache — one generic performance predictor — for latency — and one generic demand predictor — for file cache. Of these, the CPU, memory, and latency predictors were developed as part of this thesis work: I describe them in detail, and the others briefly.

5.5.1 CPU

This component predicts, at the start of each multi-fidelity operation, the CPU supply available to it in cycles/s. It is based on a number of simplifying assumptions:

- the computation of interest is contained within one process or thread, and its behaviour is independent of the other processes in the system.
- this process is entirely CPU-bound during the computation
- the scheduler is *round-robin*: it gives an equal processor share to all runnable processes.
- we have *temporal locality*: the recent past predicts the near future when predicting the number of runnable processes
- temporal locality operates at all time scales of interest. I.e., the average behaviour of the last 1 s predicts the average for the next 1 s, the last 10 s predict the next 10 s, etc. In this dissertation I am concerned with time scales between 0.5 s and 50 s.

These assumptions are idealistic, but we can derive from them a simple predictive model that is effective in practice. I predict a process p's CPU supply over the next T seconds as

$$S_{cpu} = P/N$$

where P is the processor speed, and N is the predicted number of runnable processes — the *processor load* — over the next T seconds. To predict N, the system periodically samples the instantaneous load average from **/proc/loadavg** and smoothes this measurement using an exponential decay scheme:

$$N_{i+1} = \alpha N_i + (1 - \alpha) n_i$$

Here N_i is the smoothed value corresponding to the i'th raw measurement n_i. The decay constant α is computed based on the desired *decay time constant* T_S:

$$\alpha = e^{-t_p/T_S}$$

where t_p is the time interval between samples: my prototype uses a t_p of 0.5 s. With this value of α, each sample will decay by a factor $1/e$ every T_S seconds, i.e., T_S determines our "window" into the past. Based on the assumption of temporal locality over all time scales, T_S is set to the time period T over which prediction is desired. This is also the desired latency of the computation: if we are aiming for a latency of 1 s, then we try to predict the CPU supply over the next 1 s.

This load-smoothing scheme is identical to the standard load smoothing scheme used in Linux. However, Linux only computes smoothed values with certain fixed decay time constants (60 s, 300 s, and 900 s). My scheme allows the decay time constant T_S to be chosen dynamically to match the desired time period of prediction. We could also imagine more complex autoregressive models [10, 23] for better prediction accuracy: however, the simple exponential-decay scheme works sufficiently well for my purposes.

So far I have assumed that the future load of the system is predicted by the past load, at the instant process p begins a computation. This is a reasonable assumption for all processes but p, given that their behavior is independent of p's. However, p itself will be CPU-bound, and contribute a load of 1 during the next T seconds, irrespective of its offered load in the past. p's past load is thus an underestimate of its future load and so N_R will underestimate the total load. The system actually uses a *corrected* estimate

$$N'_R = N_R + 1 - n(p)$$

where $n(p)$ is the amount of load offered by p in the past. At any given moment, $n(p)$ is either 0 (not runnable) or 1 (runnable); it is sampled from **/proc/*pid*/stat** and smoothed in a manner similar to N_R. Thus my final predictor for a process p's CPU share relies on two measurements: the overall system load N_R and as p's past load $n(p)$.

5.5.2 Memory

Physical memory on Linux and many other OSes is not a hard constraint: applications can use more virtual memory than the available physical memory. However, if the combined *working set* [22] of all applications exceeds the available physical memory, then the system thrashes. Thus, I can define the "supply" of memory to an application as the amount of memory it can use without the system thrashing (Section 2.3.3).

There are three kinds of virtual memory pages that an application can safely use:

- pages already owned by the application: its *resident set*
- unmapped pages from the *free pool*
- pages owned by some other process, but not in that process's working set: *inactive pages*

Thus, the total size of an application's working set should be limited to the sum of resident size, free pool size and number of inactive pages. The resident set and free pool sizes are straightforward to measure; how do we estimate the number of inactive pages in the system?

The Linux 2.4 VM system uses *page aging*, an LFU scheme for page replacement. The least frequently used pages are placed on an "inactive list": these are assumed to be the pages least likely to be accessed in the near future. Victims for eviction are chosen from this list. The list is divided into clean and dirty pages, depending on whether the pages have data to be flushed to disk. The number of inactive clean and inactive dirty pages can be read from **/proc/meminfo**.

It would seem that that the "inactive pages" statistic is exactly the one we need: it tells us how many pages can be reclaimed with a low probability that they will be accessed again. Unfortunately, Linux's notion of inactivity is *relative*: membership of the list indicates only that a page is less active than other pages in the system, not that it is inactive in any absolute sense. When there is little memory pressure, page aging happens relatively slowly, and only pages that have not been accessed for a long time will make the inactive list. When there is demand for memory, the kernel ages more aggressively, and the threshold for inactivity is lowered.

Thus, the "inactive pages" statistic might over- or under- estimate the actual number of inactive pages in the system. Further, some of these inactive pages belong to the process for which we are determining memory supply: adding together the resident set size and the inactive page count will double-count these pages. To compensate for these errors, I use a control-feedback approach. At each decision point, the system increases its estimate by the amount of available memory; if there is swapping activity, it infers that the application is using too much memory, and reduces its estimate:

$$S_{memory} \leftarrow S'_{memory} + F + I_C - P$$

Here S_{memory} and S'_{memory} are the new and old memory supply estimates (measured in pages) respectively; F is the number of free pages; I_C is the number of inactive clean pages; and P is the system paging activity during the last operation.

$$P = \min(P_{swapin}, P_{swapout})$$

where P_{swapin} and $P_{swapout}$ are the number of pages swapped in from, and swapped out to, disk. The intuition is that we need only worry about pages that are swapped in as well as out: these are potential indicators of thrashing.

5.5.3 Latency

The generic latency predictor computes latency as a linear additive cost metric (Section 2.4.1). It can be used for any operation which spends all its elapsed time using either the CPU (either local or remote) or the network: i.e., it does not overlap computation and I/O, or use multiple CPU's in parallel.

The predicted latency is

$$L = \frac{D_{cpu}}{S_{cpu}} + \frac{D_{remote\ cpu}}{S_{remote\ cpu}} + \frac{D_{xmit}}{S_{xmit}} + \frac{D_{recv}}{S_{recv}} + \frac{D_{rtt}}{S_{rtt}}$$

Here S_{cpu} and $S_{remote\ cpu}$ are the available CPU supply in cycles/s, returned by the CPU supply predictors running on the local and remote machines respectively. S_{xmit} and S_{recv} are the transmit and receive bandwidth measured by the network predictor; S_{rtt} is the inverse of the network round-trip time, i.e. the number of round-trips per second. Resource demand (D_{cpu}, D_{xmit}, etc.) is obtained by invoking application-specific demand predictors. Here D_{rtt} is the number of round-trips (i.e. server RPCs) made during the operation.

5.5.4 Other predictors

Network

For network prediction, I use the existing Odyssey network monitoring infrastructure [71]. Odyssey passively monitors traffic on each connection, and estimates its bandwidth and round-trip time. The monitor makes two assumptions:

- the bandwidth bottleneck is at the first hop, which is shared by all connections: this is a reasonable assumption for mobile hosts with a wireless first hop.
- the bottleneck link is symmetric, with equal the transmit and receive bandwidths.

Energy

The energy predictor monitors battery charge levels and power state, and estimates the remaining battery lifetime. I use the battery monitoring infrastructure developed by Flinn [32], which queries the hardware using the ACPI [49] interface.

File cache predictor

I use the Coda file system [55] to store data shared by clients and servers, and to manage data consistency. The file cache predictor [32] queries Coda to get the file cache supply, i.e., the list of valid cache entries.

5.6 Demand monitors

At the end of each operation, the system collects information about the operation's resource consumption and performance. This is done by a set of demand monitors, one per resource or performance metric. We observed that supply predictors and demand monitors for a given resource use the same mechanisms: in my design, I combine them into a single module. Each resource is handled by a single code component that is responsible for measuring both supply and demand.

Currently, my prototype monitors

- the number of CPU cycles (both local and remote) used by the operation, from **/proc**. Actually, **/proc** provides the CPU usage in seconds; I scale this by the known processor speed in MHz (from **/proc/cpuinfo**) to obtain millions-of-cycles.
- the number of network bytes transmitted and received, and the number of RPC's (round trips) made.
- the operation's working set size: I describe this measurement in detail below.
- the energy consumption, if the hardware supports ACPI, or if we are using the PowerScope energy profiling infrastructure [31].
- the Coda files accessed by the operation, by querying Coda.
- the operation latency, by calling *gettimeofday*.

5.6.1 Measuring an operation's working set size

How do we measure the working set size of an operation? I assume that the process's resident set size is an upper bound on its working set size: this is true if there is no memory contention, i.e., the process's pages are never evicted during the operation. I approximate the working set size by measuring the resident set size at the end of each operation. On Linux, memory allocation does not always create new resident pages, nor does deallocation always reduce the resident set size: the process's heap may contain resident but unallocated pages. This means the memory demand measurement is only valid for the first operation executed by the application process: when we generate logs of memory demand, we must restart the application for each operation.

Thus, memory demand measurements are only valid under controlled experimental conditions: in Chapter 7, I will show how I generate logs of memory demand under such conditions, and use them to generate predictors that can be used even when those conditions are no longer true.

5.7 The logger

The logger assembles all the information about the operation — the failure code passed in to *end_fidelity_op*, the tunable and nontunable parameter values, the resource consumption and latency — and writes it out as a timestamped log entry. I use a text format that is meant to be human-readable, and also easily parseable by scripts: a series of white-space separated terms of the form $\langle variable \rangle = \langle value \rangle$. A sample log file entry is shown in Figure 5.3.

```
...
995296364.088980 [radiator:radiosity] opid=0 dataname="dragon_lighted"
polygons=108590.000000 algorithm="progressive" resolution=0.078310
failed=0 energy:pid=16747.000000 energy:start_ts=995296354.075474
energy:stop_ts=995296364.086669 cpu=2014.159510 child_cpu=2014.159510
memory=56221696.000000 latency=9.935351 net:xmit=0.000000
net:recv=0.000000 net:roundtrips=0.000000 remote_cpu=0.000000
remote_child_cpu=0.000000 remote_coda:files= predicted_cpu=1782.002520
predicted_memory=56652476.388394 predicted_latency=8.559642
constraint_latency=10.000000 constraint_memory=61245030.400000
...
```

Figure 5.3: A log entry for the “radiator” program

5.8 Remote execution

Remote servers are accessed through the Spectra [33] remote execution engine, which is integrated into the multi-fidelity runtime. These servers run application-specific services: Spectra keeps track of available servers and the services provided by them. It tracks bandwidth and round-trip time to each server, and communicates with CPU and memory predictors running on the server to obtain estimates of CPU and memory supply. Spectra also provides an RPC-like interface for applications to communicate with the servers. It tracks the traffic over each RPC connection, and reports it to the network demand monitor.

When a fidelity decision is to be made, the solver obtains a list of candidate servers from Spectra, as well as the corresponding resource supply values. It chooses the server that will maximize operation utility, and uses that server for all remote executions requested by that operation. Thus, server discovery and selection are managed by the runtime system and are completely transparent to the application.

5.9 Overheads

In order to measure the runtime overhead of the various components, I created a “null” operation: one that uses the multi-fidelity API but does no computation. For the null operation, overhead is just the elapsed time from the invocation of *begin_fidelity_op* to the return from *end_fidelity_op*. I computed the per-component cost by selectively enabling components and comparing the resulting overhead to the baseline. Since a single null operation takes less than 1 ms, I ran 1000 consecutive operations during each trial, and divided by 1000 to get the per-operation elapsed time. Each overhead number is the mean of 100 such trials. I also report the standard deviations, which were computed similarly: I computed the standard deviation over the 100 trials, each of 1000 operations, and divided by $\sqrt{1000}$ to get the per-operation standard deviation.

Component(s)	Overhead		
	Total		Component
(Baseline)	0.72 ms	(0.36 ms)	
Logging	0.87 ms	(0.29 ms)	0.15 ms
Logging + *Synchronous flush to kernel*	0.92 ms	(0.40 ms)	0.05 ms
Local CPU monitor	2.10 ms	(0.84 ms)	1.38 ms
Local memory monitor	8.82 ms	(0.53 ms)	6.72 ms
Loc. CPU mon. + *Solver*	12.66 ms	(1.50 ms)	10.56 ms
Loc. CPU mon. + Solv. + *Generic latency pred.*	20.66 ms	(0.87 ms)	8.01 ms

The table shows microbenchmark-based overheads for various runtime component. The first column shows what components were active in addition to the baseline overhead of two calls to the API. The second column shows the mean per-operation overhead (standard deviations in parentheses) computed over 100 trials of 1000 operations each. The third column shows the additional overhead imposed by the highlighted component alone.

Figure 5.4: Per-operation overhead of runtime system

Figure 5.4 shows the component overheads measured on an IBM ThinkPad 560 with a 233 MHz Mobile Pentium MMX processor on a Linux 2.4.2 kernel. The baseline overhead — making two calls to the runtime system — is primarily due to the communication overhead of the Odyssey infrastructure: each API call is converted to a system call, which enters the kernel and is then redirected to a user-level process; the return path also goes through the kernel. There is a small cost for logging each operation to a disk file, and a small additional cost for synchronously flushing the log from user space to the kernel after each operation. This ensures that all completed operations are logged even if the user-level server crashes. Note that this does not protect against OS crashes, since the OS kernel still buffers disk writes: synchronous writing to disk on each operation would impose an even larger overhead.

The CPU and memory monitors must open and read files in **/proc** at the beginning and end of each operation, in order to compute the resource demand at the end of the operation. These file operations are the main source of overhead in the resource monitors.

To measure the overhead added by the solver, I added a single continuous-valued tunable parameter to the null operation: the applications studied in this dissertation have at most one continuous-valued tunable parameter. I also added application-specific CPU and latency predictors. Since the tunable parameter is real-valued, the measured overhead is really that of the iterative gradient-descent algorithm. Adding discrete parameters would multiply this overhead by the number of discrete value combinations, since the solver would check each of them separately. Finally, the generic predictor checks local CPU, network, and remote CPU availability: thus it has an additional overhead compared to the application-specific predictor, which “knows”, for example, that the operation only uses the local CPU resource.

In the worst case, the system could have an overhead of about 27 ms: adding the overhead of the last experiment (local CPU monitor, solver, and generic latency predictor) to that of the local memory monitor. Although a total overhead in the tens of milliseconds seems high, I found it acceptable for the applications studied in this dissertation. All these applications have operation latencies on the order of 1 s or more, and the latency gains due to multi-fidelity adaptation outweigh the costs by far. There are two main ways to reduce the overhead further:

- more efficient implementations of the application-system interface and the solver.
- custom interfaces for efficiently reading kernel resource statistics when measuring operation resource demand for the logs. My gray-box approach restricts the system to the standard system interface (**/proc**): I have made a conscious decision to accept this limitation rather than sacrifice deployability.

5.10 Summary

In this chapter I described the design and implementation of a runtime system to support the multi-fidelity API. I use a minimalist, gray-box approach that avoids kernel modification but sometimes relies on knowledge of kernel internals. I presented the top-level design as well and then described the principal system components: the solver, the utility functions, and predictors for resource supply, resource demand, and performance. I described in detail the CPU, memory, and latency monitors, which were built as part of this thesis work; we briefly described the remote execution engine and the network, energy and file cache monitors, which were built by other researchers. Finally, I showed that the total runtime overhead per operation can range from under 1 ms to almost 30 ms, depending on which components are used. The primary source of overhead is reading kernel statistics from the **/proc** file system.

Chapter 6

Machine Learning for Demand Prediction

Thou art beside thyself; much learning doth make thee mad.

(ACTS *26:24)*

Resource prediction is crucial for adaptation. In Chapter 5, I described how the runtime system predicts resource supply. Here I address the complementary problem: predicting an operation's resource demand as a function of its tunable and nontunable parameters. Resource demand predictors are application-specific: here I describe the basic techniques that we use to construct them. Chapter 7 describes the application-specific resource demand predictors that we built using these methods.

To build a resource demand predictor, I observe resource demand at different points in the parameter space; I append this information to a *history log*; and I fit the log data to some parametric model using machine learning. The aim here is not to develop sophisticated machine learning algorithms, but to show the feasibility of history-based prediction. I use simple learning techniques, refining them only when necessary. This chapter describes the learning techniques used in this dissertation: *least-squares estimation*, *online update*, *data-specific learning*, and *binning*.

I start with an outline of the steps involved in constructing a resource demand predictor from an application log. I then describe each learning technique in detail. Finally, I describe how I evaluate predictor accuracy, by measuring *common-case error* and *bad prediction frequency*.

6.1 How to build a predictor

Using machine learning to build predictors consists of four steps:

1. Select the *input* and *output features*. The output feature is the value we wish to predict. The input features are those tunable and nontunable parameters that might affect the output. E.g. if we wish to predict the CPU demand of a rendering operation, we would expect that it will depend on the number of polygons to be rendered. If the application can scale the number of polygons before rendering, then the scaling factor will also have an impact on the CPU demand. In this case, the polygon count and the scaling factor are the inputs; CPU demand is the output.
2. Select a *parametrized model* that represents the relationship between the inputs and the output. Clearly, a thorough understanding of the computation being measured and the multi-fidelity algorithms in it will greatly aid the selection of a good model. However, I have found that it is possible to build good predictors even with a small amount of domain knowledge and simple learning models.
3. Extract the input and output features from the application log, and find the optimal model parameter values: those values that minimize the overall *prediction error*.
4. If required, add an update method: a way to recompute the optimal parameter values when new measurements are obtained. This allows the predictor to track changes in environmental conditions or user behaviour over time.

6.2 Least-squares regression

In all the applications I studied, one basic technique — least-squares estimation [42, 98] — proved to be of great value. Often, the output is some linear function of the input features; in other cases, we can transform the input features in some way, and then apply a linear function.

The least-squares method fits data to a linear model of the form

$$y = c_0 + c_1 x_1 + c_2 x_2 + \dots c_n x_n = \vec{X}^T \cdot \vec{C}$$

where $\vec{X}^T$ is the transpose of the input feature vector:

$$\vec{X}^T = [1, x_1, x_2, \dots, x_n]$$

and $\vec{C}$ is the coefficient vector

$$\vec{C} = [c_0, c_1, \dots, c_n]^T$$

Given a set of data points

$$\langle y_1, \vec{X}_1 \rangle, \langle y_2, \vec{X}_2 \rangle, \dots, \langle y_k, \vec{X}_k \rangle$$

we find the value of $\vec{C}$ that minimizes the *sum-of-squares error*

$$E = \sum_{i=1}^{k} (y_i - \vec{C} \cdot \vec{X}_i)^2$$

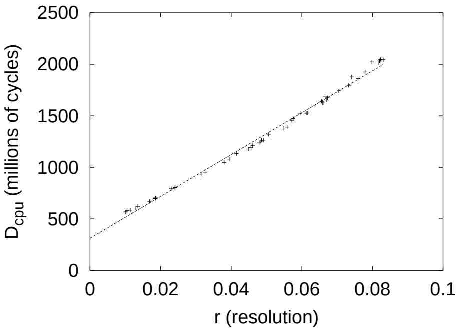


Figure 6.1: Least-squares linear fit for CPU demand in Radiator

by solving

$$\widehat{M} \cdot \vec{C} = \vec{Y}$$

where the matrix $\widehat{M}$ and the vector $\vec{Y}$ are given by

$$\widehat{M} = \sum_{i=1}^{k} \vec{X}_i \vec{X}_i^T$$

$$\vec{Y} = \sum_{i=1}^{k} y_i \vec{X}_i$$

(Each $\vec{X}_i^T$ is the transpose of the corresponding $\vec{X}_i$.)

My code base includes scripts that, given an application log and the names of the input and output features, will compute the regression coefficients.

For example, the "progressive radiosity" operation in the *Radiator* application has a CPU demand that can be modelled as

$$D_{cpu} = c_0 + c_1 p + c_2 pr$$

Here p is a nontunable parameter (number of input polygons) and r is a tunable parameter (scaling factor or resolution). To build a CPU demand predictor for Radiosity, I transformed the input feature set from $[p, r]$ to $[p, pr]$, and then used least-squares estimation to find the optimal values c_0, c_1, c_2.

If we want to build a predictor for a specific data object, then p is fixed, and we can combine the terms $c_0 + c_1 p$ into a single constant. In this case we can use a simple model with only 2 coefficients:

$$D_{cpu} = c_0 + c_1 r$$

Figure 6.1 shows graphically how I fit the latter model to the CPU demand of Radiator, for a specific scene ("dragon") containing 100K polygons. The points represent measurements of D_{cpu} and r obtained by running the program with different values of r. The line represents the least-squares regression fit to these points, i.e. it is the line $y = c_0 + c_1 x$. (In this case, $c_0 = 311.97$ and $c_1 = 20299.08$).

6.3 Online updates

Some application runtime parameters are *invisible*: i.e. the predictor is unaware of these parameters. Consider a user browsing a virtual art gallery, who moves from paintings by Renoir to sketches by Picasso. The JPEG-compressibility of the images changes, and with it the network and energy demand of fetching them; but the application and system are unaware of these changes. In other cases, the predictor is aware of some runtime parameters, but does not know how they impact resource demand. E.g., in a rendering application, CPU demand varies with the camera position and orientation, but we do not know the relationship between camera parameters and CPU demand: the camera parameters are as good as invisible, since we do not know how to make use of them.

When there are invisible parameters, the predictor must use an *incomplete* model: one that maps visible parameters to resource demand, but ignores invisible ones. When the value of an invisible parameter changes, so does the resource demand. From the predictor's point of view, it appears that the mapping from visible parameters to resource demand has changed: the predictor must update itself in order to reflect the new mapping. How do we perform this update without knowledge of the invisible parameters?

In most cases, we can make use of a common property of dynamic systems: *temporal locality*. We can assume that the invisible parameters change only by small amounts, or with low probability, over short intervals of time. This means that the mapping from visible parameters to resource demand also changes gradually. Recent measurements of visible parameters and resource demand will reflect the current mapping more accurately than older data.

To make use of temporal locality, we need predictors that gradually forget the past as they acquire new data: i.e., we need online update methods that give greater weight to recent data. Such methods allow us to track changes in application behaviour over time, without knowing the underlying causes for these changes.

6.3.1 Recursive Least Squares

For linear predictors generated by least-squares estimation, there is a well-known online update method: *exponentially weighted recursive least-squares estimation* [98, pp. 60–65], which we will refer to as RLS. RLS is a simple modification to least-squares estimation that gives greater weight to recent data. It is iterative: the estimation coefficients can be recomputed cheaply whenever we acquire a new data point.

RLS uses the *weighted* sum-of-squares error metric

$$E = \sum_{i=1}^{k} (y_i - \vec{C} \cdot \vec{X}_i)^2 \alpha^{k-i}$$

where α ($0 < \alpha < 1$) is the exponential decay constant. Intuitively, each new data point

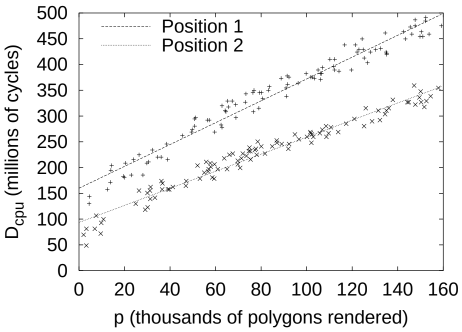


Figure 6.2: Change in linear fit with camera position in GLVU

causes the importance of every previous data point to decay by a factor α. Smaller values of α cause the past to be forgotten more rapidly.

To find $\vec{C}$, we solve

$$\widehat{M} \cdot \vec{C} = \vec{Y}$$

where

$$\widehat{M} = \sum_{i=1}^{m} \vec{X}_i \vec{X}_i^{\,T} \alpha^{k-i}$$

$$\vec{Y} = \sum_{i=1}^{m} y_i \vec{X}_i \alpha^{k-i}$$

Whenever we receive a new data point $\langle y_{j+1}, \vec{X}_{k+1} \rangle$, we update

$$\widehat{M} \leftarrow \alpha \widehat{M} + \vec{X}_{k+1} \vec{X}_{k+1}^T$$

$$\vec{Y} \leftarrow \alpha \vec{Y} + y_{k+1} \vec{X}_{k+1}$$

and recompute $\vec{C}$ before we make another prediction.

RLS requires us to maintain state of size $O(n^2)$, and the complexity of solving the matrix equation on each update is $O(n^3)$, where n is the number of coefficients. However, the costs of state maintenance and incremental update are *independent of the number of data points*: thus, RLS is extremely efficient when we have a linear model with a small number of coefficients.

For example, the CPU demand of rendering a scene in the virtual walkthrough application GLVU is linear in the number of polygons used. We can use

$$D_{cpu} = c_0 + c_1 p$$

where p is the number of polygons that we scale the scene to. However, the value of $\vec{C} = [c_0, c_1]^T$ also depends on the camera position and orientation. I.e., as the camera moves, the relationship between CPU demand and polygon count also changes: when we are looking at a complex portion of the scene, D_{cpu} will be higher (for the same value of p) than when we are looking at a portion of lower complexity. Figure 6.2 shows the CPU demand of GLVU as a function of polygon count, at two different camera positions. We see that each position has a different best fit line, i.e., different coefficient values c_0, and c_1. As the camera moves from position 1 to position 2, RLS will move the best fit line from the first line to the second.

A hybrid approach

An online method such as RLS customizes the predictive model to the recent behaviour of the application. Sometimes, the recent past might be misleading and cause prediction anomalies. E.g., in the short term, even if fidelity is decreasing, resource demand might be increasing for other reasons. RLS cannot distinguish such *temporal* variation in resource demand (e.g., because of changing camera position) from *parametric* variation (due to changing fidelity). Thus, it will conclude that resource demand is a decreasing function of fidelity; the solver will then attempt to maximize utility by picking the highest possible fidelity, leading to a large latency for that operation. These anomalies are rare and are immediately corrected by RLS; however, they do cause annoying performance glitches when they occur.

A static predictor, since it incorporates a wide range of input data, workload scenarios, and fidelities, is robust against such anomalies, but has bad accuracy in the common case: it does not customize its behaviour to the current situation. I combine the advantages of the static and dynamic methods by using a *hybrid* RLS approach. I use a modified update:

$$\widehat{M} \leftarrow \frac{\alpha\widehat{M} + \beta\widehat{M}_0 + \vec{X}_{k+1}\vec{X}_{k+1}^T}{\alpha + \beta + 1}$$

$$\vec{Y} \leftarrow \frac{\alpha\vec{Y} + \beta\vec{Y}_0 + y_{k+1}\vec{X}_{k+1}}{\alpha + \beta + 1}$$

Here $\widehat{M_0}$ and $\vec{Y_0}$ are the values for the offline least-squares fit: we periodically "re-inject" them into the online predictor, to exponential decay from forgetting the offline history altogether. In other words, we now have a history-based predictor where

- the most recent observation is given a weight of 1.
- recent (exponentially-decayed) history is given a collective weight of α.
- offline measurements are given a collective weight of β.

Such a predictor avoids the unstable worst-case behaviour without reducing agility in the common case. I believe that merely increasing the value of α will not provide this benefit, but only increase stability at the cost of agility. However, I have not done a detailed analysis to validate this claim.

In practice, β should be small: the offline data should provide stability against rare anomalies, but not interfere with prediction accuracy in the common case. For the virtual walkthrough application, I use $\alpha = 0.5$ and $\beta = 0.05$. I have found that these values work well in practice, although I have not done a sensitivity analysis on these parameters.

6.3.2 Gradient Descent

Incremental gradient descent [63, pp. 89–94] is an online update algorithm for linear models. It is a cheaper but less robust way to implement the same functionality as RLS. Given a linear model

$$y = \vec{X}^T \cdot \vec{C}$$

we update $\vec{C}$ for a new data point $\langle y_{k+1}, \vec{X}_{k+1} \rangle$ with

$$\vec{C} \leftarrow \vec{C} + \eta \cdot (y_k - \vec{C} \cdot \vec{X}^T_{k+1})\vec{X}^T_{k+1}$$

Intuitively, we move $\vec{C}$ in the direction of steepest gradient wrt y, by an amount proportional to the error observed on the latest data point. I.e., we want to reduce the error with the least possible perturbation of $\vec{C}$. The learning rate η determines how aggressively we compensate for this error: higher values of η cause us to forget the past more rapidly.

Although this algorithm is more efficient — $O(n)$ in both time and space — than RLS, it is not as robust. Its accuracy depends critically on all input features x_i being on the same scale: if one x_i is much larger than the others, the corresponding c_i will see a correspondingly large correction. The net effect will be an oscillation in the value of this c_i, and little change in the other components of $\vec{C}$.

For the case studies in this dissertation, I use linear models with a small number of input features: I use RLS for online updates because the extra overhead is negligible and justified by the added robustness.

6.3.3 Online learning as feedback-control

Online learning forms part of a control-feedback loop: it modifies the predictors, which influence the adaptive behaviour of the system, which in turn determines what new data points will be fed to the online learning algorithm. Thus, the efficacy of online learning will depend on the nature and the rate of changes in application and user behaviour. It is properly viewed as a fall-back technique, for situations where offline learning is not adequate.

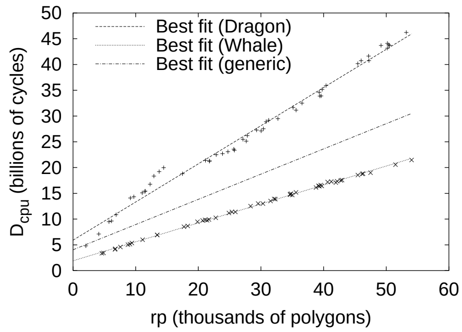


Figure 6.3: Data-specific CPU demand in Radiator

6.4 Data-specific prediction

The resource demand of a computation depends not only on its runtime parameters, but also on its input data. Some features of the input data are easily extracted, and can be included as nontunable parameters to the operation. E.g., input data size is often an important factor in determining resource consumption. However, in the case of complex operations, such as the 3-D graphics computations considered in this thesis, there often are data-dependent effects too complex for us to express or even to understand.

In some applications, the same operation might be executed more than once on the same input data, with different runtime parameter values. Our architect might first request a quick-and-dirty rendering of a scene, and then wish to view the same scene at a higher fidelity. In such cases, we would like our resource predictors to make use of previously acquired, *data-specific* knowledge. In order to do this, I require applications to provide a unique label for its input data: this could be a filename, a hash of the data, or any other uniquely identifying string. (This label is passed to *begin_fidelity_op* as the *dataname* parameter.)

We can now make data-specific predictions by maintaining a cache of previously seen labels: we have a generic predictor computed over all log entries, and also a data-specific predictor for each label seen so far. When we see a new label, we create a new predictor for it, and initialize it with a copy of the generic predictor. Subsequently, only log entries with that particular label will be used to update the data-specific predictor.

Of the applications studied in this dissertation, data-specific prediction is of great value in the two 3-D graphics applications: GLVU and Radiator. The resource demand of both of these depends not only on parameters such as polygon count, but also on the specific 3-D scene being operated on. Often we will perform several operations on a single scene: in this case, we will want to tune our predictor for that particular scene.

For example, the CPU demand of radiosity depends on the resolution r (a tunable parameter) and on the original polygon count p (a nontunable parameter). In fact, CPU demand D_{cpu} is a linear function of the scaled polygon count rp. However, different scenes have different linear relationships between D_{cpu} and rp: i.e. there are data-specific effects that are not captured by the polygon count p. Figure 6.3 shows CPU demand for the "hierarchical radiosity" operation as a function of the scaled polygon count rp for two different scenes. We see that data-specific linear predictors are much more accurate than a generic linear predictor that attempts to fit data from both scenes.

Similarly, when using JPEG-compression on web images, we could use data-specific predictors for the compressibility of each individual image. These could be computed offline: e.g., by compressing the image at different quality levels, measuring the compression ratio in each case, applying a least-squares-based model, and storing the least-squares coefficients along with the image.

In speech recognition also, resource demand depends on the input data, i.e. the utterance being recognized. However, we are extremely unlikely to see exactly the same utterance (i.e. sampled sound waveform) ever again. Thus data-specific prediction is of little use in this application.

6.4.1 Binning

When we have discrete parameters with a small number of value combinations, we can build a predictor that is essentially a lookup table. Each value or combination of values maps to an entry in the table. Each entry would contain a predictor that factored in the remaining input features: those that are continuous-valued, or are discrete but with a large number of possible values.

E.g., the speech recognition application has two discrete-valued features. "Vocabulary size" can be *small* or *large* and "location" can be *local*, *remote*, or *hybrid*. We also have one continuous-valued feature, "utterance length", which ranges from 0 to ∞. My local CPU, remote CPU, and network predictors for this application are all lookup tables with 6 entries each: one for each combination of vocabulary size and location. Each entry is a linear predictor whose single input is utterance length.

Building such predictors is straightforward: we collect logs that cover all the value combinations, which we call "bins". We then separate out the log entries belonging to each bin, and apply other techniques such as least-squares estimation to create a predictor for that bin.

We call this technique "binning"; it is extremely simple, and perhaps does not even qualify as "machine learning". However, it has great practical value: many applications have a small number of discrete-valued features with a small number of possible values. We can view data-specific prediction as a special case of binning, where we do not know all the bins beforehand, but create them dynamically as and when we see new data labels.

	Full vocabulary	Reduced vocabulary
Local	{local:map.dict, local:map.lm}	{local:smallmap.dict, local:smallmap.lm}
Hybrid	{remote:map.dict, remote:map.lm}	{remote:smallmap.dict, remote:smallmap.lm}
Remote	{remote:map.dict, remote:map.lm}	{remote:smallmap.dict, remote:smallmap.lm}

(a) File access predictions

map.dict	55 KB
map.lm	277 KB
smallmap.dict	27 KB
smallmap.lm	224 KB

(b) File sizes

For each combination of vocabulary size and location, the table shows the local and remote files accessed by Janus. The lower table shows the size of each file (the local and remote versions of each file are identical).

Figure 6.4: Binning for file access prediction in Janus

6.4.2 File access prediction

I use a simple extension of the binning method to predict *file cache demand*: the Coda files that will be accessed by an operation. The file cache demand predictor, built by Jason Flinn [32], runs both on clients and on compute servers. It uses binning on all discrete parameters and ignores continuous parameters: typically, it is only discrete parameters that have an effect on which files are accessed, since they change the execution path of the application.

Within each bin, the predictor maintains a list of files, with an associated probability of access. These probabilities are updated after each operation using an exponential decay scheme: if the file was accessed by the last operation, we have a sample value of 1, otherwise 0. Given the access probability for each file and the file cache supply (i.e., which files are cached), the predictor computes find the probability of a cache miss on each file; by combining this with the last-known file sizes, it estimates the *cache miss cost* in terms of bytes fetched from the file server(s). It then queries Coda to find the bandwidth to the file server(s), and computes the expected latency cost of the cache misses.

E.g., the Janus speech predictor has two discrete parameters: *vocabulary size* (either full or reduced) and *location* (either local, hybrid, or remote). Each combination of these parameters causes Janus to read a different dictionary and language model from Coda. The vocabulary size and location determine exactly which files are accessed; these files are read-only, and hence their sizes are also known exactly. These observations gave rise to a very simple binning predictor, shown in Figure 6.4. When the vocabulary size is “full”, Janus accesses the large dictionary and language model (**map.dict** and **map.lm**); when it is “reduced”, it accesses the smaller versions (**maps.dict** and **maps.lm**). When the location is “local”, these files are accessed from Coda at the local client. For “hybrid” and “remote”, the computation is shipped to a remote compute server, and so the files are accessed at the server. Since the vocabulary size and location exactly determine the files accessed, the probability of access is always either 1 or 0. In general, however, file accesses may not be exactly predictable given the parameter values. In these cases, for each bin, we estimate the probability that each file will be accessed, based on log entries belonging to that bin.

6.5 Evaluating prediction accuracy

The error made by a predictor is the difference between the prediction p and the actual observed value x. When p and x are numerical values (e.g. CPU, memory, or network demand), we can represent error by the numerical difference $|p - x|$. Typically, however, we are interested not in the absolute error, but in the relative error

$$\epsilon = \frac{|p - x|}{x}$$

This is the error metric that I use for real-valued predictors throughout this dissertation.

Clearly, the error on a single prediction is not a good indicator of overall predictor accuracy. We must measure the error over a set of samples that covers the space of predictor inputs. We already have such a sample set: the log data used to derive the predictor in the first place. Conventionally, a predictive model is evaluated on a *test set* which is distinct from the *training set* used to generate the model: the two sets are chosen randomly from the total set of observations. This would help us to detect cases where the model performs well on observed data (the training set) but badly on future data (the test set).

In the case of linear regression, there is no need for such separation, provided we have enough data. I.e., a linear model computed from a (large) randomly chosen subset of the data will look very similar to that computed over all the data, and thus will have the same prediction accuracy for future data points. In this thesis, I use linear predictors of very low complexity (2 or 3 parameters) with data sets with 50 or more points, and, for simplicity, I use the entire data set both to derive and to evaluate the predictors.

Given the relative error for each sample, we can compute two aggregate metrics:

- the *bad prediction frequency* (f_{20}): the percentage of samples that had a prediction error above some threshold. I use a threshold of 20%.
- the *common-case error* (E_{90}): an error bound satisfied by the majority (90%) of predictions. I.e., the common-case error is the 90-th percentile value of the error distribution.

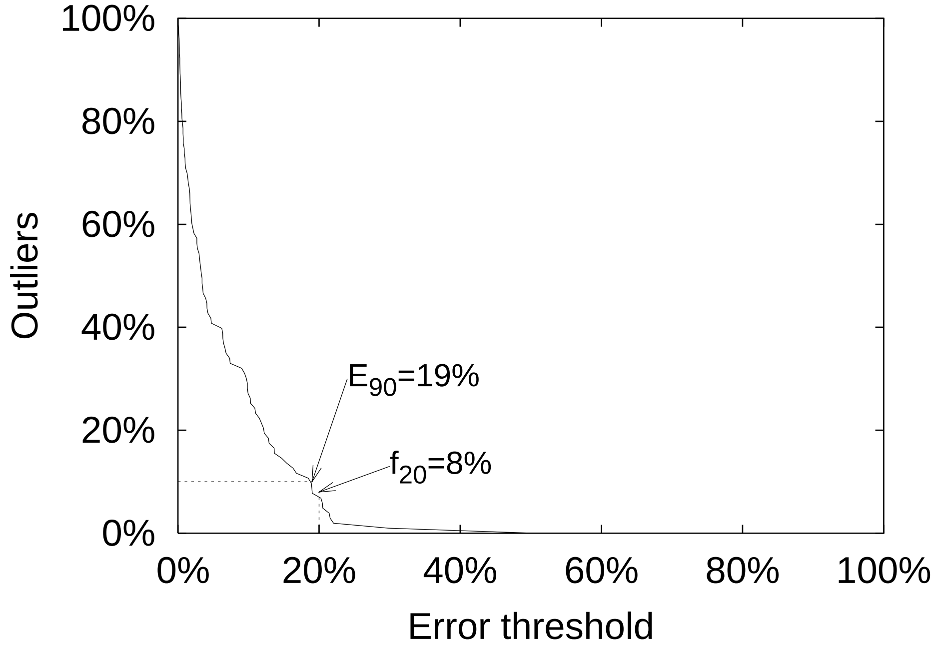


Figure 6.5: Example of outlier distribution plot

Often, however, I will simply show the entire distribution of errors with an *outlier distribution plot*. This plot has error tolerance on the x-axis, and the frequency of outliers — prediction errors that exceed the tolerance — on the y-axis. In other words, it is the cumulative frequency distribution of (relative) prediction errors. Figure 6.5 shows an example of such a plot. On such a plot, f_{20} is the y corresponding to an x of 20%; E_{90} is the x corresponding to a y of 10%. In this example, $f_{20} = 8\%$ and $E_{90} = 19\%$, which means that:

- 92% of the time, we will see less than 20% error
- 90% of the time, we will see less than 19% error

6.6 Summary

This chapter described the techniques I use to build and evaluate application resource demand predictors.We use *least-squares regression*, and its online variant, *recursive least-squares regression*, to model the dependence of resource demand on continuous, real-valued parameters. I use *binning* to model the effect of discrete-valued parameters, effectively by building a separate predictor for each value combination. Similarly, I handle dependence on input data with *data-specific predictors*. The chapter concludes with a discussion on evaluating predictor accuracy. In this dissertation, I use the *outlier distribution* *plot* to represent predictor accuracy. I also use two values derived from this plot: *the bad prediction frequency* f_{20} and the *common-case error* E_{90}.

Chapter 7

Applications

Il n'existe pas de sciences appliquées, mais seulement des applications de la science.
(Louis Pasteur)

In order to validate the multi-fidelity API and implementation, I and other researchers ported a small, selected set of applications to the multi-fidelity API. I chose applications that:

- are realistic, and not toy applications.
- were written by other people, to evaluate the difficulty of porting legacy code to the API.
- are interactive, and would be useful in a mobile/wearable scenario.
- are resource-intensive, requiring adaptation to run with good performance on mobile hardware.
- permit adaptation of fidelity and performance: this rules out applications that cannot tolerate fidelity degradation (e.g. medical imaging) or require performance guarantees (hard real-time applications).
- are compatible with, or easily ported to, the base OS (Linux).

I chose four applications for my case studies:

- GLVU, a "virtual walkthrough" program.
- *Radiator*, a radiosity program
- *Netscape*, a web browser
- *Janus*, a speech recognizer

Web browsing and speech recognition are well-known applications, and have previously been shown to benefit from adaptation [71, 31]. Here I show how the benefits of adaptation can be obtained with a small amount of code modification, and how the effects of adaptation on resource demand can be predicted.

GLVU and Radiator are relatively novel to the mobile computing world: they are 3-D graphics applications, chosen for their relevance to the augmented reality scenario of Chapter 1. Section 7.2 explains briefly why these applications are important, and how *multi-resolution models* provide them with an intrinsic notion of fidelity.

The remainder of this chapter is organized as follows. Section 7.1 broadly describes the methodology and evaluation metrics used in my case studies. Sections 7.3–7.6 are detailed case studies of the four applications, in the order shown above.

7.1 Methodology

In each case study, I describe the application and its core multi-fidelity operation; the operation's tunable and nontunable parameters; the work involved in porting the application to use the multi-fidelity API; the resource demand predictors I constructed for it; and the accuracy of these predictors. The experimental platform in all cases was an IBM ThinkPad 560 with a 233 MHz Mobile Pentium MMX processor and 96 MB of memory, running Linux 2.4. The remainder of this section describes the metrics that I use in each case study to quantify *porting cost* and *prediction accuracy*.

7.1.1 Porting cost

One of the principal aims of the multi-fidelity API is to minimize the cost of porting legacy applications to it. In each of the case studies presented here, I evaluated this porting cost in terms of *number of modified source lines and files*. I derived this number by running the "diff" program to compare every file in the modified source to the corresponding file in the original source. I then quantified the results of this diff in terms of the number of lines and files modified, as described below. I chose the method to err on the side of caution: i.e., always to overestimate rather than underestimate line counts.

For each differing segment, I counted the number of lines in the old and new versions, and took the larger of the two values. The sum of all these numbers is the porting cost. When new files were added, I added their entire line count to the porting cost; when files were deleted, I ignored them. Similarly, modified and added files contribute to the modified file count, but deleted files do not.In some cases, there were changes unrelated to the multi-fidelity API: e.g. bug fixes and addition of trace playback for repeatable experiments. When the diffs corresponding to these changes were clearly identifiable, I excluded them from the line count; otherwise, I included them.

Of course, "lines and files changed" is not the only way to measure the cost of modifying code. However, it is the only metric that was easily reliably measurable in each case study. Further, the small amount of modification required in each case study indicates that other metrics of modification cost (e.g. man-hours) would also be small.

7.1.2 Prediction accuracy

For each application, I also measured the accuracy of history-based demand prediction. Recall from Chapters 4 that these are built by logging application resource demand for various input data and runtime parameter values. I applied the prediction techniques discussed in Chapter 6 — least-squares regression, data-specific prediction, and binning — to these logs. I then compared the values generated by the predictors to the actual logged values, and computed the prediction errors. I report the bad-prediction frequency (f_{20}) and the common-case error (E_{90}); I also show the data graphically as an *outlier distribution plot*. Recall from Section 6.5 that f_{20} is the percentage of predictions that exceed an error threshold of 20%; $E90$ is the highest error observed on the best 90% of the predictions (i.e., the 90th percentile error); and the outlier distribution shows f_e (percentage predictions exceeding an error e) as a function of e.

I evaluated demand predictors for the following resources:

- CPU (both local and remote), measured in millions of cycles. Recall that CPU demand is actually measured in seconds, and then scaled by the processor speed (in MHz). In Section 7.4.4 we will see that although this scaling does not compensate perfectly for all the differences between different processors, it does so much better than a time-based (unscaled) measurement scheme.
- Memory, measured in MB
- Energy, measured in Joules
- Network transmit and receive, measured in bytes

The core computations in these applications are often large and complex. My objective in building and evaluating resource demand predictors was not to understand or capture every detail of these computations, but to extract a small, simple set of features that account for most of the variation in resource demand. I believe that the simple predictors described here show the viability of resource demand prediction: if greater accuracy is required, it can be provided by more sophisticated predictors built by domain experts and machine learning researchers.

7.2 Augmented Reality

Section 1.1 described an architect using augmented reality software to aid in a design task. One key functionality of such software is the overlaying of virtual 3-D objects or scenes on the real world. E.g., in the architect scenario, we might want to virtually add a wall to the interior of a room; this requires the software to overlay the relevant portion of the user's field of view with the image of the wall.

A full-fledged augmented reality application would be extremely complex, and require substantial domain expertise to understand and modify. Further, such applications are hard to find in source-available, Linux-compatible form. I decided instead to experiment with two simpler but representative 3-D graphics applications that implement different pieces of augmented-reality functionality. **GLVU** is a virtual walkthrough program: it allows a user to specify a 3-D scene, and "walk through" the scene: i.e., *render* it from any camera position and orientation. **Radiator** uses *radiosity* [18] to colour and shade a 3-D scene according to the light sources illuminating that scene. We can think of Radiator as a pre-processing step for GLVU: it produces a more realistic looking scene for rendering.

7.2.1 Multiresolution models

The input to both GLVU and Radiator is a *3-D scene* represented as a polygonal surface model: a set of vertices and polygons. A vertex is simply a point in 3-D space; a polygon is an ordered list of vertices. The resource demand of many 3-D graphics applications, including GLVU and Radiator, scales with the number of polygons in the input scene.

The **QSlim** [40] program converts a 3-D surface model into a *multiresolution model*: an annotated version of the original that provides successive approximations by eliminating some polygons and vertices. The degraded versions are similar to the original, but with fewer polygons and corresponding loss of detail. This gives us a natural way to trade fidelity for resource consumption. The fidelity metric is *resolution*: the number of polygons retained in the degraded version, expressed as a fraction of the original polygon count. This number varies from 0 to 1.

QSlim uses a quadric-based simplification algorithm [41] to annotate the 3-D scene with a list of *edge contractions*. Each contraction eliminates two adjacent vertices and replaces them with a single new vertex. It also eliminates any polygons incident on the contracted edge. By applying or undoing these contractions, we can scale the model to any desired resolution. The contractions are ordered in increasing order of error: we first do the contractions which cause the least deviation from the original shape.

I use QSlim-generated multiresolution models with both GLVU and Radiator. For GLVU, we can reduce the resolution of the input scene before the main computation: i.e., resolution is a data fidelity metric. In Radiator, on the other hand, the bulk of the computation is done on the degraded version, but the final step is applied to the original, full-resolution version. I.e., both the input and output of Radiator have resolution 1, but the detail and accuracy of the shading and colouring in the output depend on the resolution chosen for the computation. In this case, resolution is more properly viewed as a computational fidelity metric.

```
description glvu:draw
logfile /usr/odyssey/etc/glvu_draw.log
mode normal
hintfile /usr/odyssey/lib/glvu_draw.so
utility glvu_draw_utility
update glvu_draw_update
init glvu_draw_init

param polygons ordered 0-infinity
fidelity resolution ordered 0-1

hint latency glvu_draw_latency_hint
hint cpu glvu_draw_lcpu_hint
```

Figure 7.1: Application configuration file for GLVU

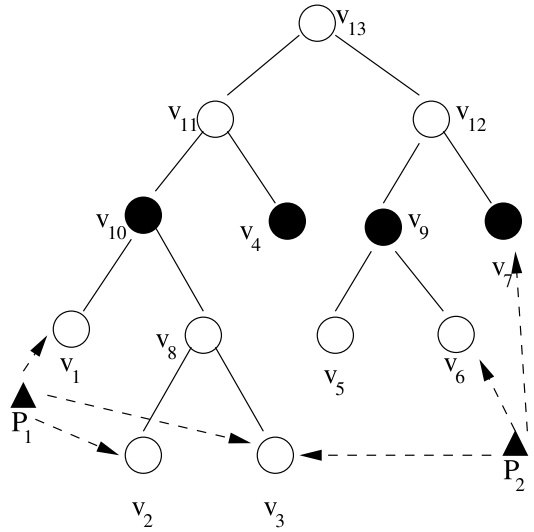
7.3 GLVU

GLVU [75] is a toolkit that allows us to virtually "walk through" any 3-D scene. The core computation performed by GLVU is to *render* a view of the scene based on the current position and orientation of the camera, i.e. the user. Thus, "render" is the multi-fidelity operation for this application: we trade its fidelity against its resource consumption and latency. Rendering is almost entirely processor-bound in the absence of specialized 3-D graphics hardware, so I focus on the CPU resource. Figure 7.1 shows the Application Configuration File for the render operation. It has 1 tunable parameter — the resolution — and 1 nontunable parameter — the scene's original polygon count.

In the remainder of this section, I describe the multiresolution extensions that I added to GLVU, and evaluate the cost of this modification as well as that of inserting multi-fidelity calls into GLVU. I then describe the construction of a CPU demand predictor for the "render" operation: creating logs of resource demand, creating a predictive model with offline learning, and updating the model through online learning. Finally, I evaluate the predictor's accuracy in various scenarios.

7.3.1 Extending GLVU for multiresolution rendering

I extended GLVU's file parser to read multiresolution models, and the "scene" object to support dynamic scaling of polygon count. The vertices of the multiresolution model are stored in a binary tree (Figure 7.2). Whenever an edge contraction collapses two vertices u and v into a new vertex $w = e(u, v)$, we make w the parent of u and v. Each leaf of the tree correspond to a vertex of the original model; each interior vertex corresponds to an edge



The figure shows how vertices of a multiresolution model are organized into a tree. The original vertices of the model are the leaves of the tree, i.e., $v_1 - v_7$. A simplified version is given by a cross-section of the tree, e.g. the vertices v_{10}, v_4, v_9, v_7. The figure also shows two of the model's polygons, P_1 and P_2. In the simplified version, P_1 becomes invisible since two of its vertices (v_2 and v_3) have been collapsed. P_2 is transformed from $[v_3, v_6, v_7]$ to $[v_{10}, v_9, v_7]$.

Figure 7.2: Vertex tree for multiresolution models

contraction.

The set of *visible* vertices at any given resolution is given by a cross-section of the tree (the coloured vertices in Figure 7.2). We apply a contraction by moving up the tree (colouring v_{11} instead of v_{10} and v_4) and marking as invisible the polygons incident on both the contracted vertices: there are at most two such polygons. Reversing these steps undoes the contraction; both contraction and its inverse operation take $O(1)$ time.

In addition to adding the vertex-tree data structure and a visible-polygon bitmap, I modified the core rendering loop to skip over any polygons marked invisible. For each visible polygon, we find its vertices by starting from each original vertex and moving up the tree until we find its unique "visible ancestor". To avoid repeated pointer-chasing, we cache pointers to the visible ancestor of each vertex. These pointers are soft state: if we follow one and do not find a visible vertex, we recompute the pointer.

These data structures and algorithms are designed for efficient contraction and decontraction, especially when the target resolution is close to the current resolution. The assumption is that small changes in fidelity will be frequent and larger ones less so.

Files	Lines	Purpose
1	13	Application Configuration File
1	245	Hint module
1	170	Calls to glue code
2	132	Glue code
5	560	Total modifications for multifidelity
144	26951	Original code base
2	369	Multiresolution extensions

Figure 7.3: Modifications made to GLVU

7.3.2 Porting GLVU to the multi-fidelity API

To make GLVU adaptive, I wrote an Application Configuration File (Figure 7.1), as well as a hint module containing the CPU demand predictor described later in this section. I also modified GLVU's source code to invoke the multi-fidelity API. I did this by writing a *glue code* layer that converts from a GLVU-specific interface to the generic multi-fidelity API; I then inserted calls to the application-specific interface into the source code. The purpose was to minimize changes to existing source files, at the cost of adding two additional files.

Figure 7.3 shows the cost of these modifications: 560 lines in 5 files, about 2.1% of the total code base. I also show the cost of the multiresolution extensions: these changes are independent of the multi-fidelity API but required for adaptation.

7.3.3 Predicting GLVU's CPU demand

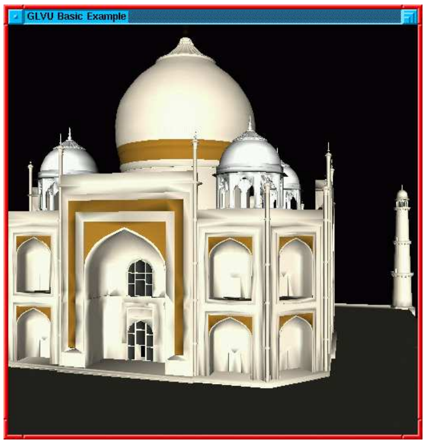
My approach to demand prediction is empirical (Chapters 4 and 6): I measure the application's resource consumption over a variety of tunable and nontunable parameter values, and derive a model that maps parameter values to resource consumption. I generated logs of GLVU's resource consumption at different resolutions for 4 different scenes, ranging in size from 127K to 236K polygons. Figure 7.4 shows the four scenes and their polygon counts. Figure 7.5 shows the visual difference between full resolution and 10% resolution on the "Notre Dame"scene.

For GLVU's CPU demand, I use a linear predictor of the form

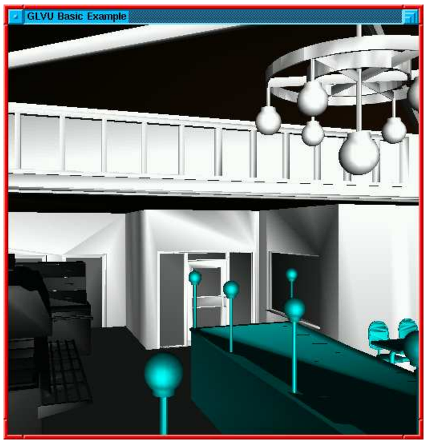
$$D_{cpu} = c_0 + c_1 pr$$

where p is the original polygon count, r is the resolution, and pr is the polygon count after multiresolution scaling.

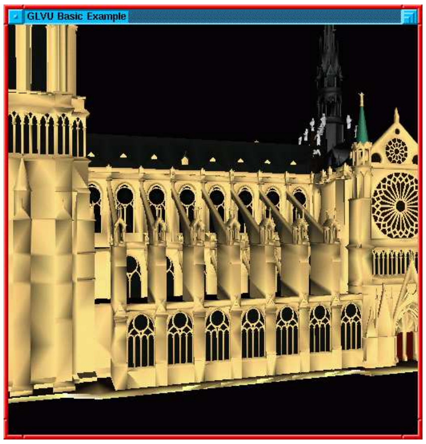
To evaluate this predictor, I collected 100 log entries per scene, each with a randomly chosen resolution and camera position/orientation. Figure 7.6(a) shows the graph of CPU



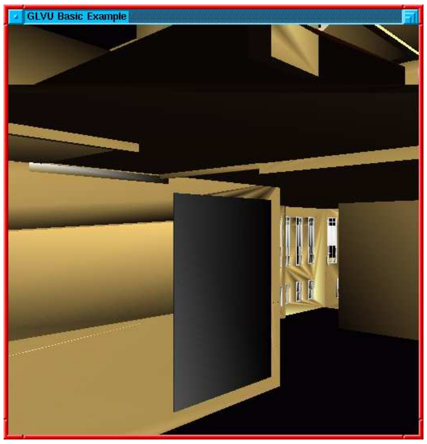
(a) Taj Mahal



(b) Café (interior)



(c) Notre Dame

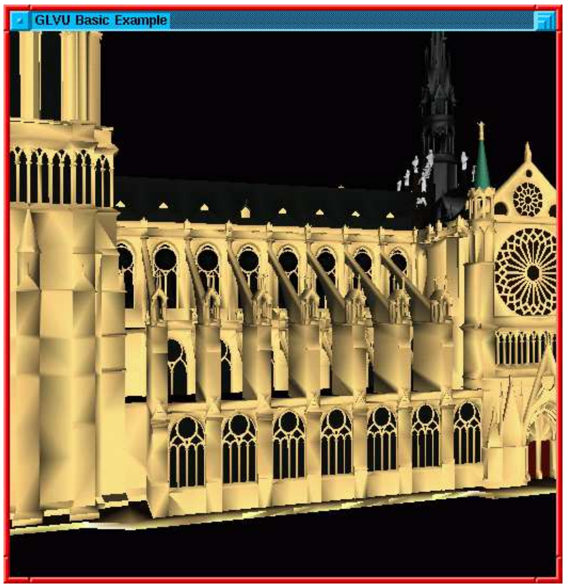


(d) Buckingham Palace (interior)

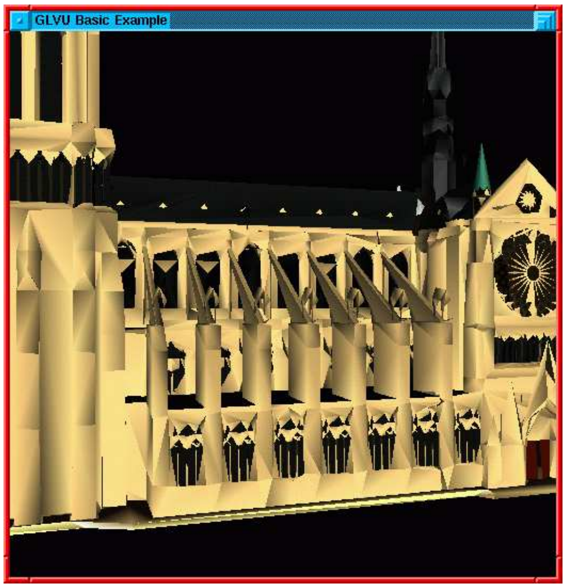
Scene	Number of polygons
Taj Mahal	127406
Café	138598
Notre Dame	160206
Buckingham Palace	235572

(e) Scene polygon counts

Figure 7.4: 3-D scenes used in GLVU

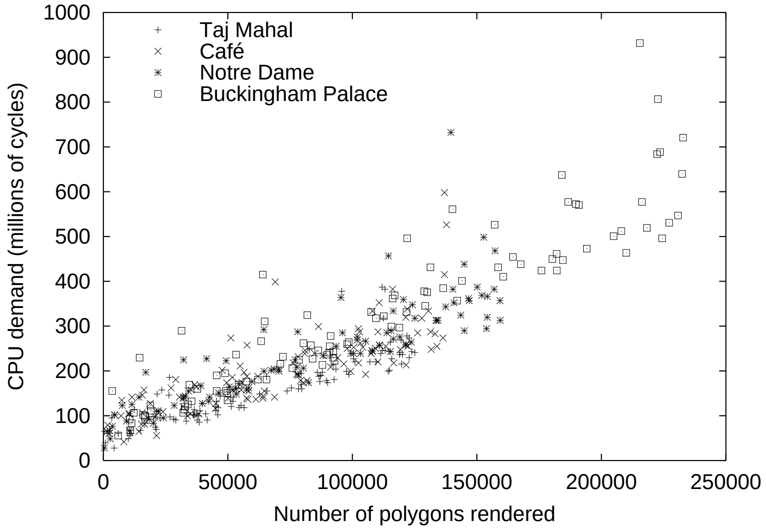


(a) Notre Dame, full resolution

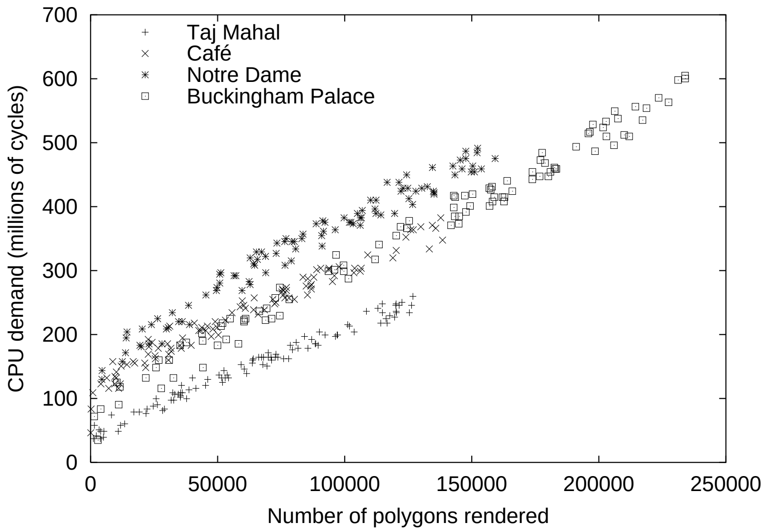


(b) Notre Dame, 0.1 resolution

Figure 7.5: Effect of resolution on output in GLVU



(a) Random camera position



(b) Fixed camera position

The x-axis is the number of polygons rendered, i.e. pr where p is the original model size and the r is the resolution. The y axis is the CPU demand in millions of cycles.

Figure 7.6: GLVU's CPU demand: random (top) and fixed (bottom) camera positions

demand against rendered polygon count for these experiments. We see a rough correlation between rendered polygon count and CPU consumption, but with a large amount of variation. I repeated the experiment, this time with the camera fixed at an arbitrary position for each scene (Figure 7.6(b)). This gave us a strong linear correlation between polygon count and CPU demand, with some amount of noise. However, each scene follows a different line, indicating that we need *data-specific* prediction.

I built both data-specific and data-independent (generic) linear predictors from these logs. Each data-specific predictor was derived through least-squares regression on the log entries from a specific scene; the generic predictor was derived from the union of all the logs. For each scene, I evaluated the accuracy of the data-specific and the generic predictor: I then aggregated the results for the generic predictor to evaluate its accuracy across all 4 scenes.

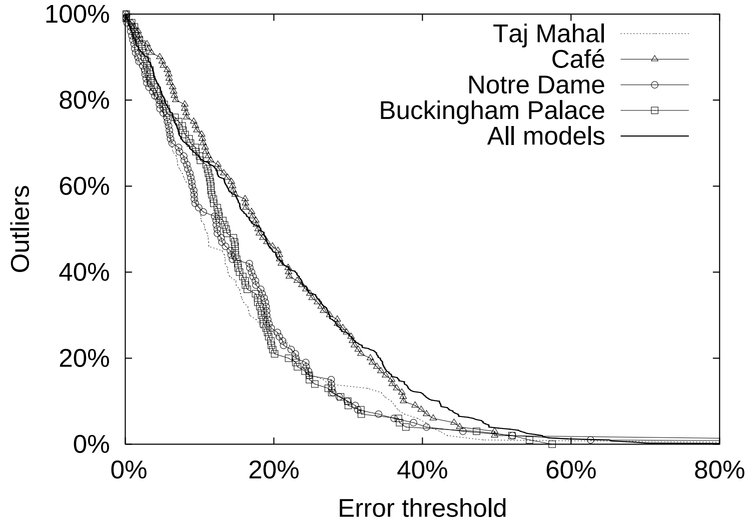
Figures 7.7 and 7.8 show these results. With a fixed camera position, data-specific prediction does very well (the bad prediction frequency is at most 5%), but the generic predictor performs poorly. This is unsurprising: camera position is scene-specific, and fixing the camera position does not reduce the variation across scenes. When the camera position is random, both schemes perform poorly: the variation in camera position injects a large amount of noise into the data. These results show that data-specific predictors offer a clear benefit over generic predictors, but only if we can somehow account for camera movement.

User path traces

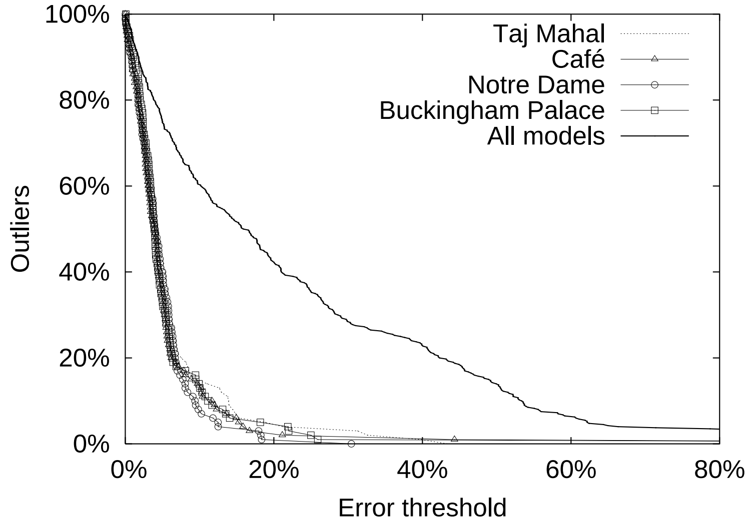
The results just presented show that CPU demand is affected by camera movement. However, neither the "fixed camera" nor the "random camera" scenarios is a realistic model of such movement. Users rarely jump randomly from one position to another; and they do not stay at a fixed position. Rather, they move along some path, causing a continuous variation in camera position and orientation over time.

In order to create repeatable yet realistic workloads, I used GLVU's *path record and playback* functionality: as a user navigates a scene, the software can save the camera position at each step to a disk file. Subsequently, we can replay this trace, creating exactly the same effect as the original user. The trace does not record timing information, however, and thus does not capture "user think time".

I used these traces as *closed-loop workloads*, where each operation starts as soon as the preceding one is completed. Such workloads model an impatient user who is always ahead of the software. I created four such traces of real users: one for each of the 3-D scenes in the case study. I also modified GLVU to automatically replay these traces in batch mode, without requiring user interaction. In the remainder of this dissertation, all experiments with GLVU are based on these traces.



(a) Random camera



(b) Fixed camera

The graphs show the outlier distribution for linear predictors of CPU demand in GLVU, both when the camera is positioned randomly for each render, and when it is fixed. I evaluated a data-specific predictor for each scene, as well as a generic predictor over all four scenes.

Figure 7.7: CPU demand prediction in GLVU: outlier distribution

Scene	Random camera		Fixed camera	
	f_{20}	E_{90}	f_{20}	E_{90}
Taj Mahal	27%	36%	5%	14%
Café	46%	39%	3%	12%
Notre Dame	27%	31%	1%	9%
Buckingham Palace	22%	30%	5%	12%
All scenes	45%	42%	42%	54%

The table shows the bad prediction frequency and common-case error for linear predictors of CPU demand in GLVU, both when the camera is positioned randomly for each render, and when it is fixed. We show the accuracy of a data-specific predictor for each scene, as well as a generic predictor over all four scenes.

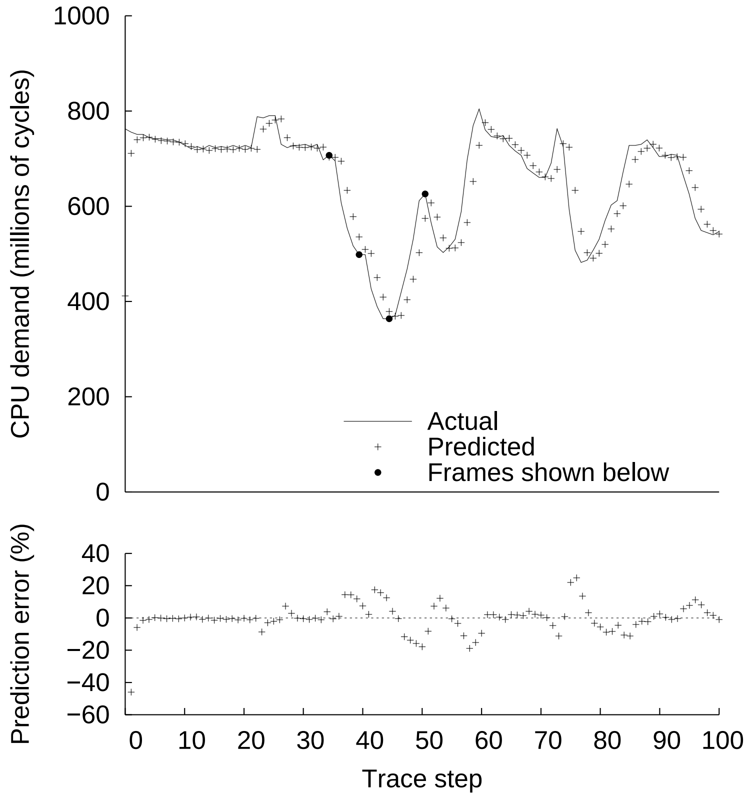
Figure 7.8: CPU demand prediction accuracy in GLVU

Online-update predictor

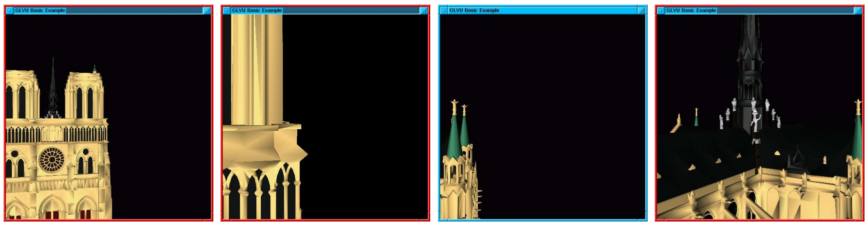
Given that camera position varies over time, and that CPU demand depends on camera position, how do we build a camera-aware predictor? It seems we must learn CPU demand as a function of all 6 camera parameters, in addition to polygon count: 3 parameters for position and 3 for orientation. Other camera parameters — aspect ratio, field-of-view, and near and far clip plane distances — typically do not change over time. However, building such a predictor is difficult, and certainly impossible with a simple linear model. The CPU demand at any particular camera position will depend on the local properties of the scene, and there is no single global model that can capture the CPU demand at all positions. We might try to sample the camera parameter space, and use a *nearest neighbour* [63, pp. 231–236] approach to find the closest matching sample to any given position at runtime. Unfortunately, sampling at a fine granularity in an 6-dimensional space would require a prohibitively large number of samples.

Instead, I adopted a simpler approach. I observed that, in real use, camera position has *temporal locality*: i.e., it changes incrementally over time. If CPU demand has *spatial locality* — a small change in camera parameters causing only a small change in CPU demand — then it should also have *temporal locality* — i.e., vary smoothly over time. This suggests an online-update predictor — one that tracks recent history, and gradually forgets the more remote past.

I implemented a data-specific, online-update prediction scheme for GLVU. At application startup, the demand predictor uses a generic linear estimator derived from the "random camera" data. Whenever a new scene is loaded, it creates a data-specific predictor which is initially identical to the generic predictor. With each subsequent operation on the scene, the data-specific predictor updates itself using a hybrid RLS predictor (Section 6.3.1) with $\alpha = 0.5$ and $\beta = 0.05$. Thus we have specialized the predictor not only to the scene, but also to the camera position within the scene.



(a) CPU demand and prediction error



(b) Views at highlighted time points

The graph shows the CPU demand of rendering at full resolution, for the first 100 steps of a user trace on the "Notre Dame" scene; we also show the % error of the RLS predictor along the same time line. The images correspond to the camera views at the the four highlighted time points are shown below.

Figure 7.9: GLVU's CPU demand: user trace, "Notre Dame" scene

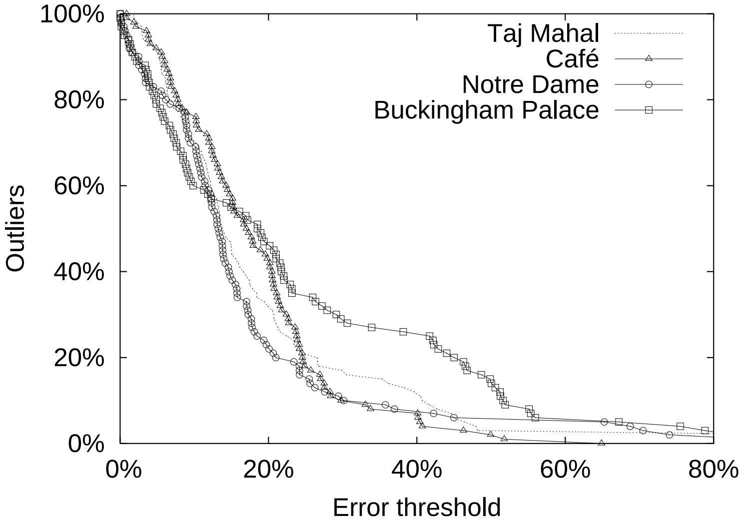
I evaluated this online-update scheme on user path traces for each of the four scenes. I.e., I ran GLVU on these traces, and logged both the actual as well as the predicted CPU demand. Figure 7.9(a) shows how the predictor tracks CPU demand over time. We see that the very first prediction was extremely inaccurate: we started without any scene-specific or camera position specific knowledge. As soon as the predictor acquired feedback from the first operation, it rapidly adjusted itself to track the current scene and camera position, and subsequent predictions were much more accurate. Note that although the CPU demand varies by a factor of 2, prediction error is usually under 20%.

To illustrate the reason for the wide variation in CPU demand, I selected four points in the trace with widely different CPU demand values (the four highlighted points in Figure 7.9(a)). The camera views at these four points are shown in Figure 7.9(b). Intuitively, CPU demand depends on the "complexity" of the visible portion of the scene: however, I know of no quantitative measure for this complexity that could aid in CPU demand prediction.

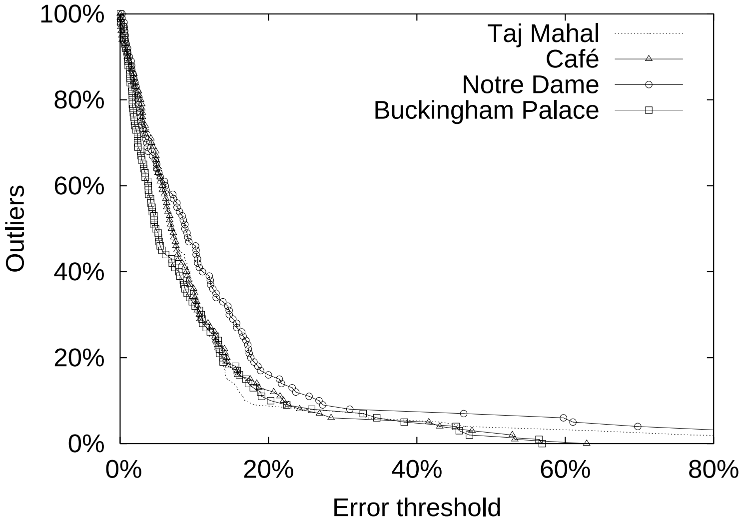
Figures 7.10, 7.11, and 7.12 show the overall prediction accuracy achieved by the data-specific dynamic predictors on the first 100 steps of each of the 4 user traces. I compared these with data-specific, but static, predictors computed using least-squares regression over all log entries for a particular experiment. I ran the experiments with the resolution (fidelity) varying randomly, as well as with resolution fixed at 1. In all cases, the dynamic predictors were substantially more accurate than the static predictors; both static and dynamic predictors did better when resolution is fixed rather than varying randomly. In real use, we would expect the resolution to vary, but not randomly: the system will adapt the resolution to varying CPU load, and we can expect a prediction accuracy somewhere between the "random resolution" and the "fixed resolution" cases.

7.3.4 Summary

I studied GLVU, a virtual walkthrough program whose core task is the rendering of 3-D scenes. I described how I modified this application for multi-fidelity adaptation, and showed that the cost of this modification was reasonable. I observed that this was a CPU-bound operation, and that CPU demand could be adapted by changing the number of polygons rendered. I used a simple linear model to built a CPU demand predictor for GLVU, based on logs from 4 different scenes. I showed that predictor accuracy is significantly improved by *data-specific learning* — customizing the predictor to new scenes as they appear — and *online learning* — tracking variation over time due to changes in camera position. With these improvements, the common-case prediction error varied between 4% and 27%, and the bad prediction frequency between 1% and 16%.



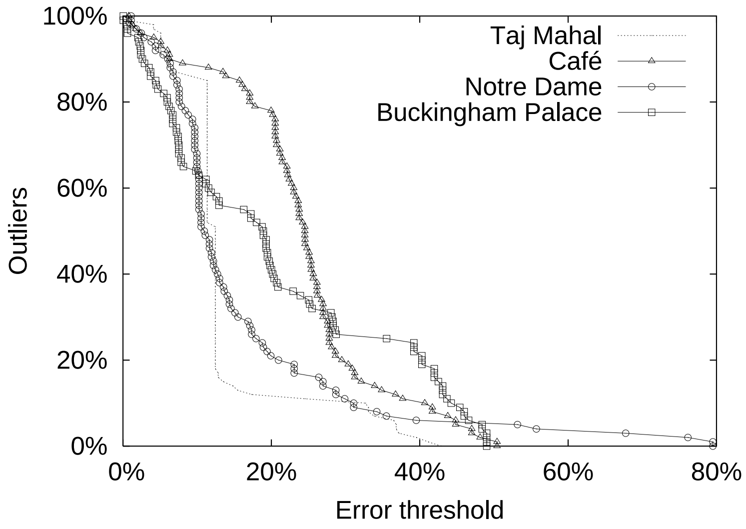
(a) Static predictors



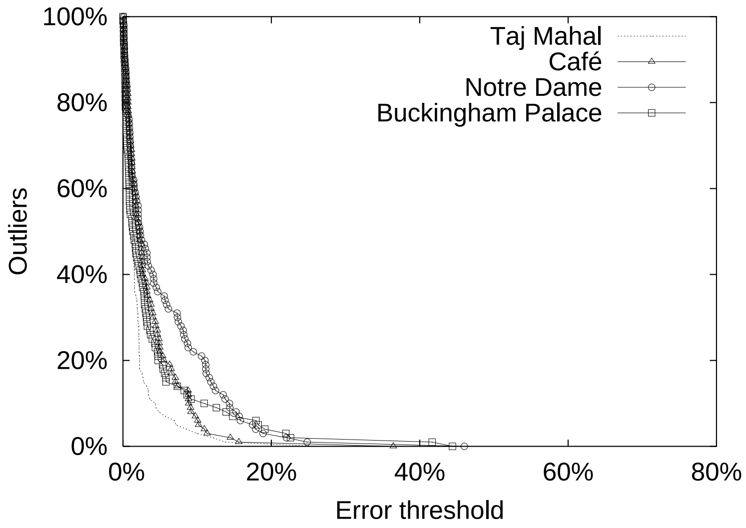
(b) Dynamic predictors

The graphs show CPU demand prediction accuracy for GLVU running user trace workloads: for each scene, the camera moves through the first 100 steps of the corresponding user path trace, and the resolution varies randomly. We compare the static linear predictor with a dynamic RLS-based linear predictor.

Figure 7.10: Static vs. dynamic predictors: random resolution



(a) Static predictors



(b) Dynamic predictors

The graphs show CPU demand prediction accuracy for GLVU running user trace workloads: for each scene, the camera moves through the first 100 steps of the corresponding user path trace, and the resolution is fixed at 1. We compare the static linear predictor with a dynamic RLS-based linear predictor.

Figure 7.11: Static vs. dynamic predictors: fixed resolution

Scene	Static predictors				Dynamic predictors			
	Random r		Fixed r		Random r		Fixed r	
	f_{20}	E_{90}	f_{20}	E_{90}	f_{20}	E_{90}	f_{20}	E_{90}
Taj Mahal	32%	42%	12%	33%	9%	18%	1%	4%
Café	43%	33%	78%	42%	13%	23%	1%	9%
Notre Dame	23%	36%	21%	31%	16%	27%	3%	14%
Buckingham Palace	47%	52%	42%	45%	11%	22%	4%	13%

The table shows CPU demand prediction accuracy for GLVU running user trace workloads: for each scene, the camera moves through the first 100 steps of the corresponding user path trace. We compare the static linear predictor with a dynamic RLS-based linear predictor in two cases: with randomly varying resolution, and resolution fixed at 1.

Figure 7.12: CPU demand prediction accuracy for GLVU running user traces

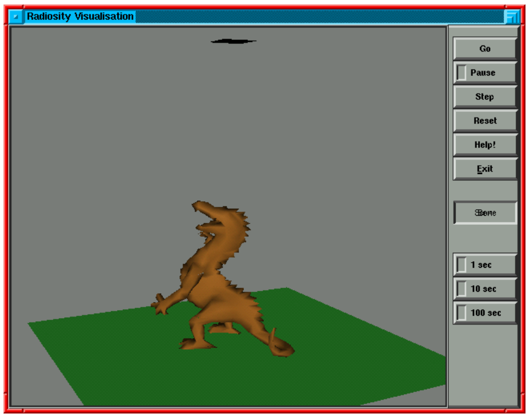
7.4 Radiosity

A *radiosity* [18] computation colours and shades a 3-D scene according to the light sources present in the scene. We would run such a computation on our virtual or augmented reality whenever the scene changes, i.e., when objects or light sources are added, removed, or modified. Radiosity is *view-independent*: it is computed over over the entire scene, and need not be recomputed when the camera position or orientation changes. The output of radiosity has each input polygon annotated with colour and lighting information that make for more realistic rendering. Figure 7.13 shows a scene rendered before and after radiosity, both at low fidelity (0.01) and at high fidelity (0.1) In theory, the fidelity (i.e., the resolution) can be as high as 1, but for a scene with 100k polygons, this is well beyond the resource limits of my test platform.

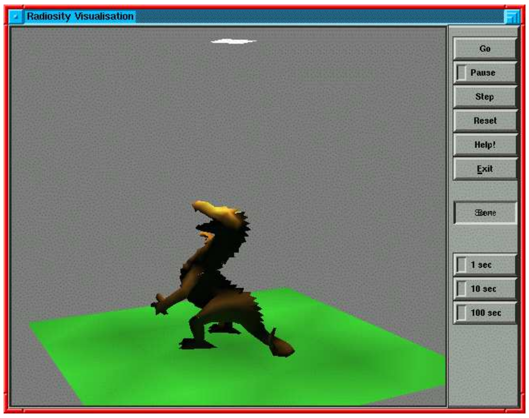
Radiator [96] is a publicly available implementation of several common radiosity algorithms, with built-in support for multiresolution models. It allows the user to load a scene, select a radiosity algorithm and a resolution, and run the algorithm. In this dissertation, I consider two of the most common algorithms: *progressive* and *hierarchical* radiosity. While both algorithms have non-trivial CPU and memory demand, progressive radiosity is more memory-intensive and hierarchical radiosity is more CPU-intensive. Thus the optimal choice of algorithm at runtime will depend on the CPU and memory supply. The output qualities of the two algorithms are comparable, and I assume here that the they are equal at any given resolution: a study of user-perceived quality is beyond the scope of the thesis.

Figure 7.14 shows the Application Configuration File for Radiator. It has two tunable parameters, one discrete — the choice of algorithm — and one continuous — the resolution. There is one nontunable parameter: the input scene's polygon count.

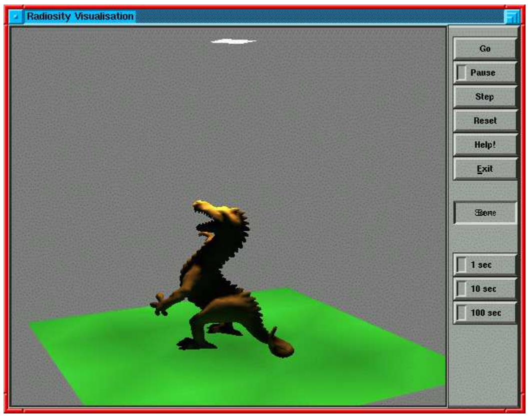
The remainder of this section describes the cost of modifying Radiator to use the multi-fidelity API; the CPU and memory demand predictors for both progressive and hierarchical



(a) Before radiosity



(b) After radiosity (low fidelity)



(c) After radiosity (high fidelity)

This scene contains three elements: a 100K-polygon model of a dragon, the ground, represented by a single green square below the dragon, and a light source vertically above the dragon. The top image shows the image as rendered before a radiosity computation: we see that the lighting has no effect on the observed image. The bottom images show the same image after a radiosity computation at low fidelity (0.01), and at high fidelity (0.1).

Figure 7.13: Effect of radiosity on a 3-D model

```
description radiator:radiosity
logfile /usr/odyssey/etc/radiator.radiosity.log
mode normal
constraint latency 10
param polygons ordered 0-infinity
fidelity algorithm unordered progressive hierarchical
fidelity resolution ordered 0.01-1

hintfile /usr/odyssey/lib/rad_hints.so
hint cpu radiator_radiosity_cpu_hint
hint memory radiator_radiosity_memory_hint
hint latency radiator_radiosity_latency_hint
update radiator_radiosity_update
utility radiator_radiosity_utility
```

Figure 7.14: Application Configuration File for Radiator

radiosity; and the accuracy of these predictors.

7.4.1 Porting Radiator to the multi-fidelity API

Figure 7.15 summarizes the modifications that I made to Radiator. In order to make Radiator adaptive, I wrote an ACF, provided glue code for the multi-fidelity API, and inserted calls to the glue code into the application, both the command-line and the GUI version. I also wrote a hint module containing the CPU and memory demand predictors described in this section. In all, I modified 5 files and 599 lines, about 1.2% of the total code base.

I also simplified Radiator's GUI. The original GUI had several controls for expert users to experiment with radiosity parameter settings. I eliminated these: the algorithm and the resolution are now chosen adaptively by invoking the multi-fidelity API, and all other parameters are fixed at their default values. The user is given a single control to specify their desired latency. My aim was to simplify the use of the application for a non-expert user, yet intelligently adapt those parameters that impact resource consumption and performance. To modify the GUI, I made a small number of changes to 3 source files, and edited the GUI layout definition using the XForms User Interface Designer [100]. The last two lines of Figure 7.15 show the cost of these modification.

Additionally, I fixed a number of memory leaks and other bugs in Radiator: these modifications are not counted in the evaluation, as they are tangential to multi-fidelity adaptation.

Files	Lines	Purpose
1	15	Application Configuration File
1	274	Hint module
1	68	Calls to glue code (command-line version)
1	123	Calls to glue code (GUI version)
2	119	Glue code
6	599	Total modifications for multifidelity
222	51065	Original code base
1	125	GUI layout description
3	23	Other GUI modifications

Figure 7.15: Modifications made to Radiator

Model	Polygon count
Dragon	108590
Whale	101814
Bunny	69543
Car	56972
Polar bear	48963
Human bust	29450

These 3-D models were provided by Andrew Willmott, designer and author of Radiator.

Figure 7.16: 3-D models used with Radiator

7.4.2 Predicting Radiator's CPU demand

Radiator's CPU demand predictor is similar to GLVU's: I measured the CPU demand of both progressive and hierarchical radiosity for 6 different scenes. Each scene consists of a 3-D model, an overhead light source, and a green patch of "ground" underneath. The models were chosen to have a wide range in size: from 29k polygons to 109k polygons. Figure 7.16 lists the 3-D models used in the test scenes.

For each scene, I ran progressive and hierarchical radiosity 50 times each, with a randomly chosen resolution for each computation. The logs from these experiments were used to derive both CPU and memory demand predictors. To avoid measurement errors due to heap re-use (Section 5.6.1), I restarted the application for each experimental run. I limited the CPU consumption of each run to 300 s (about 70 billion cycles on my test platform) and the memory footprint to 64 MB. Each run includes application initialization and loading of the scene file, as well as the radiosity computation itself: thus the actual resource limits for the radiosity computation were somewhat lower. This meant that, for some of the larger models, I could not cover the entire range of possible resolutions.

Figure 7.17 shows the results of these experiments. We notice that the CPU demand for hierarchical radiosity is an order of magnitude higher than that of progressive radiosity: when the processor is the bottleneck, we should always use the latter algorithm. We also see that CPU demand increases linearly with polygon count, and that each scene has a different slope.

These results suggest data-specific predictors for CPU demand. Since part of the radiosity algorithm operates on the original model (i.e., on all the polygons), we expect that the original polygon count p, as well as the scaled polygon count pr, will also have an effect on CPU demand. I used a linear model:

$$D_{cpu} = c_0 + c_1 p + c_2 pr$$

for the data-independent predictor. For data-specific prediction, however, p is a constant, and I could reduce the model to

$$D_{cpu} = c'_0 + c'_1 r$$

Figures 7.18 and 7.19 show the accuracy of data-specific predictors for each scene, and that of a data-independent predictor. We see that for all scenes except for the "Human bust", progressive radiosity has very predictable CPU demand; the data-independent predictor has a higher, but still acceptable common-case error (17%) than the data-independent predictors. Hierarchical radiosity's CPU demand is slightly less predictable for data-dependent predictors, and very unpredictable with the data-independent predictor ($E_{90} = 55\%$).

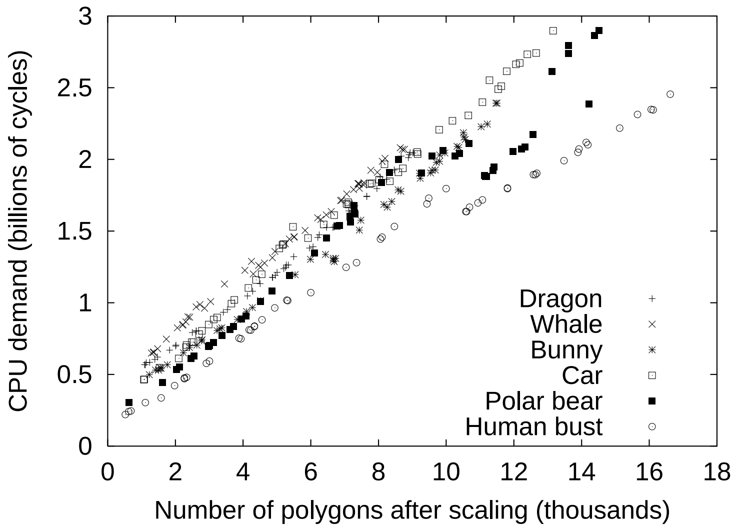
Unlike GLVU, Radiator is view-independent and unaffected by variations in camera position: thus, we do not need online learning updates to capture such variations over time. However, I do use online learning to customize the data-independent predictor to every new scene, i.e., to create data-specific predictors for new scenes at runtime.

7.4.3 Predicting Radiator's memory demand

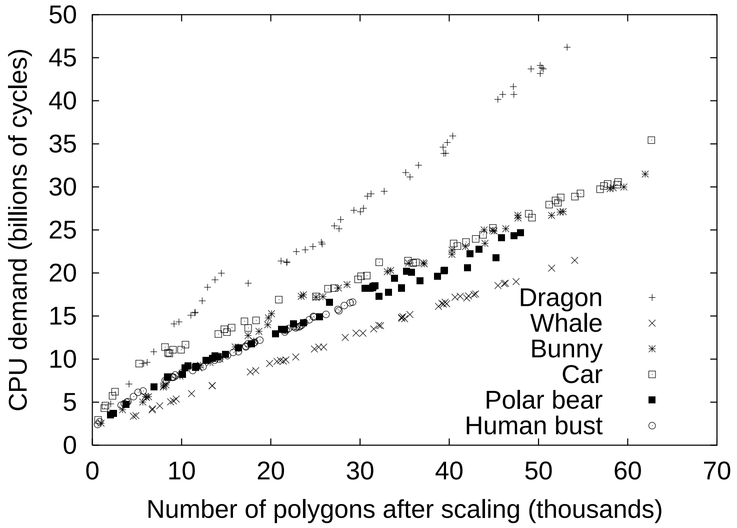
To build a memory demand predictor for Radiator, I took the same logs as before and plotted memory demand against resolution (Figure 7.20). We see that progressive radiosity's memory demand increases almost 7 times as fast as hierarchical radiosity's. We also see, for both algorithms, a strong linear correlation between memory demand and the polygon count pr. There seems to be data-specificity — each scene follows a different line — but all these lines have similar slopes, and differ only in their offsets. I hypothesized that the initial offset is contributed by the original (full-resolution) model, and the remainder by the degraded version. Thus, a linear model of the form

$$D_{memory} = c_0 + c_1 p + c_2 pr$$

should be a good predictor of memory demand, where c_0 is the fixed memory overhead, $c_1 p$ is the original model's contribution, and $c_2 pr$ is the memory for operating on the degraded version.



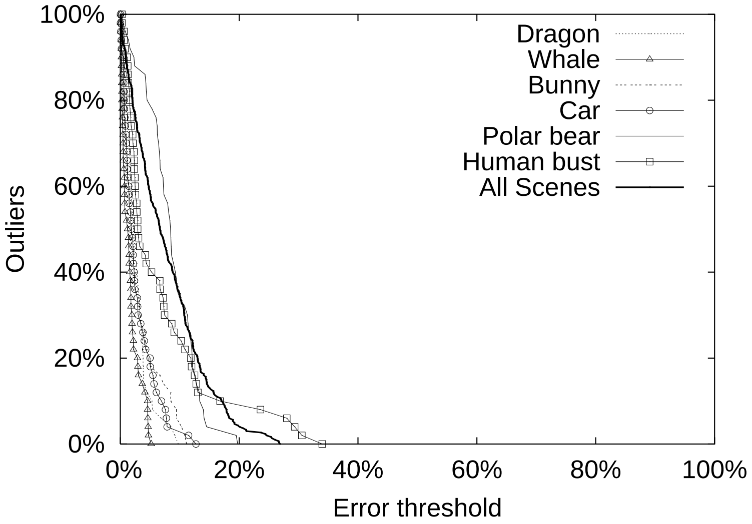
(a) Progressive radiosity



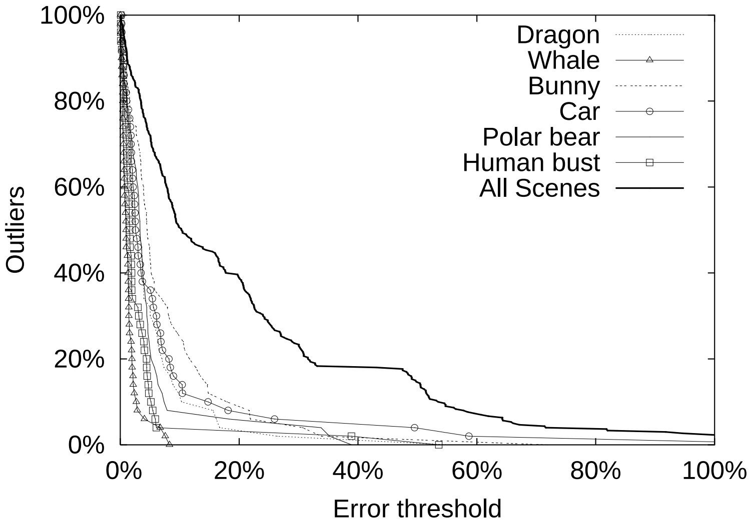
(b) Hierarchical radiosity

The x-axis is the number of polygons after scaling, i.e. pr where p is the original model size and the r is the resolution. The y axis is the CPU demand in millions of cycles. Note that both axes are on very different scales for the two graphs. Progressive radiosity uses much less CPU, but is constrained by its memory demand to a maximum of 18k polygons.

Figure 7.17: CPU demand of radiosity



(a) Progressive radiosity



(b) Hierarchical radiosity

The graphs show the outlier distribution for linear predictors of CPU demand, for both progressive and hierarchical radiosity. I show the distribution both for data-specific predictors, and for a single generic predictor applied across all scenes.

Figure 7.18: CPU demand prediction for Radiator: outlier distribution

Model	Progressive Radiosity		Hierarchical Radiosity	
	f_{20}	E_{90}	f_{20}	E_{90}
Dragon	0%	5%	4%	16%
Whale	0%	5%	0%	3%
Bunny	0%	9%	10%	22%
Car	0%	8%	8%	18%
Polar bear	0%	14%	6%	8%
Human bust	19%	24%	4%	5%
All scenes	4%	17%	39%	55%

The table shows the bad prediction frequency f_{20} and the common case error E_{90} for linear predictors of CPU demand, for both progressive and hierarchical radiosity. We show the accuracy both of data-specific predictors, and for a single generic predictor applied across all scenes.

Figure 7.19: CPU demand prediction accuracy for Radiator

Figures 7.21 and 7.22 show the accuracy of a data-independent linear predictor of this form, as well as that of data-specific predictors. We see that the data-independent model has very good accuracy: we have 0% outliers for both algorithms, and the common-case error is 1.3% for progressive and 3.3% for hierarchical radiosity. At runtime, I use the data-independent predictor for its simplicity: it avoids the overhead of doing online learning and maintaining scene-specific predictors, at a very small cost in increased prediction error.

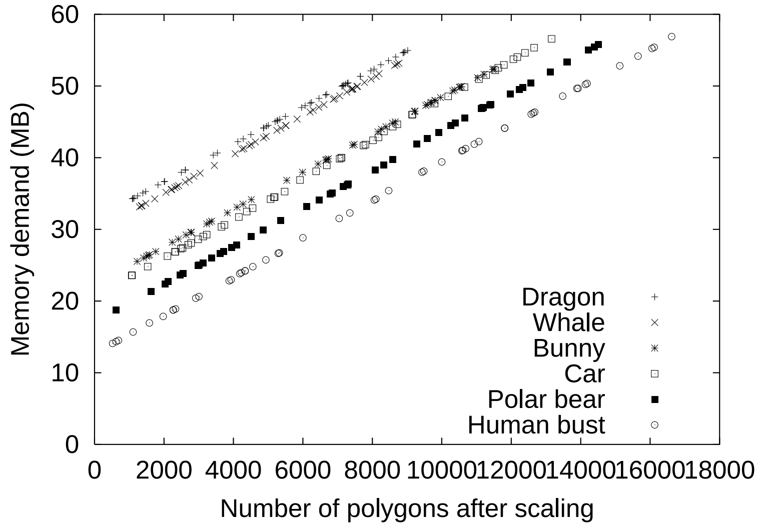
7.4.4 Extrapolating predictors for higher fidelities

The limitations of my mobile hardware platform made it impossible to measure Radiator's resource demand over the entire range of fidelities: I had to restrict memory demand to avoid thrashing and CPU demand to bound runtime to a reasonable value. I could measure progressive radiosity only up to 8k polygons and hierarchical radiosity up to 70k polygons, whereas the largest model has 109k polygons at full fidelity.

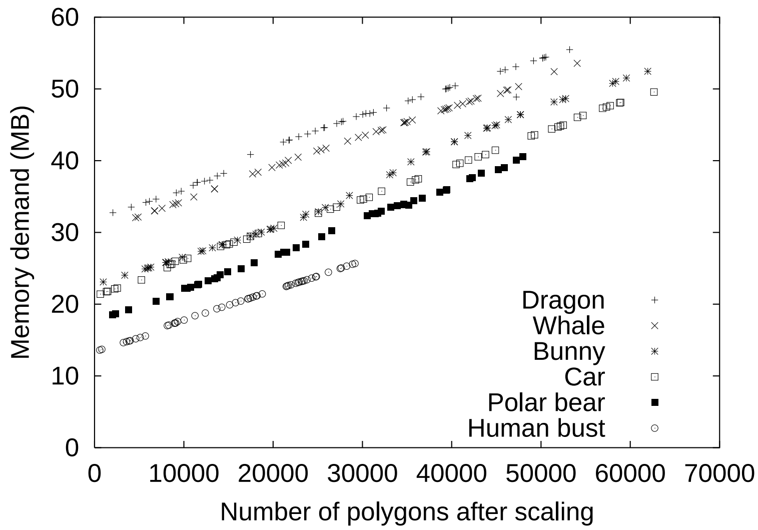
This limitation led us to ask the following questions:

- Does the data across all fidelities, on a faster machine, show the same trends as before: linear and data-independent for memory; linear and data-dependent for CPU?
- Can we extrapolate the measurements on the baseline configuration (IBM ThinkPad 560 with 233 MHz Mobile Pentium and 96 MB memory) to predict resource demand on the faster machine?

To answer these questions, I repeated the measurements of Radiator on a fast server machine (2.2 GHz Intel Xeon processor, 512 MB of memory): this time I tested the entire range of resolutions for both progressive and hierarchical radiosity. Figures 7.23 and 7.24 show the CPU and memory demand curves from these experiments. We see the same



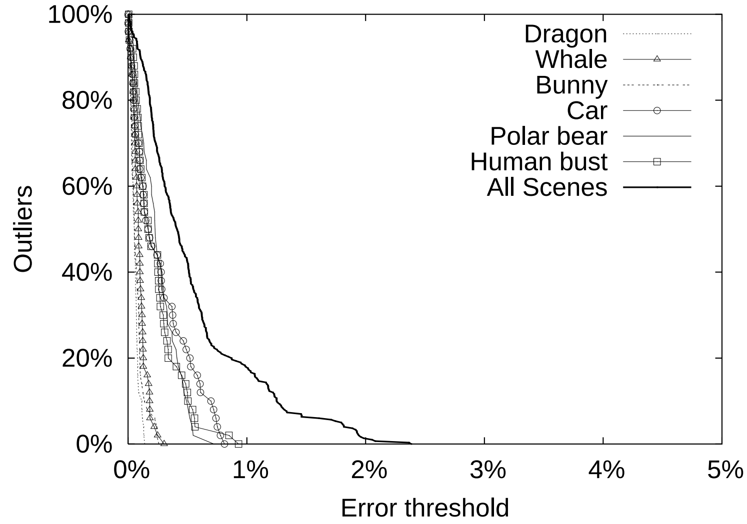
(a) Progressive radiosity



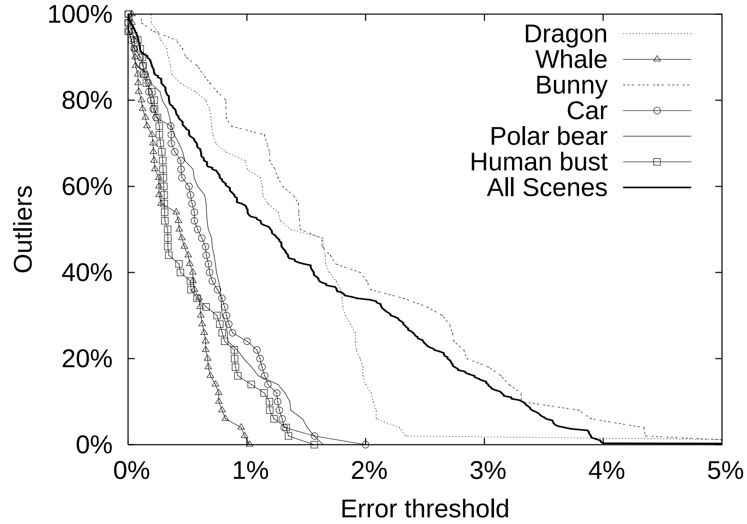
(b) Hierarchical radiosity

The x-axis is the number of polygons after scaling, i.e. pr where p is the original model size and the r is the resolution. The y axis is the memory demand in megabytes. Note that the x-axis is on different scales for the two graphs.

Figure 7.20: Memory demand of radiosity



(a) Progressive radiosity



(b) Outlier distribution: hierarchical radiosity

The graphs show the outlier distribution for linear predictors of memory demand, for both progressive and hierarchical radiosity. We show the distribution both for data-specific predictors, and for a single generic predictor applied across all scenes.

Figure 7.21: Memory demand prediction for Radiator: outlier distribution

Model	Progressive Radiosity		Hierarchical Radiosity	
	f_{20}	E_{90}	f_{20}	E_{90}
Dragon	0%	0.1%	0%	2.1%
Whale	0%	0.2%	0%	0.9%
Bunny	0%	0.2%	0%	3.8%
Car	0%	0.7%	0%	1.2%
Polar bear	0%	0.5%	0%	1.3%
Human bust	0%	0.5%	0%	1.2%
All objects	0%	1.3%	0%	3.3%

The table shows the bad prediction frequency f_{20} and the common case error E_{90} for linear predictors of memory demand, for both progressive and hierarchical radiosity. We show the accuracy both of data-specific predictors, and for a single generic predictor applied across all scenes.

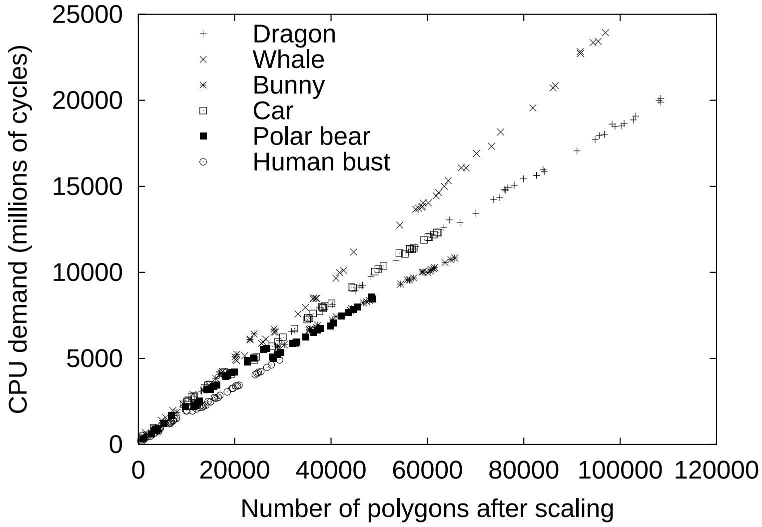
Figure 7.22: Memory demand prediction accuracy for Radiator

trends on the fast server as on the ThinkPad (Sections 7.4.2 and 7.4.3): CPU demand is linear but data-dependent; memory demand is linear and appears to be data-dependent, but the differences between scenes are due to the differences in their original polygon count p.

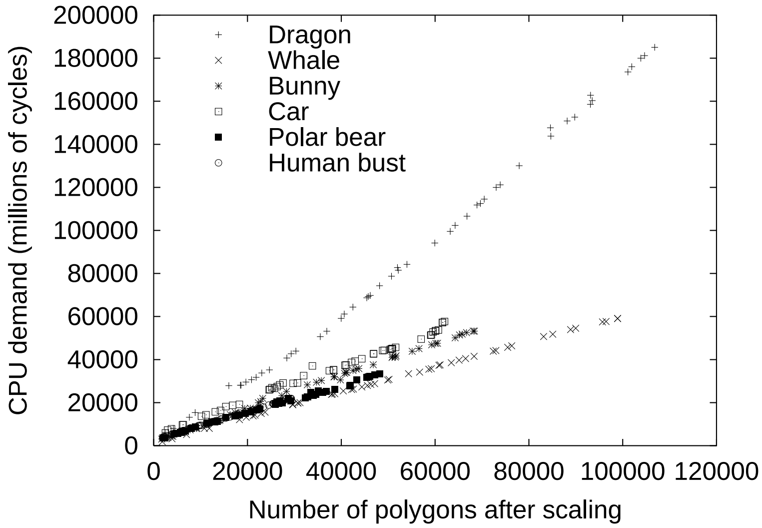
To confirm these findings, I generated linear predictors — both data-dependent and data-independent — for the new data, and measured their prediction errors. Figure 7.25 shows the CPU demand prediction accuracy for the new data, and compares it with two other cases: the original (ThinkPad) predictors tested on the new (Xeon) data, and the ThinkPad predictors tested on the ThinkPad data. We see that:

- as before, data-dependent predictors do better than data-independent ones.
- linear predictors do well, but not as well as before: perhaps because non-linear effects such as L1 cache contention are more apparent at higher fidelities.
- The extrapolated predictor (TP/Xeon) does well for progressive radiosity, but very badly for hierarchical radiosity. In the latter case, the slope of CPU demand vs. polygon count (Figure 7.23(b)) is consistently higher for the Xeon. I.e., the faster processor is actually doing *less* per cycle, causing the extrapolated predictor to underestimate the number of cycles required.
- The extrapolated predictor (TP/Xeon/unscaled) has enormous errors. In other words, if we just measure CPU demand in terms of time, without scaling for processor speed, then we consistently overestimate CPU demand by a factor of 5-11 (the Xeon's processor speed is 9.5 times that of the ThinkPad's).

These findings confirm my previous observations: that CPU demand is linear and data-dependent. They also show that "cycles consumed" as a metric of CPU demand does not always translate well across processor architectures: predictors derived on one processor must be modified when used on another. For simple predictors, online updates will achieve



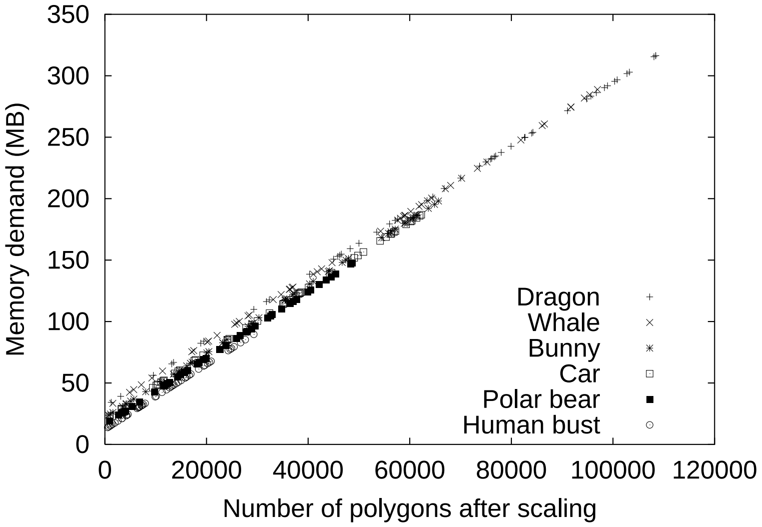
(a) Progressive radiosity



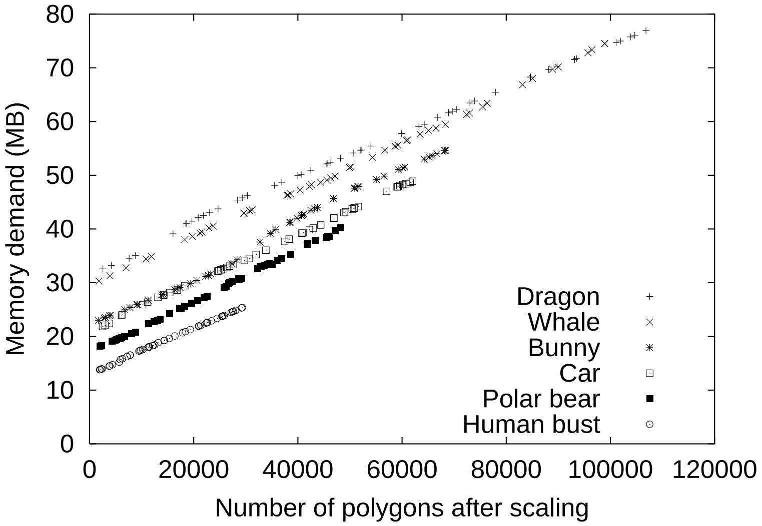
(b) Hierarchical radiosity

The x-axis is the number of polygons after scaling, i.e. pr where p is the original model size and the r is the resolution. The y axis is the CPU demand in millions of cycles: note that it is on very different scales for the two graphs.

Figure 7.23: CPU demand of radiosity on fast server



(a) Progressive radiosity



(b) Hierarchical radiosity

The x-axis is the number of polygons after scaling, i.e. pr where p is the original model size and the r is the resolution. The y axis is the memory demand in megabytes: note that it is on very different scale for the two graphs.

Figure 7.24: Memory demand of radiosity on fast server

Model	Xeon/Xeon		TP/Xeon		TP/Xeon/unscaled		TP/TP	
	f_{20}	E_{90}	f_{20}	E_{90}	f_{20}	E_{90}	f_{20}	E_{90}
Dragon	4%	4%	0%	6%	100%	872%	0%	5%
Whale	8%	13%	14%	22%	100%	751%	0%	5%
Bunny	24%	78%	14%	25%	100%	944%	0%	9%
Car	4%	6%	0%	9%	100%	857%	0%	8%
Polar bear	14%	30%	2%	16%	100%	890%	0%	14%
Human bust	2%	11%	4%	19%	100%	940%	19%	24%
All objects	13%	23%	19%	25%	100%	882%	4%	17%

(a) Progressive radiosity

Model	Xeon/Xeon		TP/Xeon		TP/Xeon/unscaled		TP/TP	
	f_{20}	E_{90}	f_{20}	E_{90}	f_{20}	E_{90}	f_{20}	E_{90}
Dragon	10%	24%	94%	54%	100%	563%	4%	16%
Whale	2%	4%	96%	36%	100%	594%	0%	3%
Bunny	0%	11%	84%	34%	100%	1114%	10%	22%
Car	2%	6%	92%	40%	100%	766%	8%	18%
Polar bear	0%	5%	52%	26%	100%	908%	6%	8%
Human bust	0%	4%	42%	23%	100%	849%	4%	5%
All objects	64%	95%	67%	62%	100%	862%	39%	55%

(b) Hierarchical radiosity

The first six lines of each table show the bad prediction frequency f_{20} and common-case error E_{90} for data-specific linear predictors; the last line corresponds to a data-independent predictor evaluated over all the objects. We show four combinations of predictors and test data: Xeon/Xeon (derived from and tested on Xeon data); TP/Xeon (derived from ThinkPad data and tested on Xeon data); TP/Xeon/unscaled (the same as the former, but without CPU speed scaling); and TP/TP (derived from ThinkPad data and tested on ThinkPad data).

Figure 7.25: CPU demand prediction accuracy for Radiator on a fast server

Model	Xeon/Xeon		TP/Xeon		TP/TP	
	f_{20}	E_{90}	f_{20}	E_{90}	f_{20}	E_{90}
Dragon	0%	1.6%	0%	1.5%	0%	0.1%
Whale	0%	0.4%	0%	1.2%	0%	0.2%
Bunny	0%	1.2%	0%	2.4%	0%	0.2%
Car	0%	0.5%	0%	2.8%	0%	0.7%
Polar bear	0%	0.5%	0%	1.4%	0%	0.5%
Human bust	0%	1.1%	0%	2.3%	0%	0.5%
All objects	0%	1.4%	0%	1.4%	0%	1.3%

(a) Progressive radiosity

Model	Xeon/Xeon		TP/Xeon		TP/TP	
	f_{20}	E_{90}	f_{20}	E_{90}	f_{20}	E_{90}
Dragon	0%	1.4%	0%	3.7%	0%	2.1%
Whale	0%	1.3%	0%	2.2%	0%	0.9%
Bunny	0%	3.9%	0%	4.8%	0%	3.8%
Car	0%	1.2%	0%	1.9%	0%	1.2%
Polar bear	0%	1.2%	0%	2.5%	0%	1.3%
Human bust	0%	1.1%	0%	3.3%	0%	1.2%
All objects	0%	3.6%	0%	4.2%	0%	3.3%

(b) Hierarchical radiosity

The first six lines of each table show the bad prediction frequency f_{20} and common-case error E_{90} for data-specific linear predictors; the last line corresponds to a data-independent predictor evaluated over all the objects. We show three combinations of predictors and test data: Xeon/Xeon (derived from and tested on Xeon data); TP/Xeon (derived from ThinkPad data and tested on Xeon data); and TP/TP (derived from ThinkPad data and tested on ThinkPad data).

Figure 7.26: Memory demand prediction accuracy for Radiator on a fast server

this after a few operations; more complex predictors need to be "ported" by other means. Nevertheless, "cycles consumed" is a much better metric than "CPU time", which does not compensate in any way for changes in processor speed.

Figure 7.26 shows the prediction accuracy for memory demand. All three test configurations show very low error, both for data-dependent and data-independent prediction (the TP/Xeon predictor has slightly higher errors, but this is only because the extrapolation inevitably magnifies any noise in the original data). This confirms my original findings about Radiator's memory — that it is linear and data-independent – and also shows that it is not sensitive to changes in processor architecture.

7.4.5 Summary

I studied Radiator: an implementation of radiosity, a view-independent 3-D shading algorithm. With a small number of modifications, I made the application adaptive: it now chooses one of two algorithms, and a resolution between 0 and 1, by invoking the multi-fidelity API. I built and evaluated linear predictors for both CPU and memory demand. Data-specific predictors for CPU demand gave us a common-case error of 3%–24%, and were significantly better than data-independent predictors. Data-independent predictors for memory had an error of at most 3.3%; data-specific ones did slightly better.

7.5 Web browsing

Web browsing is a common activity on both mobile and desktop computers. In the mobile case, fetching large images over a wireless network results in consumption of valuable battery energy, as well as increased network traffic. For such images, lossy JPEG compression [94] is a well-known way to adapt energy [34] and network [37, 71] demand to resource supply.

My adaptive web browser [71] consists of an unmodified Netscape binary and a local HTTP proxy called the *cellophane*, originally written by Eric Tilton. The cellophane intercepts all web requests and transforms them into Odyssey [71] system calls. Odyssey then requests a degraded version of the image from a distilling server located on the other side of the wireless link. The distiller fetches the image from the web server, JPEG-compresses it if required, and returns it to the client; it uses the Independent JPEG Group library [43] to do the actual compression.

For this application, a multi-fidelity operation consists of compressing, fetching and rendering a single image on the screen. Currently I ignore the cost of fetching the image from the web server to the distiller: this step has little impact on the mobile client's energy usage or wireless traffic, though it does increase the overall latency. Each operation has one data feature or nontunable parameter — the size of the original image — and one

```
description web:fetchimage
logfile /usr/odyssey/etc/web.fetchimage.log
param imagesize ordered 0-infinity
fidelity jqf ordered 5-80 step 1
```

Figure 7.27: Application Configuration File for web image fetch

fidelity metric — the JPEG Quality Factor (JQF) [94, 14] at which compression is done. Figure 7.27 shows the Application Configuration File for this application. The JQF can take any integer value from 0–100: however, I found that values below 5 cause the compression algorithm to behave unreliably, and values above 80 cause a sharp increase in the size of the compressed image, sometimes exceeding that of the original image. Consequently, I restricted the JQF to the range 5–80.

In the remainder of this section I describe how I modified the cellophane to use the multi-fidelity API, and how I predicted the network and energy demand of fetching images over a wireless network. In all of my experiments, the distiller ran on an IBM ThinkPad 570 with a 366 MHz Mobile Pentium II processor and 128 MB of RAM. The wireless link between client and server was provided by a 2 Mbps, 2.4 GHz Lucent WaveLAN in ad-hoc mode. All images were fetched from a web server running on the same host as the distiller.

7.5.1 Porting Netscape to the multi-fidelity API

The Netscape "application" consists of three parts: the Netscape binary (at the time, the source code was not freely available), the cellophane, and the distilling server. In order to use the multi-fidelity API, I only needed to modify the cellophane. On receiving a request for an image, the modified cellophane

- gets the image size from the distiller.
- invokes $begin\_fidelity\_op$ to find the appropriate fidelity (JQF) for that image size.
- passes the JQF to Odyssey.
- fetches the degraded image from the distilling server (through Odyssey), and passes it to Netscape.
- waits for Netscape to finish rendering the image, by tracking Netscape's status window.
- calls $end\_fidelity\_op$.

Normally, Netscape's operation is driven by a user interacting with its GUI. However, when collecting log data, I had to automatically execute a large number of operations without requiring user interaction. For this purpose, I used a *remote control* program, based on the Netscape Remote Control reference implementation [99], which allowed us to issue "fetch and display URL" commands to Netscape from another process. An extended ver-

Figure 7.28: Netscape's "shooting stars" status window: idle (left) and animated

Files	Lines	Purpose
1	83	Insertion of multi-fidelity calls
1	393	Remote control program
1	211	Tracking Netscape status window
1	4	Application Configuration File
2	170	Hint module
6	861	Total modifications for multifidelity
2	697	Cellophane (original size)
7	3224	Distiller (unmodified)

Note that the count for the remote control program (line 2) excludes code from the original Netscape reference implementation, and code to track Netscape's status window.

Figure 7.29: Code modifications for adaptive web browsing

sion, built by Jason Flinn and Kip Walker [81], can also detect when Netscape has finished displaying a document. It does so by tracking the state of Netscape's "shooting stars" status window (Figure 7.28), which displays a static image when Netscape is idle and an animated one when it is busy fetching or rendering a document. The remote control program also invokes multi-fidelity API and sets the JQF on behalf of the application: thus, in batch mode, I used the unmodified cellophane which does not make multi-fidelity calls.

Figure 7.29 shows the cost of porting Netscape to the multi-fidelity API.

7.5.2 Predicting the network demand of image fetch

The network demand — the number of bytes read from the server — is determined by the size of the JPEG-compressed image. Can we predict this size as a function of the JPEG Quality Factor? I measured the compression ratio achieved by JPEG at various JQF values, on six images of sizes varying from 8 KB to 1.3 MB. (Figure 7.30). Figure 7.31(a) shows the compression ratios as a function of the JQF. We see that the plot is roughly linear in

Image	Size (bytes)
nsh	1394081
apple	174650
radio	114816
castle	58223
circuit	19685
laserdt	8802

Figure 7.30: Test images used with web browser

the range 5–80: beyond 80 it begins to increase rapidly and non-linearly. We see this non-linearity even on a log-log plot (Figure 7.31(b)), indicating that it is not a simple polynomial or exponential relationship. Since compression ratios close to or exceeding 1 are neither predictable nor useful, I avoided JQF values above 80.

In the range 5–80, I model the compression ratio r as a linear function of the JQF f:

$$r = c_0 + c_1 f$$

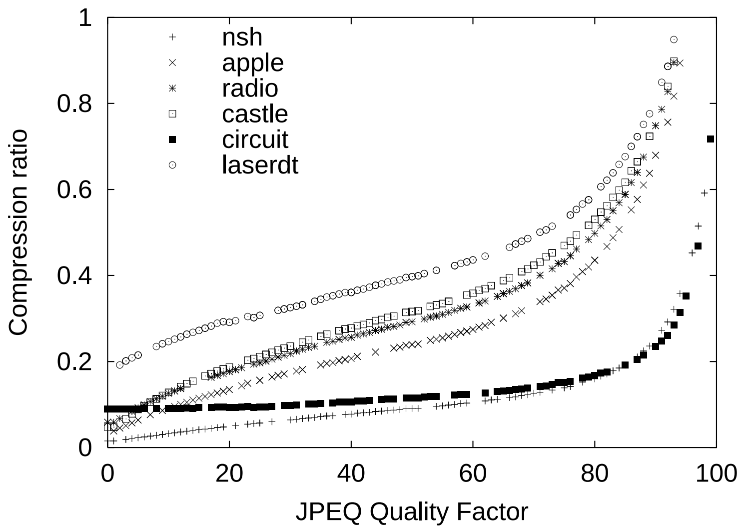
As Figure 7.31 shows, the values of c_0 and c_1 are different for different images, i.e. the model needs to be data-specific. Given the compression ratio r and the original image size S, we can predict the network demand as

$$D_{recv} = c_0' + rS = c_0' + (c_0 + c_1 f)S = c_0' + c_1' S + c_2' f S$$

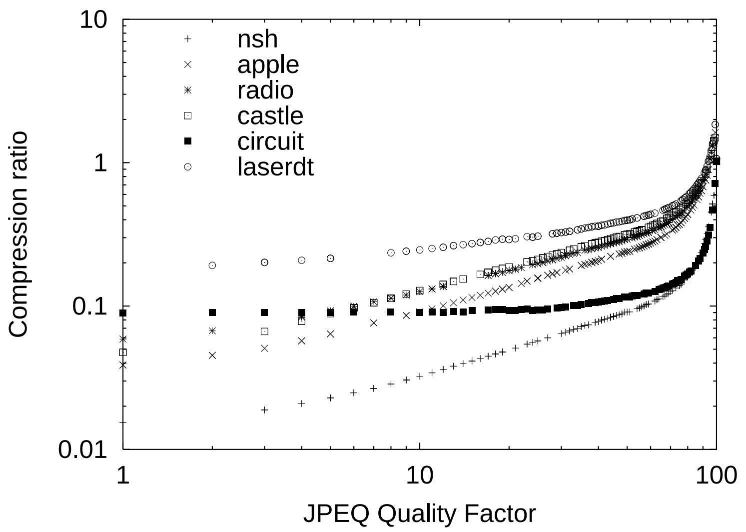
Figure 7.32 shows the accuracy of this model, evaluated over 6 images with 100 data points each. As we might expect, we see high accuracy for data-specific predictors and low accuracy for the data-independent predictor: we have already seen that the relationship between JQF and compression ratio is data-specific.

7.5.3 Predicting the energy demand of web image fetch

Fetching large objects over a wireless network consumes a substantial amount of energy. Compression can help here also: it reduces wireless traffic, and also the time for which the wireless interface is active. To measure the effect of compression on energy usage, I ran the same experiments as in the previous section and measured the energy usage of each operation using PowerScope [31]. PowerScope allows us to sample the power consumption of a laptop, and to attribute it to one of the many processes running on the machine. I extended PowerScope to include a timestamp with each sample. In post-processing, I used these timestamps to correlate power samples with the operations logged by Odyssey. I computed the total energy consumed during an operation, subtracted out the known background power consumption, and attributed the remaining energy consumption to that operation. On my test machine, this background or baseline power consumption was 7.94 Watts.



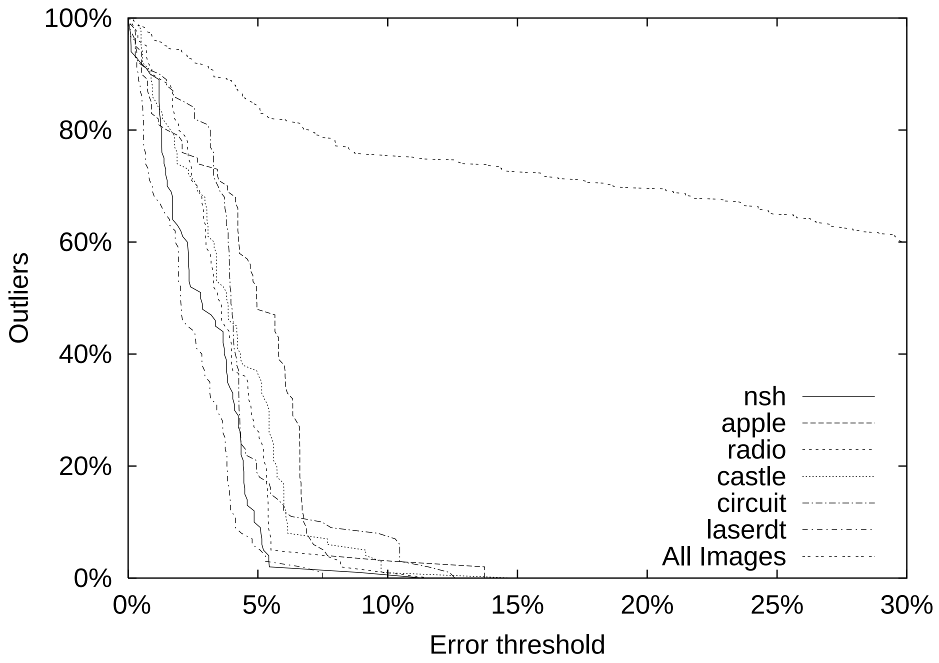
(a) Normal scale



(b) Log scale

The x-axis shows the JPEG Quality Factor f, from 0 to 100. The y-axis shows the compression ratio r achieved at that quality factor.

Figure 7.31: Compression ratio as a function of JPEG Quality Factor



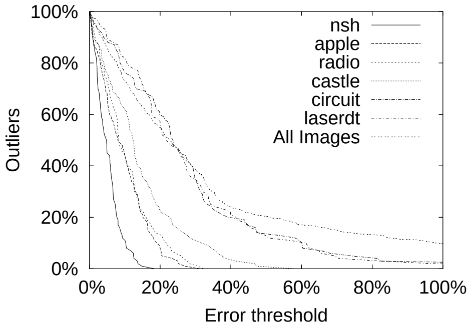
(a) Outlier distribution

Image	Pred. error	
	f_{20}	E_{90}
nsh	0%	5%
apple	0%	7%
radio	0%	5%
castle	0%	6%
circuit	0%	8%
laserdt	0%	4%
All images	70%	526%

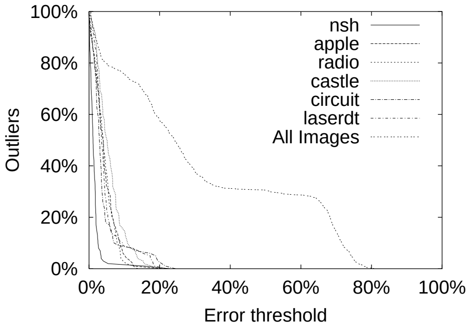
(b) Bad prediction frequency and common-case error

The graph shows prediction accuracy for data-specific and data-independent linear models of network demand for image fetches. The table shows the corresponding bad prediction frequency f_{20} and the common case error E_{90}.

Figure 7.32: Network demand prediction accuracy for web image fetch



(a) Outlier distribution: Netscape



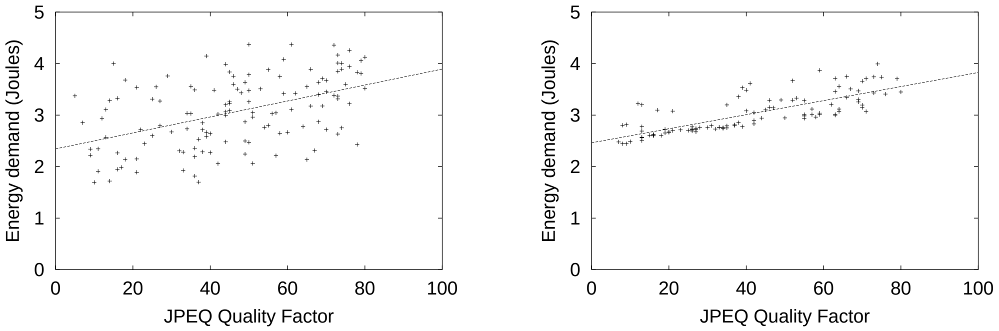
(b) Outlier distribution: simple browser

Image	Netscape		Simple browser	
	f_{bad}	E_{90}	f_{bad}	E_{90}
nsh	0%	10%	1%	3%
apple	9%	20%	1%	9%
radio	14%	22%	0%	8%
castle	23%	34%	0%	11%
circuit	60%	60%	3%	8%
laserdt	58%	62%	1%	10%
All images	55%	100%	58%	72%

(c) Bad prediction frequency and common-case error

The graphs show prediction accuracy for a linear model of network demand for image fetches: using Netscape, and using a simple browser. The table shows the corresponding bad prediction frequency f_{bad} and the common case error E_{90}.

Figure 7.33: Energy demand prediction accuracy for web image fetch



(a) Netscape (b) Simple browser

The x-axis is the JPEG Quality factor. The y-axis is the energy demand in Joules of fetching and rendering the "castle" image. Each point represents an experimental run; the lines are linear best fits through the sample points.

Figure 7.34: Energy demand of different browsers

We expect the energy demand to have a fixed portion, a network portion linear in the amount of data fetched, a decompression portion linear in the compressed and uncompressed image sizes, and a rendering portion linear in the uncompressed image size. I.e.,

$$D_{energy} = c_0 + c_1 rS + (c_2 rS + c_3 S) + c_4 S$$

where r is the compression ratio. This reduces to

$$D_{energy} = c'_0 + c'_1 rS + c'_2 S$$

If we use a linear model for compression ratio r as a function of the JQF f, we get

$$D_{energy} = c''_0 + c''_1 S + c''_2 fS$$

Figure 7.33(a) shows the accuracy of this predictor for Netscape's energy demand. We see that, as before, data-specific predictors outperformed the generic predictor; however, even the data-specific predictors had a high error rate, especially for the smaller images.

A predictor might perform badly for two reasons: either the underlying model was poorly chosen (i.e. the true relationship is not linear), or there are noise sources that cause deviations from the model. In the case of Netscape, the poor accuracy was due to noise in the data. I suspect that the noise is caused by the background thread that animates Netscape's "shooting stars" status window: non-deterministic effects in the user-level thread scheduler cause this thread to consume a variable amount of CPU and hence energy.

How well would my predictors perform in the absence of these non-deterministic effects? Since I did not have source code to Netscape at the time, I answered this question by building another, simpler browser. This program can send HTTP requests to the cellophane, read image data, and display the image using *xv* [11], a freely available image editor for X. I repeated my experiments with this browser and found a dramatic improvement in prediction accuracy, as shown by Figures 7.33(b) and 7.33(c). Figure 7.34 shows graphically the contrast between Netscape and the simple browser for the "castle" object.

7.5.4 Summary

I studied the network and energy demand of web image fetch, and showed that both can be reduced through multi-fidelity adaptation. I also showed that multi-fidelity adaptation is possible even without application source code, through the use of proxies. However, this black-box approach leads to additional complexity of implementation. It also hinders the building of simple and accurate predictors: sometimes the application exhibits side-effects that cannot be debugged without source code access.

I also showed that data-specific predictors can accurately predict the compressed size of an image as a function of the JPEG Quality Factor. However, a data-independent predictor of compressed size had low accuracy. This is unsurprising: JPEG compression ratios depend on image content, and cannot be predicted from image size alone. My results here are broadly in agreement with those of Han et al. [44], who studied JPEG transcoding for a large number of web images. They show that, for a fixed JPEG level:

- There is a linear correspondence between uncompressed and compressed image size, but with a large amount of noise, i.e. variation across images.
- There is a better linear correspondence between the *area* (number of pixels) of the uncompressed image and the compressed image size.
- The variance from the trend is higher for larger images (in terms of either byte size and area), i.e. the distribution is *heteroscedastic*.

By combining these results with mine — perhaps with a linear model on JPEG level and image area as well as input byte size — the accuracy of the data-independent predictor might be improved.

7.6 Speech recognition

Janus [92] is a speech recognition system that takes a digital sound sample — an *utterance* — and returns the words it recognizes in the utterance as an ASCII string. It is both CPU and memory-intensive: here I focus on adapting Janus's CPU demand using the multi-fidelity API. The CPU demand depends on the size of the *language model* used: one adaptation technique is to use a smaller language model for the same task, giving us faster recognition at the price of degraded recognition. In the experiments described here, I allow Janus to switch between a normal ("large") language model and a reduced ("small")

```
description janus:recognize
logfile /usr/odyssey/etc/janus_recognize.log
mode normal
hintfile /usr/odyssey/lib/janus-recognize.so
utility janus_recognize_utility
init janus_recognize_init

param utterance_length ordered 0-infinity
fidelity vocab_size unordered small large
fidelity location unordered local hybrid remote
```

Figure 7.35: Application Configuration File for Janus

version.

The heavy resource demands of Janus also make *remote execution* an attractive strategy. Jason Flinn has modified Janus so that it can execute each recognition in one of three different configurations: entirely local, entirely on a remote compute server, or "hybrid". In the hybrid mode, the first phase is done locally, in order to reduce the amount of data shipped over the network. This is a *vector quantization* [76] step that transforms the raw speech utterance into a more compact representation. The bulk of the processing is then done at the remote server. Thus "location" is a tunable parameter for Janus, with three possible values. Although the output quality is the same in each case, the resource tradeoffs between local CPU, memory, network bandwidth, and remote CPU are different.

Depending on the resource supply at runtime, the solver will pick one of these three modes. If a remote server is required, the solver also determines the best server to use: this information is not exposed to the application. To perform the remote execution, the system uses Spectra [33], an engine for server discovery and client-server communication that was built by Jason Flinn, and is integrated with the multi-fidelity runtime.

The resource demand of speech recognition also depends on the length of the utterance being recognized. Thus, Janus has two fidelity metrics — model size and location — and one input parameter — utterance length (Figure 7.35).

7.6.1 Porting Janus to the multi-fidelity API

The Janus code base is structured as a toolkit. The core functionality is implemented as C procedures; Janus "applications" are Tcl scripts that invoke this functionality. To make a multi-fidelity Janus applications, we

- wrote an ACF and resource hint module.

Files	Lines	Purpose
1	10	Application Configuration File
1	86	Hint module
1	394	Command-line client
3	350	Server-side wrapper (C)
4	241	Server-side wrapper (Tcl)
10	1081	Total modifications for multifidelity
209	126397	Original code base
1	1045	Large (original) dictionary
1	105	Small (reduced) dictionary

Note that the Tcl files were largely based on existing Janus source code: the line counts here are conservative, i.e. they over-estimate the number of new lines of code written. Also note that the small dictionary was created simply by deleting entries from the large dictionary (each dictionary entry counts as one line).

Figure 7.36: Modifications made to Janus

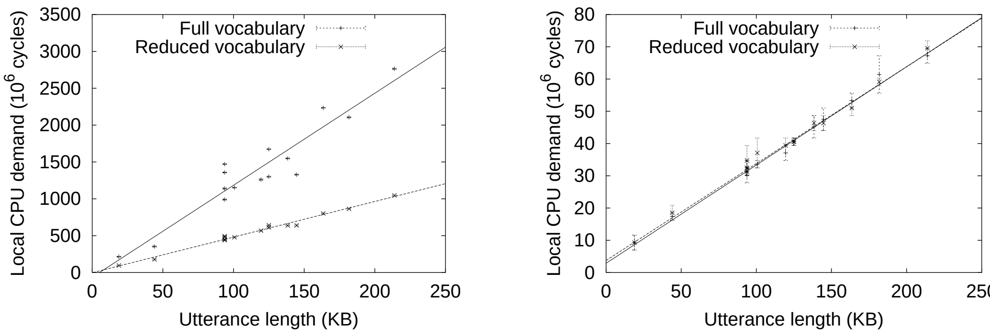
- created a reduced version of the language model. The language model is automatically created from a *dictionary* — a text file with one entry per line. The experiments in this section use a "Pittsburgh tourist" language model, which is part of the Janus distribution. I created the reduced version of the language model by removing all proper nouns from the dictionary.
- created a remote execution server, by adding a code layer that interfaces between the Spectra server-side support and the Janus toolkit. This server exports five RPC's: "load vocabulary", "unload vocabulary", "do quantization", "recognize quantized data", and "recognize raw (unquantized) data". This server runs both on the client machine and on remote compute servers.
- wrote a simple client program, which reads a sound file and invokes the multi-fidelity API to find the appropriate location and vocabulary size. The client then invokes Spectra to perform the computation on the local and/or remote servers.

Figure 7.36 shows the cost of these modifications.

7.6.2 Predicting resource consumption for Janus

To build resource demand predictors for Janus, I used 15 pre-recorded utterances ranging in size from 20 KB to 140 KB. For each utterance, I performed 5 recognitions for each of the 6 combinations of location and vocabulary size, for a total of 450 samples.

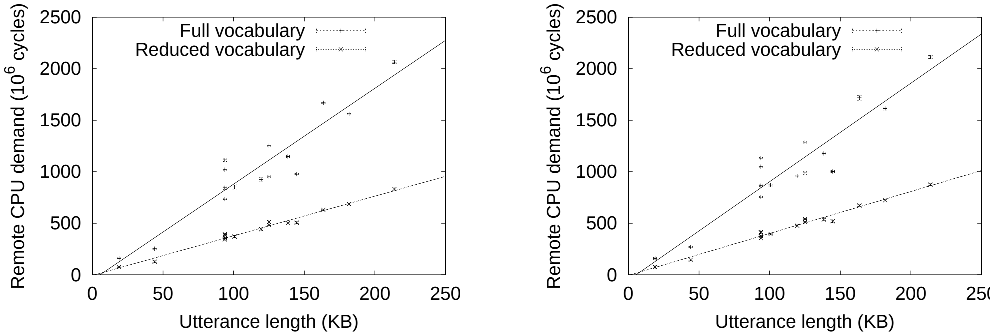
Figures 7.37, 7.38, and 7.39 show the local CPU, remote CPU, and network transmission demand respectively, for these experiments. In each case, I plot resource demand



(a) "Local" mode (b) "Hybrid" mode

The x-axis is the utterance length in bytes. The y-axis is the local CPU demand in millions of cycles on a Mobile Pentium MMX processor. Each bar is the min-max range for 5 recognitions of a single utterance. The lines are best fits for the full and reduced vocabulary case, respectively. Note the different y scales for the "local" and "hybrid" modes. The "remote" mode is not shown as it uses a negligible amount of local CPU.

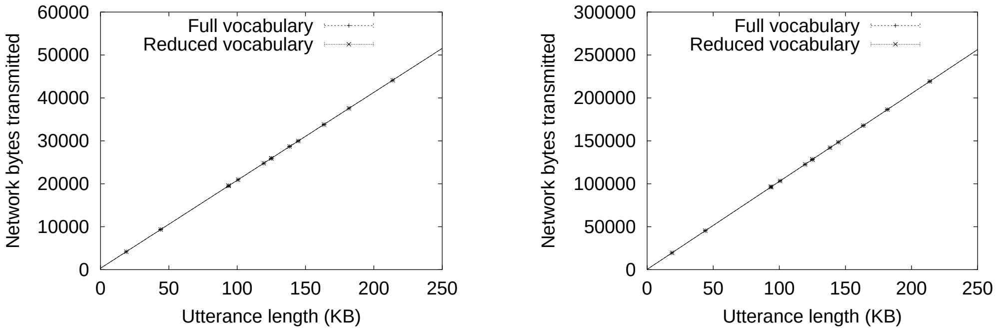
Figure 7.37: Local CPU demand of speech recognition



(a) "Hybrid" mode (b) "Remote" mode

The x-axis is the utterance length in bytes. The y-axis is the remote CPU demand in millions of cycles on a Mobile Pentium II processor. Each bar is the min-max range for 5 recognitions of a single utterance. The lines are best fits for the full and reduced vocabulary case, respectively.

Figure 7.38: Remote CPU demand of speech recognition



(a) "Hybrid" mode (b) "Remote" mode

The x-axis is the utterance length in bytes. The y-axis is the network transmission in bytes. Each bar is the min-max range for 5 recognitions of a single utterance. The lines are best fits for the full and reduced vocabulary case, respectively. Network receive demand is not shown as it is negligible.

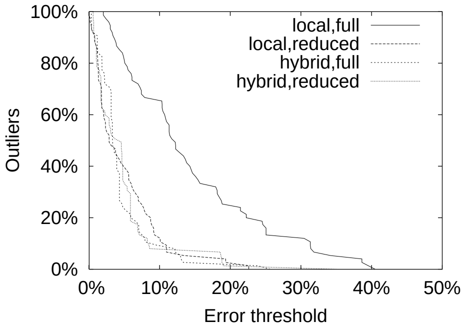
Figure 7.39: Network demand of speech recognition

against utterance size. Each utterance is plotted as a vertical bar ranging from the minimum to the maximum of the 5 runs. I also show the best-fit lines for resource demand as a function of utterance size. I do not show network or remote CPU demand for the "local" mode, since it does not use those resources; I also omit the local CPU demand of the "remote" mode, and the network receive demand in all cases, as they were negligible.

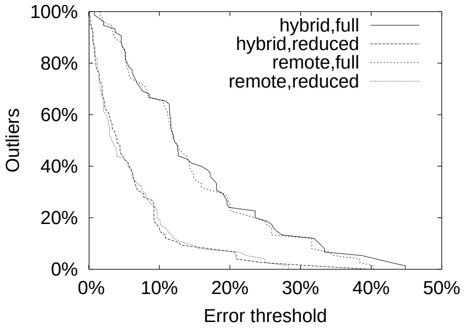
We note the following features of the data

- local and remote CPU increase roughly linearly with utterance length.
- network transmission increases exactly linearly with utterance length.
- the remote mode always transmits exactly 5 times as much data as the hybrid: the quantization step always "downsamples" the audio data by a factor of 5.
- the local CPU used by the hybrid mode is small (less than 3% of the total CPU demand of recognition); thus "hybrid" is preferable to "remote" unless we have extremely high bandwidth to the remote server, or a very large disparity between local and remote CPU speeds.
- the local CPU demand of the "local" mode differs slightly from the remote CPU demand of the "remote" mode, although both perform the same computation: this is because the two CPUs have slightly different architectures, and require different numbers of cycles for the same computation.
- the variation in resource consumption for a single utterance (for a given remote execution mode and vocabulary size) is extremely small.

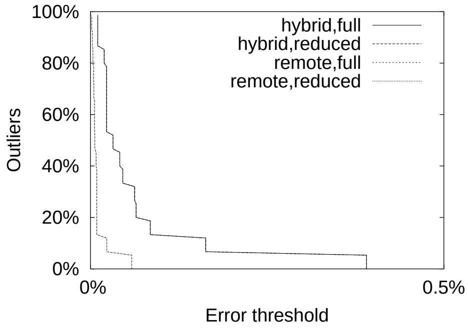
The variation across utterances indicates that a data-specific resource prediction scheme



(a) Outlier distribution: local CPU



(b) Outlier distribution: remote CPU



(c) Outlier distribution: network

Resource	Local mode				Hybrid mode				Remote mode			
	Full		Reduced		Full		Reduced		Full		Reduced	
	f_{20}	E_{90}	f_{20}	E_{90}	f_{20}	E_{90}	f_{20}	E_{90}	f_{20}	E_{90}	f_{20}	E_{90}
Local CPU	25%	31%	3%	11%	3%	10%	1%	8%	n.a.		n.a.	
Rem. CPU	n.a.		n.a.		24%	33%	8%	13%	27%	32%	8%	15%
Network	n.a.		n.a.		0%	0.2%	0%	0.2%	0%	0.02%	0%	0.02%

(d) Bad prediction frequency and common-case error

The graphs show outlier distributions for linear predictors of resource demand (local CPU, remote CPU, and network transmission) for speech recognition. The table shows the bad prediction frequency f_{20} and the common-case error E_{90} corresponding to the graphs.

Figure 7.40: Resource demand prediction accuracy for Janus

would perform well. Unfortunately, such predictors are not practical, since it is very unlikely that a real user will generate exactly the same audio data a second time. We are left with utterance length U as the only easily extracted feature to base our predictions on. I use a linear model

$$D = c_0 + c_1 U$$

for all three resources of interest: local CPU, remote CPU, and network transmission

Figure 7.40 shows the accuracy of linear predictors of local CPU, remote CPU, and network transmission. It shows the outlier distribution, bad prediction frequency and common case error for each resource, for each of the 6 “bins” defined by location and vocabulary size. We note that

- the network demand predictors are extremely accurate, with errors always below 0.5%.
- the local CPU demand for “hybrid” (i.e. the vector quantization step), apart from being small, is also very predictable, with E_{90} under 3%.
- the CPU demand of the main (post-quantization) recognition step — whether executed locally or remotely — is smaller, and also more predictable, with the reduced vocabulary than with the full vocabulary.

We have sizeable ($E_{90} = 24\%$) prediction errors only for the CPU demand of the main recognition step (i.e., local CPU demand for “local”, remote CPU demand for “hybrid” and “remote”) when we use the full vocabulary. However, in practice, even these errors do not usually lead to incorrect adaptive decisions. Typically, the difference in predicted latency between different choices — e.g. executing locally vs. remotely — is larger than the prediction error, and so the runtime system makes the correct choice. Of course, in scenarios where different adaptive choices give us only slightly differing latencies, the system is more likely to make incorrect choices: however, in such scenarios, the penalty for these incorrect choices is also low. Jason Flinn and the author have shown that the multi-fidelity system always chooses the optimal remote execution mode and vocabulary size in a variety of resource scenarios [33]: in other words although there are inevitably errors in resource prediction, they are small enough that the optimal configuration is still predicted to be optimal.

7.6.3 File access prediction for Janus

In Chapter 2, I observed that file cache state is a key resource in mobile systems. This means that it is important to be able to predict the files that will be accessed by the application at various fidelities, in order to pick fidelity values that reduce cache misses.

In the case of Janus, each scenario (e.g., “Pittsburgh tourist”) has its own specialized vocabulary. If the required language model is not in the cache, then recognition on that machine will incur an expensive cache miss. (The language models used in my evaluation are around 250 KB in size.)

I stored the language models in Coda, and used the file cache predictor (Sections 5.5.4 and 6.4.2) to predict the cache miss cost for any combination of remote execution mode and vocabulary size. This approach works extremely well for Janus: the files required on the client and server are determined entirely by the scenario, vocabulary size, and remote execution mode. I.e., it can always predict, with 100% accuracy, the files that will be accessed by a recognition operation [33].

7.6.4 Summary

I studied speech recognition, which adapts its resource demand by reducing its vocabulary size and/or offloading some computation on to remote compute servers. I showed that the local and remote CPU demand of this operation are predictable to within 24%, and that the network demand of shipping data to remote servers is predictable to within 0.5%.

7.7 Other applications

Jason Flinn and SoYoung Park have ported the following applications to the multi-fidelity API:

- LaTeX, a document processor (ported with no source code modification)
- PANGLOSS [70], a language translation system

They provide some further validation of the generality of my programming model; however, I did not evaluate them quantitatively, and do not discuss them in this dissertation.

Chapter 8

Evaluation

Measurement began our might

(W.B.Yeats, Under Ben Bulben, *IV)*

The evaluation of the multi-fidelity runtime system was driven by the following questions:

- Is application adaptation *agile*: does it respond quickly to changes in resource supply?
- Is adaptation *accurate*: does it pick the fidelity that best meets user goals at the current level of resource supply.
- Is it *beneficial*: what is the improvement over the non-adaptive case?

This chapter answers these questions through a series of experiments with GLVU and Radiator, as well as synthetic CPU and memory load generators. The chapter is organised as follows. Section 8.1 describes my evaluation methodology: the base platform, the synthetic applications and background load generators, and the performance metrics used. Section 8.2 evaluates adaptation to varying CPU load in a synthetic application as well as in GLVU. Section 8.3 evaluates adaptation to varying memory load in a synthetic application and in Radiator. Section 8.4 then evaluates system behaviour when GLVU and Radiator run concurrently and compete for resources. Section 8.5 summarizes my main evaluation results.

8.1 Evaluation Methodology

This section describes various components of the experimental setup: the hardware and software platform, the synthetic workloads and load generators, and the evaluation metrics. Real application workloads based on GLVU and Radiator are described later in the chapter.

8.1.1 Experimental Platform

All experiments used the same configuration: an IBM ThinkPad 560X with a 233 MHz Mobile Pentium processor and 96 MB of memory. This hardware is a generation or two behind today's desktop and laptop workstations, and thus representative of the processing power we can expect from lightweight handheld or wearable computers in the near future.

The base OS is a standard, unmodified Linux 2.4.2 kernel. This kernel had several virtual memory management bugs: for memory-related experiments, I used a later kernel version (2.4.17-rmap12a) in which many of these bugs have been fixed.

8.1.2 Synthetic workloads

My evaluation uses two synthetic applications: **cputest** and **memtest**. These applications use the multi-fidelity API to adapt their resource demand to the precise amount suggested by the system: thus, their performance depends entirely on the accuracy and agility of resource supply prediction. Synthetic applications provided us with easily understood microbenchmarks, while experiments with real applications gave us a picture of system behaviour in more realistic scenarios.

cputest is a synthetic, CPU-bound, multi-fidelity "application". Each of its "operations" consists of spinning in a loop for a specified amount of CPU time, measured in cycles. This amount is the single "fidelity" (tunable parameter) p. **cputest**'s utility increases linearly with p: the runtime system must choose the highest value of p that does not violate latency constraints. Thus, the optimal value p_{opt} results in a latency L_{opt} that exactly matches our constraint. To predict the latency for any particular value of p, I use the generic latency predictor (Section 5.5.3)

$$L = \frac{D_{cpu}}{S_{cpu}} = \frac{p}{S_{cpu}}$$

Hence

$$p_{opt} = L_{opt} S_{cpu}$$

L_{opt} is known; thus the accuracy of computing p_{opt} depends entirely on the accuracy of the CPU supply estimate S_{cpu}.

memtest is a memory-bound, synthetic, multi-fidelity application. Its "fidelity" w specifies the optimal working set size for each operation. An operation consists of writing one word to every page in a block of size w and then sleeping for a specified period of time T. On startup, **memtest** preallocates a large block (128 MB) of virtual memory; each operation then touches some initial portion of size w, to emulate a working set of size w. The preallocation avoids the fragmentation and unpredictable access patterns that can result from repeated calls to $malloc$ and $free$.

Since the CPU time spent in touching the pages is negligible, the expected latency is T when all the pages are resident in physical memory. When there is memory pressure, i.e., memory supply is less than w, all pages will not be resident simultaneously, and latency will increase sharply due to paging.

If memory supply decreases sharply while an operation is in flight — e.g., if another application starts up — the system might begin to thrash. The application can only adapt its fidelity at the next adaptive decision point: the start of the next operation. When the system is thrashing, the current operation might take a very long time to complete: it is sometimes better to abort it, and restart at a lower fidelity, rather than incur the penalty of thrashing. To do this, I use a *latency constraint violation callback* (Section 4.3.2), which is triggered when an operation takes much longer than expected: in my experiments, I used a threshold of 10 times the expected latency T. **memtest** then aborts the current operation and starts a new one: the assumption is that the runtime will choose a much lower fidelity for the new operation, given the increased memory pressure in the system.

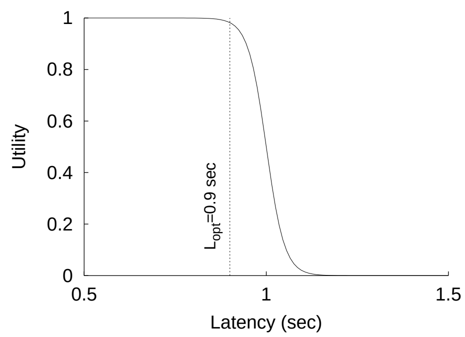
8.1.3 Synthetic load generators

Just as I use **cputest** and **memtest** to mimic adaptive multi-fidelity applications, I use synthetic load generators to mimic the non-adaptive, non-interactive background load in the system. This allows us to run experiments under variable load conditions without the complexity and nonrepeatibility of running real background applications. I use these synthetic load generators to measure the adaptive behaviour of both synthetic and real applications. These load generators are similar in motivation to *host load playback* [24] and *network trace modulation* [72], but simpler: I do not replay traces of real load, but only synthetic ones.

cpuload can emulate any integer-valued processor load average n: it spawns n threads, each of which spins in a tight loop. It takes two parameters: the desired load n, and the desired execution time T. I generated time-varying load patterns with scripts that repeatedly invoke **cpuload** with different parameter values.

memload takes three parameters: the amount of memory M to use, the execution time T, and the refresh period t. The program allocates a contiguous virtual memory block of size M, and dirties each page in this block every t seconds. In my evaluation I use a refresh period of 1 s. On my experimental platform, this is sufficient to keep all the allocated pages active with a low CPU overhead: the initial allocation and page-in costs about 0.05 ms per page (4 KB), and a 1 s periodic refresh of a 64 MB block causes a background CPU load of about 0.002% on an unloaded system. As with **cpuload**, I generated time-varying load patterns through repeated invocations of the program.

I use **cpuload** and **memload** to generate simple square load waveforms. These are simple, repeatable, and easy to analyse. I do not intend this load pattern to capture the complex properties of real workloads. Rather I use it to observe the adaptive system's response under stressful conditions (frequent, sharp transitions). I also use these waveforms to study the effect of frequency and amplitude of load variation on multi-fidelity adaptation.



The graph shows how utility varies with latency, for a latency target of 1 s and a tolerance of 10%. The dotted line shows the optimal latency L_{opt}, 0.9 s in this example.

Figure 8.1: Sigmoid utility function

8.1.4 Evaluation metrics

The primary aim of this thesis work is to improve the performance of interactive applications: to bound the latency of interactive operations, and to reduce the variability in latency. To do this, the system must must match the demand for resources — CPU and memory — to the available supply. The goal is not to minimize demand by running at the lowest available fidelity, but to keep the latency within user-specified constraints without unnecessarily sacrificing fidelity. Too high a latency degrades interactivity; too low a latency indicates an unnecessarily low fidelity. Similarly, aggressive memory usage causes thrashing, while excessive conservatism wastes opportunities to improve fidelity. For this reason, my primary evaluation metric is not the *mean value* of latency or memory demand, but its *deviation* from the optimal value.

The optimal latency L_{opt} for an operation is easily derived from the user-specified latency constraint and tolerance. E.g., a 1 s constraint with a 10% tolerance is expressed as a sigmoid (Figure 8.1) whose tolerance zone is $[0.9, 1.1]$: the utility starts dropping at 0.9 s, and is near zero at 1.1 s. L_{opt} is 0.9 s, the point at which the utility begins to drop sharply.

In a memory-bound application, it is not desirable to aim for the user-specified latency L_{opt}. The effect of memory demand on latency is non-linear: when there is no memory contention, the effect is near zero, and when there is memory contention, latency increases rapidly with small increases in memory demand. The optimal memory demand M_{opt} is at the knee of this curve: the highest possible memory demand that causes no swap activity. This is precisely the quantity that the memory supply predictor (Section 5.5.2) estimates at the start of each operation.

M_{opt} is not known a priori: how then do we know whether our supply predictor estimates it accurately? We can easily detect overestimates by observing the swap activity induced in the system; but underestimates cannot be detected without knowing M_{opt}. Even in the absence of explicit background load (i.e., **memload**), we cannot assume that M_{opt} is equal to the total physical memory in the system: there is always some memory demand from the kernel, the X server, and various background tasks.

In my evaluation, I determine M_{opt} post facto: after all the trials of any given experiment, I find the highest memory demand that the application was able to sustain with no swap activity. I assume that this value is optimal for operations when **memload** is not running. If there was a background load of M_{bg} during an operation, than the optimal value for that operation is $M_{opt} - M_{bg}$.

Agility

We want adaptation to be *agile*: to respond rapidly to changes in resource supply. I measure agility by subjecting the system to *load transitions*, both upwards and downwards, and study the system's behaviour around the transitions. Typically, the system will deviate from its optimal latency or memory demand for some period of time, and then return to normal operation (I define normal operation as "within 10% of the optimal value").

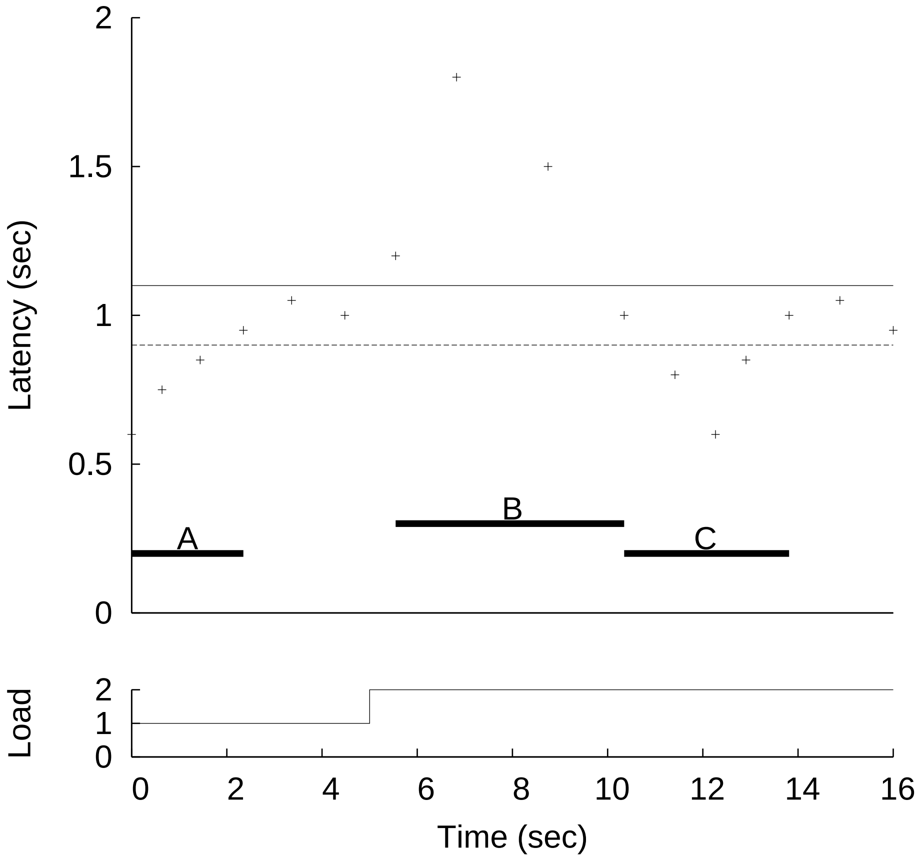
Figure 8.2 shows the latency of a hypothetical application over time. When latency is in the "target zone" (between the two horizontal lines), the system is in "normal operation"; otherwise, it is "adapting". The graph shows three distinct *adaptive phases*:

- *initial convergence* (phase 'A' in the graph): after application startup, the time for which latency is outside the target zone.
- *adapt-down* (phase 'B' in the graph): after an upward load transition, the time for which latency is above the target zone.
- *overshoot correction after adapt-down* (phase 'C' in the graph): after adapt-down, the time (if any) for which latency is below the target zone.

In addition, I define two kinds of adaptive phase not shown in the graph:

- *adapt-up*: after a downward load transition, the time for which latency is below the target zone.
- *overshoot correction after adapt-up*: after adapt-up, the time (if any) for which latency is above the target zone.

To measure agility, I subjected a synthetic application to a simple "up-and-down" background load: no load for some time T, a constant load for time T, and no load again for time T. I then measured the lengths of the adaptive phases that follow the upward and downward load transitions.



The graph depicts pictorially the notion of initial convergence, adapt-down, and overshoot correction phases: it does not represent the measured performance of any real application or system. The bottom graph shows the background load, which increases from 1 to 2 at T=5 s. In the top graph, the points represent operation latency over time. The horizontal lines at heights 0.9 and 1.1 demarcate the target zone. The three segments A, B, and C, correspond to the initial convergence, the adapt-down, and the overshoot correction periods respectively.

Figure 8.2: Adaptation phases for a hypothetical application and system

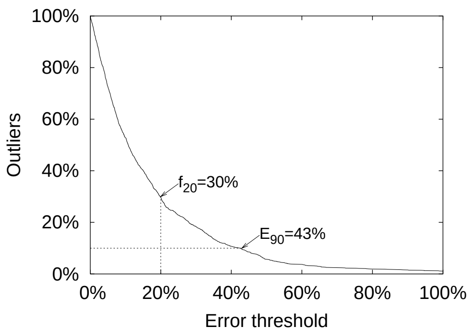


Figure 8.3: Example of deviation plot

Accuracy

To measure the accuracy of adaptation, I subjected adaptive applications — GLVU and Radiator — to a time-varying background load (CPU or memory). I then measured the latency or memory demand of each operation, and compared it to the optimal value. For CPU-bound operations, I measured the deviation of latency from L_{opt}; for memory-bound operations, that of memory demand from M_{opt}. I report these measurements in three ways:

- a timeline graph of one representative run of the experiment, showing latency (or memory demand) of the adaptive application over time, as well as the background CPU or memory load.
- a histogram of latency or memory demand over all the operations in 5 runs of the experiment.
- the outlier distribution graph, the bad deviation frequency f_{20}, and the common-case deviation E_{90} over 5 runs. These show the frequency f with which deviation from the optimal resource demand exceeds some threshold E, as a function of E. E is the unsigned relative error. E.g. the *latency deviation*

$$E_{latency} = \frac{|L - L_{opt}|}{L_{opt}}$$

  where L is the measured latency and L_{opt} is the optimal value. f_{20} is the frequency with which we see an E exceeding 0.2 (20%); E_{90} is the error threshold that bounds 90% of observations. These metrics are exactly analogous to those used for predictor accuracy (Section 6.5). In the latter case, I measured the deviation of predicted values from observed values; here, I measure the deviation of observed values from the optimal value. Figure 8.3 shows an example of a deviation plot, and the intercepts corresponding to f_{20} and E_{90}.

Benefit of adaptation

To evaluate the benefit of adaptation, I compared the behaviour of the adaptive application under time-varying load to that of the same application running without adaptation. I report the outlier distribution, the bad deviation frequency, and the common-case deviation for both configurations.

In the non-adaptive case, the application runs at a fixed fidelity. In some cases, this is the maximum possible fidelity: e.g. if GLVU does no multiresolution scaling at all, then its resolution is 1. In other cases, full fidelity is not a viable option: e.g., running radiosity at fidelity 1 would require far more memory than is available on my test platform. In this cases, I chose a reasonable value for fidelity: the chosen values are not necessarily optimal for any given experiment, but they tell us what we can realistically expect from a non-adaptive approach.

8.2 Adaptation to CPU load

8.2.1 Agility

To measure the agility of the CPU supply predictor, I measured the operation latency of **cputest** running a large number of consecutive operations. The background load was generated by **cpuload**: 30 s of no load, 30 s of a load of 1 (i.e., one CPU-bound process), and 30 s of no load again. I set a latency constraint of 1 s with a 10% tolerance, which means the optimal latency L_{opt} is 0.9 s.

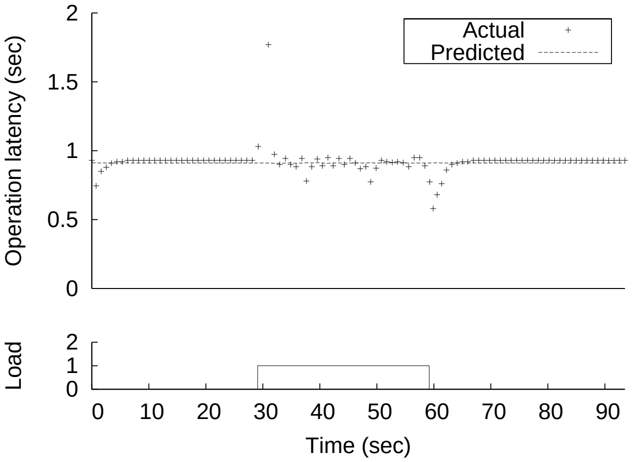
Figure 8.4 shows the results of this experiment. Figure 8.4(a) shows the operation latency over time for a single 90 s run. We see the initial convergence, adapt-down, and adapt-up phases, but no overshoot. During adapt-down, one operation suffered a large increase in latency: this was the operation in flight when the load transitioned. We also see that when there is background load, the latency constraint is maintained, but the behaviour is noisier. This is caused by the granularity of CPU scheduling in Linux: competing applications receive quanta of 200 ms at a time, and so **cputest**'s CPU-share over a 1 s period, when the background load was 1, could vary between 45% and 55%. When the system was unloaded the CPU share was constant (close to 100%) and the latency constraint was maintained exactly.

Figure 8.4(b) shows the length of each adaptive phase, averaged over 5 runs. Figure 8.4(c) shows how the operation latencies are distributed. We see that the vast majority are around 0.9 s, as expected: deviations occur only at and around load transitions. Figure 8.4(d) shows the outlier frequency distribution. 10% of the time, the error is above 9.2% (E_{90} is 9.2%); 5.4% of the time, it is above 20% (f_{20} is 5.4%); and 0.8% of the time, it is more than 40%. This 0.8% corresponds to the single operation per run that was in flight during the upward load transition.

8.2.2 Accuracy: CPU adaptation in GLVU

In order to measure the accuracy and effectiveness of adaptation in GLVU, I used a trace of a user navigating the "Notre Dame" scene (Section 7.3.3). The background load, generated by **cpuload**, was a square waveform: 10 s periods of no load alternating with 10 s periods of load 1. I used a utility function that increases linearly with fidelity, and a latency constraint of 1 s$\pm$10%, which gives us a target latency (L_{opt}) of 0.9 s. These values represent the *baseline* case: later in this section I show the effects of varying the input scene, target latency, load transition frequency, and peak load.

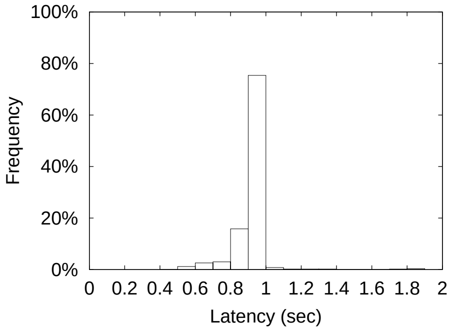
Figure 8.5 shows the results of the baseline experiment. Figure 8.5(a) shows application performance over time for one representative run: the fidelity chosen for each operation, its latency, and the background load. We see that that most operations had a latency close to optimal (0.9 s), with some deviation due to variation in CPU supply and demand. Fig-



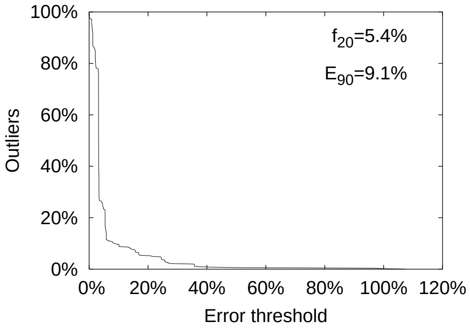
(a) Timeline

Type of adaptation	Time		Operations	
Initial convergence	2.0 s	(0.8 s)	2.4	(1.1)
Adapt-down	2.3 s	(0.9 s)	2.0	(0.7)
Overshoot correction after adapt-down	0.0 s	(0.0 s)	0.0	(0.0)
Adapt-up	2.8 s	(0.4 s)	3.8	(0.4)
Overshoot correction after adapt-up	0.0 s	(0.0 s)	0.0	(0.0)

(b) Adaptation times



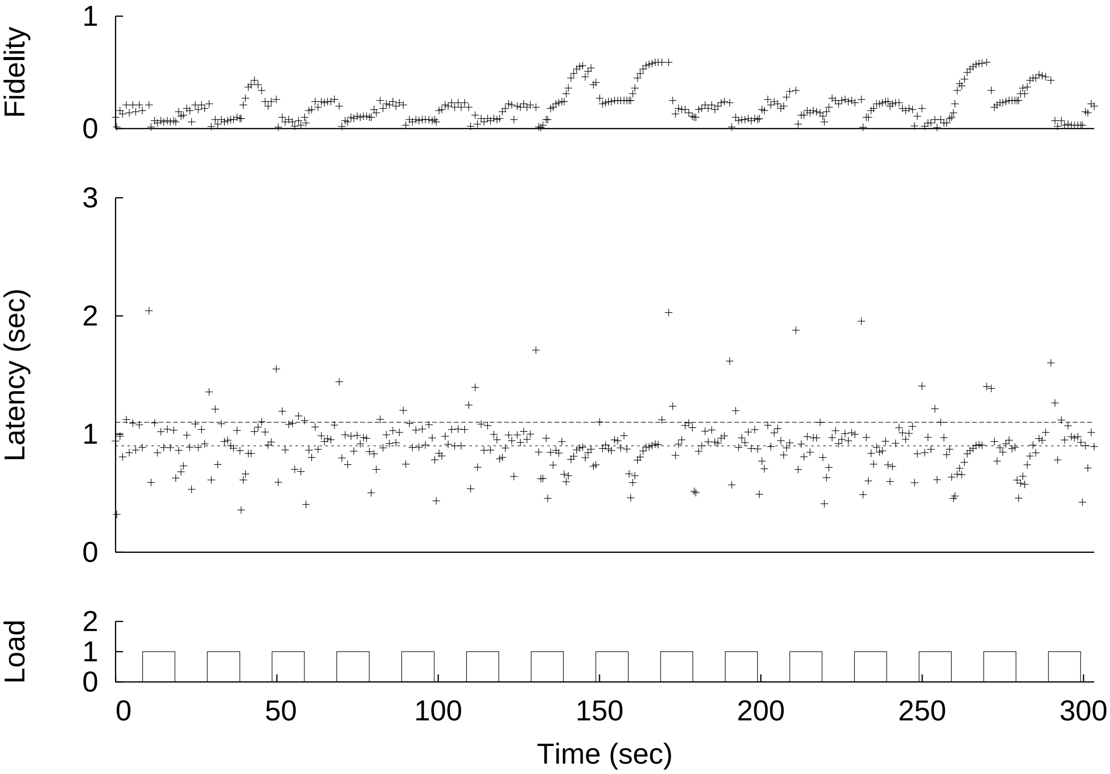
(c) Latency distribution



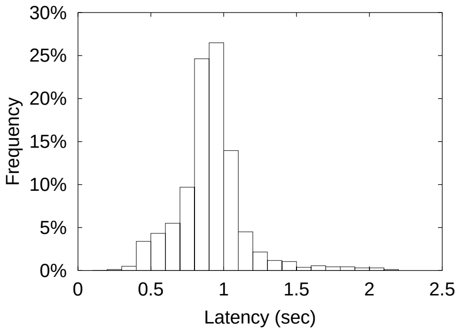
(d) Outlier distribution

The timeline shows latency and background load for one run of **cputest**. The table shows the mean length (and standard deviation) of the different adaptive phases, over 5 runs. The bottom graphs show the latency distribution and the deviation from the target (0.9 s), for 5 runs.

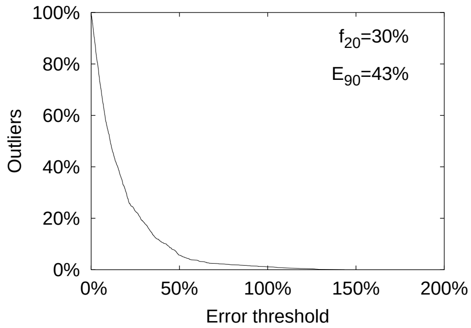
Figure 8.4: Agility of adaptation to CPU supply



(a) Timeline



(b) Latency distribution



(c) Outlier distribution

The top graph shows fidelity, latency and background load for one run of the "Notre Dame" user trace. The lower horizontal line shows the optimal latency (0.9 s) and the higher one shows the upper limit of the latency constraint (1.1 s). The bottom graphs show the latency distribution and the deviation from the optimal latency (0.9 s) over 5 runs.

Figure 8.5: Adaptation in GLVU

ure 8.5(b) shows the distribution of operation latencies aggregated over 5 runs of the experiment: we see that most of the deviation is on the low side of L_{opt}. Figure 8.5(c) shows the outlier distribution: 30% of the operations deviated from the target by more than 20% (f_{20} is 30%) and only 10% by more than 43% ($E90$ is 43%).

Benefit of adaptation

In order to measure the benefit of adaptation, I compared the results of the same experiment with three different configurations:

- *no adaptation*: GLVU did not adapt at all, but always rendered the scene at full resolution, i.e., with fidelity 1.
- *demand adaptation only*: GLVU adapted its CPU demand to meet the 1 s constraint, but ignored the variation in supply
- *supply and demand adaptation*: this is the baseline case from the previous section.

Figure 8.6 shows the results of these experiments: for each configuration, I show the timeline from one run and the latency distribution over 5 runs. With no adaptation at all, there was a very wide variation in latency: moreover, the application never met its latency constraint, but always had a latency above 1 s. With demand-only adaptation, the application was able to limit each operation to 1 s of CPU time. This gave us the desired latency when the system is unloaded, but twice the desired latency under a load of 1. When the system adapted to both supply and demand, latency was clustered around the optimal 0.9 s. Note that with a closed-loop workload, high latencies also result in a longer time to completion: thus the non-adaptive case took 1000 s, the demand-only case 400 s, and the fully adaptive case 300 s to execute the same workload.

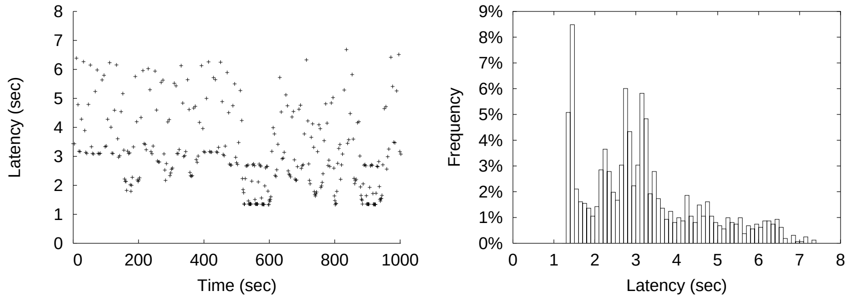
Figure 8.7 shows the outlier distributions for 5 runs of each of the three cases: we see that both supply and demand adaptation contribute to reducing deviation from the latency constraint.

Variation with input data

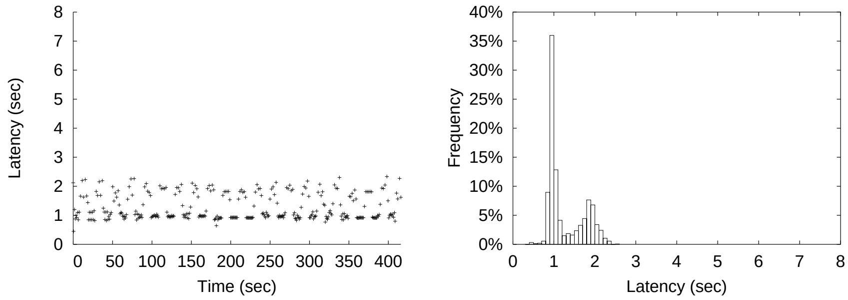
Figure 8.8 shows the outlier distribution for GLVU running a user path trace on four different scenes, with 5 experimental runs per scene. We see that the distribution is similar in all cases: the system maintains its latency constraint equally well across the different scenes.

Variation with target latency

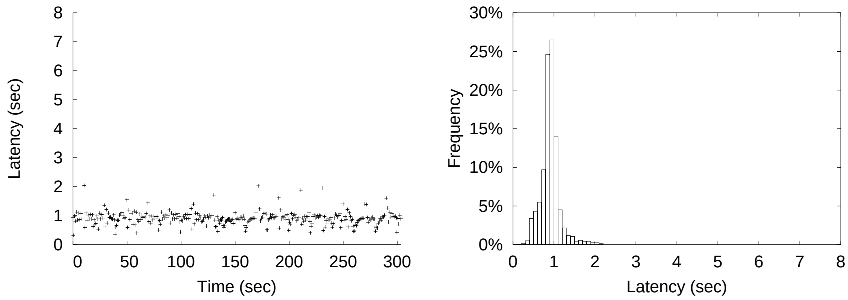
When the application's latency constraint changes, the time scale at which the system adapts, and thus its adaptive behaviour, also change. Figure 8.9 shows the outlier distribution for three different target latencies in addition to the baseline 1 s case: 0.25 s, 0.5 s,



(a) No adaptation



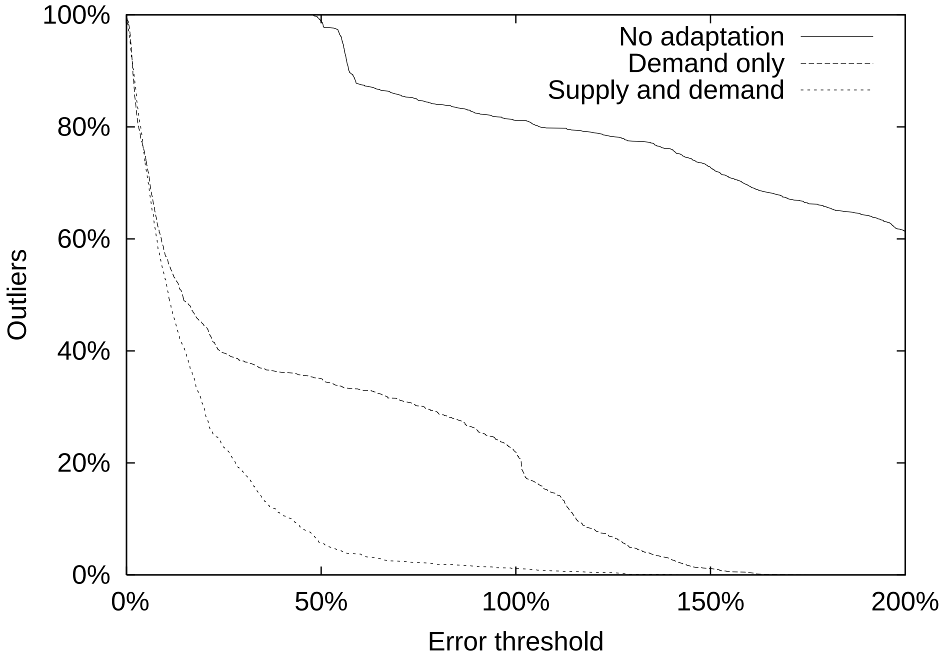
(b) Demand adaptation only



(c) Supply and demand adaptation (baseline)

The left-hand graphs show latency during one run of the "Notre Dame" trace. The right-hand graphs show the latency distribution over 5 runs. The graphs show three different configurations: no adaptation, demand adaptation only, and both supply and demand adaptation.

Figure 8.6: Latency in GLVU: with and without adaptation

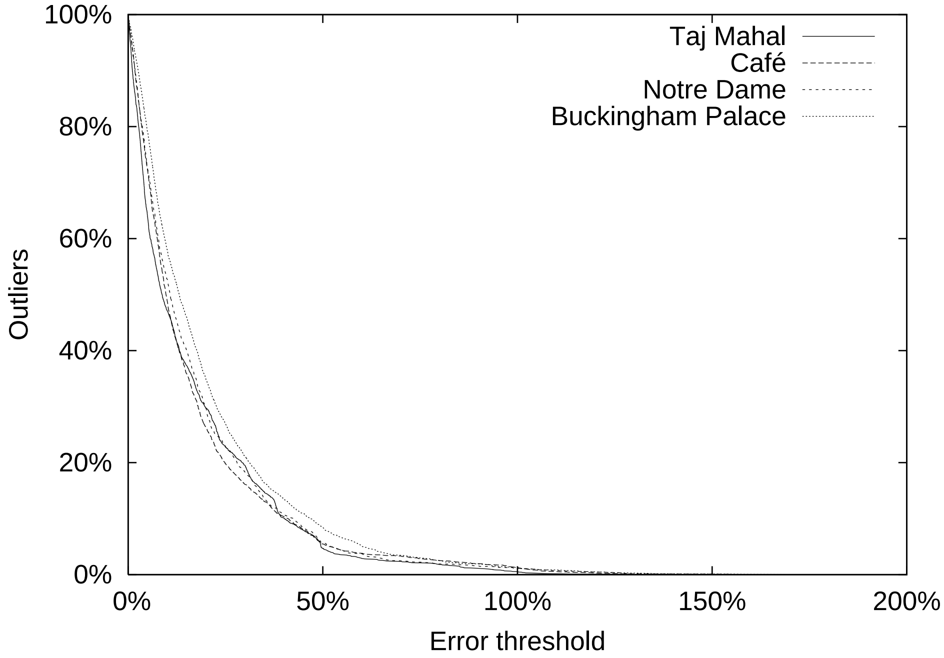


(a) Outlier distribution

Configuration	f_{20}	E_{90}
No adaptation	100%	494%
Demand adaptation only	44%	116%
Supply and demand adaptation	30%	43%

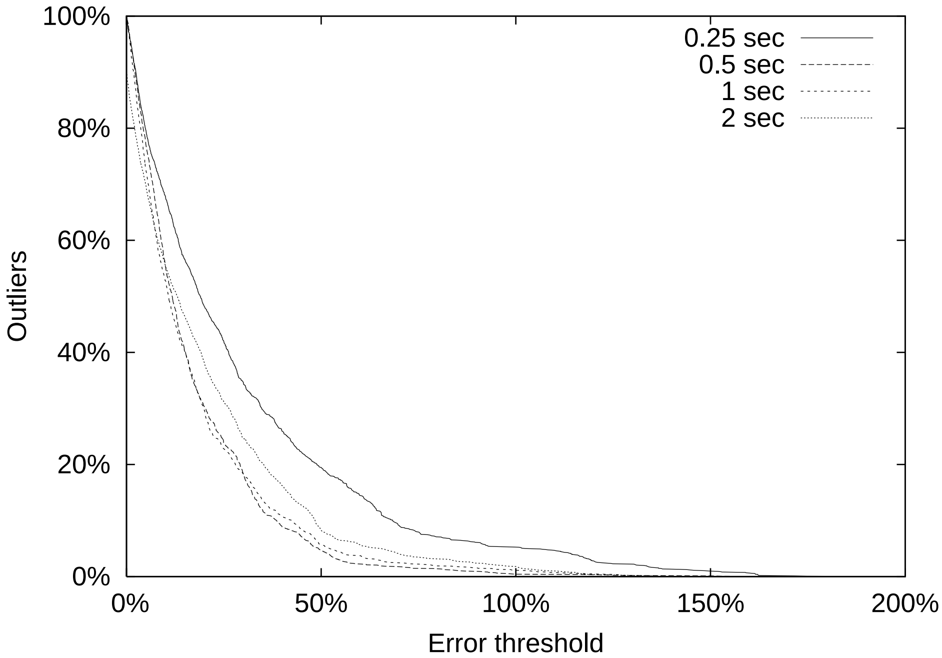
(b) Bad deviation frequency and common-case deviation

Figure 8.7: Latency outliers in GLVU: with and without adaptation



Scene	Number of polygons	f_{20}	E_{90}
Taj Mahal	127406	30%	40%
Café	138598	26%	41%
Notre Dame	160206	30%	43%
Buckingham Palace	235572	35%	47%

Figure 8.8: Outlier distribution for different 3-D scenes in GLVU



Latency constraint	f_{20}	E_{90}
0.25 s	48%	68%
0.5 s	30%	39%
1.0 s	30%	43%
2.0 s	38%	48%

For the 2 s latency constraint, 32.8 operations on average (out of 324) were at full fidelity but had a latency lower than the target 1.8 s: the latency of these is counted as 1.8 s.

Figure 8.9: GLVU outlier distribution for different target latencies

and 2 s, corresponding to L_{opt} values of 0.225 s, 0.45 s, and 1.8 s respectively. These values span the range of reasonable fidelities and interactive response times for GLVU on the test platform.

Performance was best at the 0.5 s and 1 s time scales: both higher and lower values increased the amount of deviation:

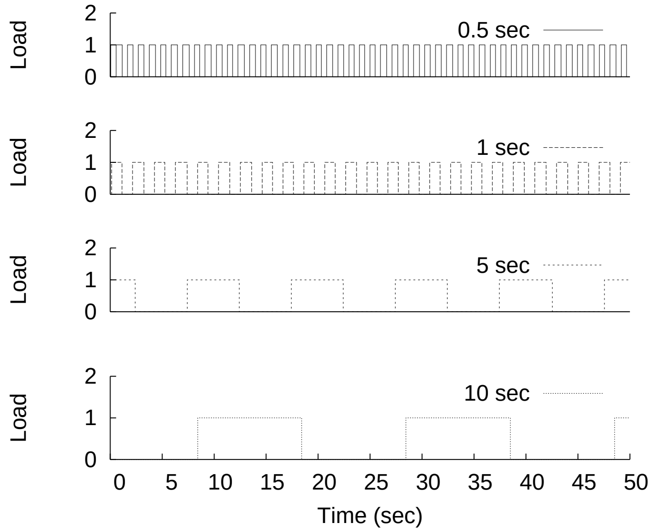
- At time scales below 0.5 s, the granularity of processor scheduling causes deviation. On Linux, CPU-bound processes get quanta of 0.2 s at a time. When there is a background load of 1, GLVU gets 50% of the CPU on average, but in any given 0.2 s period, it may receive anything from 0% to 100%.
- At time scales above 1 s, load transitions appear more frequently relative to the workload. For 1 s operations, and a load that varies every 10 s, about 10% of the operations are hit by a transition, and miss their target latency. With a 2 s target, twice as many operations are affected. At time scales much larger than the load period (10 s), we would expect deviation to decrease again: each operation would see multiple transitions, and an average load of 0.5. I.e., the variation would be smoothed out over the course of the operation. However, such large time scales are not meaningful for this application.

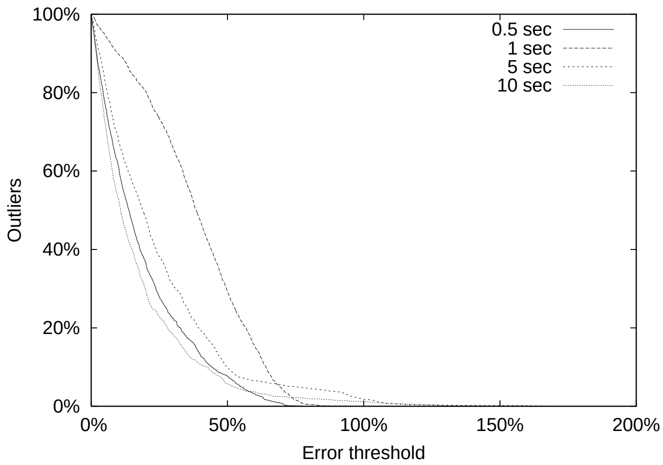
For the 2 s case, I found some operations that took less than the optimal 1.8 s, but were at full fidelity. These operations did not sacrifice fidelity unnecessarily, and thus do not represent a deviation from optimal behaviour; in the latency and outlier distributions, I report the latency of such operation as the optimal value (1.8 s).

Variation with background load

We have already seen that the time scale, i.e., frequency, of background load variation has an effect on adaptation accuracy. I measured the adaptation accuracy across a range of load frequencies: I used the same square-wave load pattern, but varied the time period between load transitions (Figure 8.10). We see that as this time period decreases, adaptation accuracy gets worse, as the probability of a transition during an operation increases. It is worst at 1 s: a load varying at the same frequency as the adaptive decision-making causes the most interference. With a further decrease to a 0.5 s time period, accuracy improves significantly: now the effect of load variation is smoothed out over the course of a single operation.

I also studied the effect of peak load on adaptation: i.e., I kept the square-waveform with a 1 s period, and varied its amplitude (Figure 8.11). As expected, larger transitions led to worse adaptation: in the “10 process” case (i.e. background load switches between 0 and 10 every second), the common-case deviation was 89%.

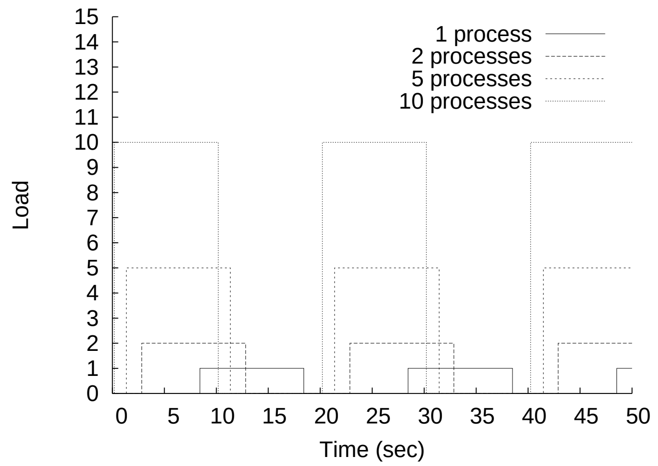


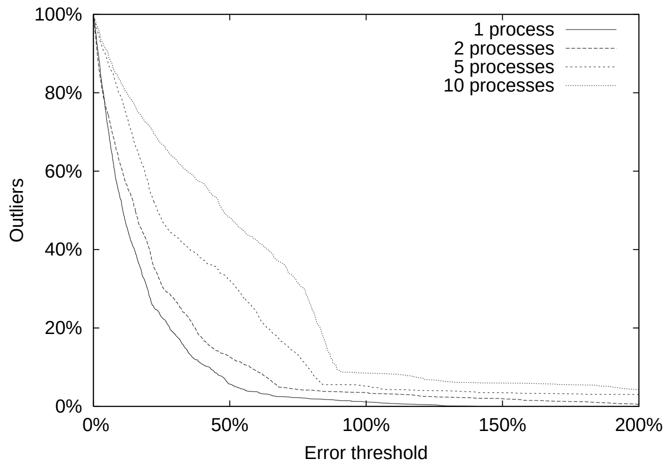


Period between load transitions	f_{20}	E_{90}
0.5 s	37%	44%
1.0 s	80%	64%
5 s	48%	50%
10 s	30%	43%

The top graph shows the different background load patterns being compared; the bottom graph and table show the latency deviation for each of the load patterns.

Figure 8.10: GLVU outlier distribution for different load transition frequencies





Peak background load	f_{20}	E_{90}
1 process	30%	43%
2 processes	41%	57%
5 processes	57%	79%
10 processes	72%	89%

The top graph shows the different background load patterns being compared; the bottom graph and table show the latency deviation for each of the load patterns.

Figure 8.11: GLVU outlier distribution for different peak loads

8.2.3 Summary

I measured the agility of CPU adaptation using a synthetic workload and a simple "up-and-down" background load; I found that the system responded to changes in load within 2-3 s. I also evaluated the accuracy of CPU adaptation in GLVU, by evaluating the deviation from the target latency in various scenarios. In the baseline case, I found that adaptation reduced latency deviation considerably — the common-case deviation was reduced from 494% in the non-adaptive case to 43% in the adaptive case — and that both supply and demand prediction were necessary to achieve this reduction. I repeated this evaluation with varying parameter values, and found that

- latency deviation was independent of the 3-D scene used.
- deviation was highest when the time scale of load variation matches that of adaptation: very rapid or very slow load variation had little negative impact.
- latency deviation increased with the magnitude of load variation: it was 43% when the load varied between 0–1, and 89% when it varied between 0–10.

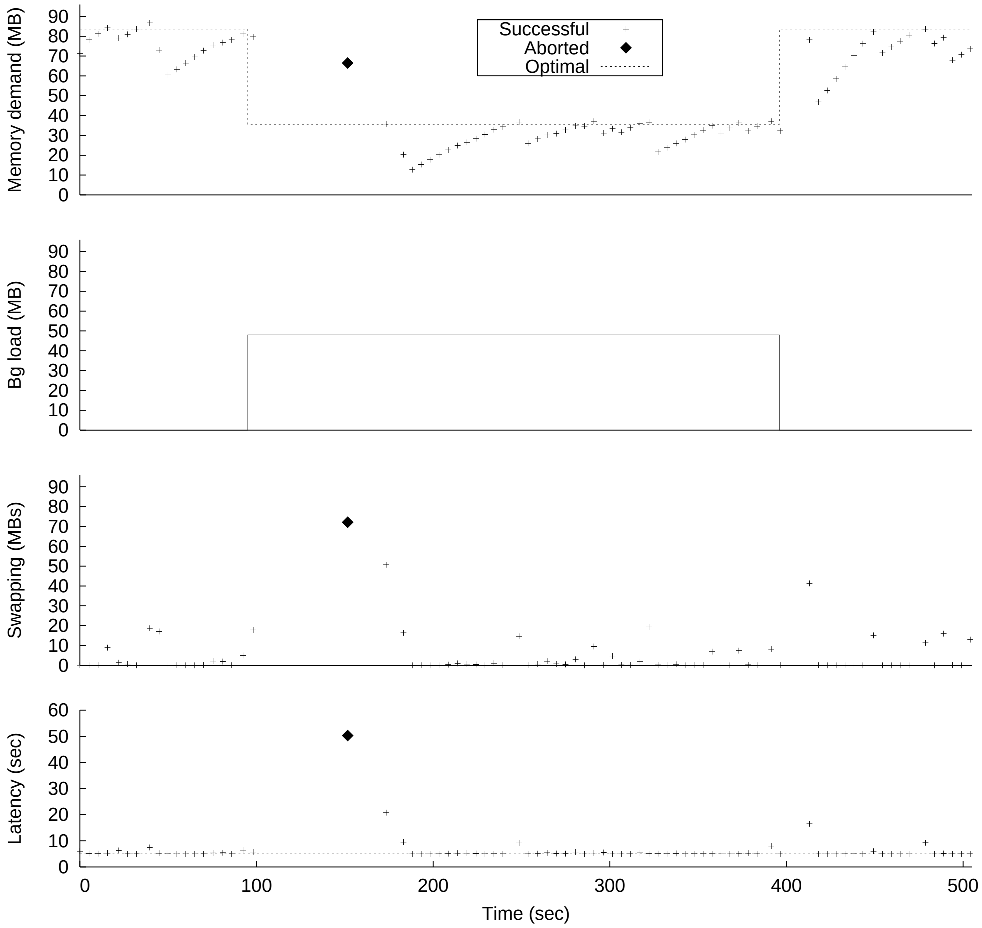
8.3 Adaptation to memory load

8.3.1 Agility

I measured the agility of memory supply prediction by subjecting **memtest** to sharp upward and downward transitions in background memory load. My measurements for memory were on a time scale of *tens of seconds* rather than seconds. Every second, the Linux VM updates the ages of $1/32$ of the system's pages (Section 5.5.2): thus it takes tens of seconds to reflect changes in load. Since the multi-fidelity runtime uses statistics exported by the kernel to guide adaptation, it cannot adapt effectively at time scales smaller than this.

Each operation in **memtest** consists of writing to every page in a block of size p — where p is a tunable parameter whose value is computed by the multi-fidelity system — and then sleeping for 5 s. Thus the expected latency for an operation with no memory contention is just over 5 s. To prevent memory contention from delaying an operation indefinitely, I used a latency constraint violation callback: any operation taking more than 50 s was aborted. I subjected **memtest** to an "up-and-down" load pattern: no load for 100 s, a load of 48 MB (half the total system memory) for 300 s, and no load again for 100 s. In order to start from a known VM state each time, each experimental run was preceded by a very heavy memory load to flush all unused pages from the system, followed by a 10 s wait for the system to quiesce.

Figure 8.12(a) shows the behaviour over time of one run of this experiment. We see a TCP-like back-off behaviour: the application ramps up its memory usage up to the optimal value M_{opt} (84 MB without **memload**); when it exceeds M_{opt}, it experiences memory contention, backs off rapidly, and then begins to ramp up again. When the background



(a) Time line

Type of adaptation	Time		Operations	
Initial convergence	6.2 s	(2.3 s)	1.2	(0.4)
Adapt-down	60.8 s	(28.5 s)	1.8	(0.4)
Overshoot correction after adapt-down	61.7 s	(20.2 s)	11.8	(3.7)
Adapt-up	22.0 s	(8.4 s)	2.4	(1.9)
Overshoot correction after adapt-up	0.0 s	(0.0 s)	0.0	(0.0)

(b) Adaptation times

The graph shows memory demand, background load, system swap activity,and latency for one run of **memtest**. The table shows the mean length (and standard deviation) of the different adaptation phases over 5 runs.

Figure 8.12: Agility of adaptation to memory supply

load increases sharply to 48 MB, the operation in flight suffers heavy memory contention, exceeds the latency callback threshold, and is aborted. **memtest** then adapts downward and resumes the TCP-like oscillation around the new M_{opt} of 36 MB. When the load is removed, it ramps back up to the original memory demand. Note that operation latency increases with system swap activity, which is my measure of memory contention. Swapping of **memtest**'s own pages increases the number of times it must stall on a disk access; swapping of other process pages increases the length of each stall by introducing contention for the disk.

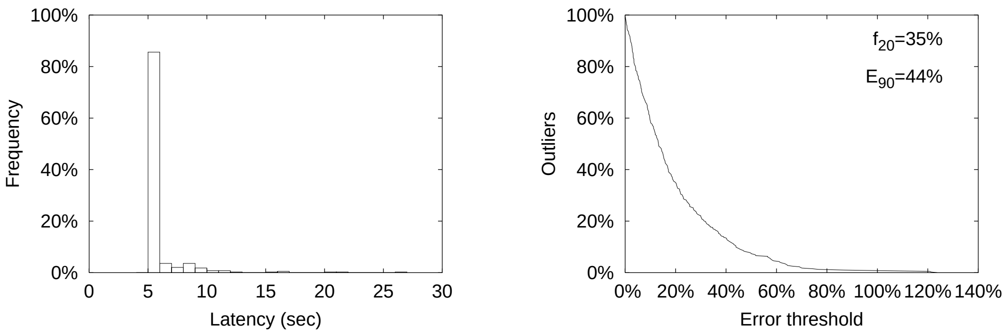
Figure 8.12(b) shows the length of each adaptation period, averaged over 5 runs. All of them are on the order of tens of seconds, except for the overshoot correction after adapt-up: when the system adapts fidelity upwards it backs off before overshooting the optimal value.

- when adapting upwards, the system rely on the free and inactive page counts exported by the VM. Inactive page counts are updated over a 32 s period. Free page counts are updated immediately, but free pages are created only on demand, i.e., when the free page pool becomes too small. Thus as each subsequent operation uses more memory, the system responds by creating a small number of free pages, and requires many operations to ramp up to the optimal value.
- adapt downwards in response to a load spike only takes a few operations. However, because of memory contention, these few operations take a long time. This time can be bounded using latency callbacks and aborts: in fact, the adapt-down time (60 s) is dominated by the latency of the single aborted operation (50 s). For **memtest**, I could have used a more aggressive (lower) callback value; however, in a real application an excessively aggressive value would lead to false callbacks, if the operation took longer than expected for reasons other than memory pressure.

Figure 8.13(a) shows a histogram of latency distribution averaged over 5 runs: most operations take 5 s, with a tail that represents the operations affected by swapping activity. Figure 8.13(b) shows the corresponding outlier distribution: the deviation of memory demand from the optimal value M_{opt} (84 MB when unloaded, 36 MB when loaded). f_{20} is 35%, i.e., 35% of operations deviated by more than 20%. From the latency distribution, we can infer that the system usually errs on the side of caution: most operations do not suffer memory contention, and have a latency of 5 s. I.e., most operations occur during a ramp-up phase, rather than a back-off.

8.3.2 Accuracy: Memory adaptation in Radiator

To measure the accuracy of memory adaptation in Radiator, I ran radiosity repeatedly on a single scene with a latency constraint of 10 s. With this constraint, on the test platform, progressive radiosity is always a better choice than hierarchical radiosity. The latter has a very high CPU demand, and is only useful on a very fast processor, or if the user can tolerate very high latencies. In all of my experiments, the multi-fidelity runtime chose progressive



(a) Latency distribution (b) Memory outlier distribution

The graphs shows the distribution of operation latency (left), and the deviation from optimal memory demand (right) over five runs of **memtest** subjected to a single upward and a single downward load transition.

Figure 8.13: Agility of memory adaptation: latency distribution and deviation from M_{opt}

radiosity for every single operation; in the remainder of this section I use "Radiator" and "radiosity" to signify progressive radiosity alone.

My method for measuring each operation's memory demand — its working set — is based on measuring its resident set size (Section 5.6.1), and is not valid when there might be memory contention. For this reason, the memory demand measurements presented here are based on the values *predicted* by the demand predictor for each operation, given its fidelity: we already know from controlled experiments that these values are extremely accurate (Section 7.4.3).

My benchmark only approximates a realistic use of radiosity: a real user would only re-run radiosity if she modified the scene in some way. Unfortunately, Radiator does not include scene editing functionality, which prevents us from modifying the scene on the fly. The effect of modifying the scene would be to introduce additional error into the resource demand predictions. However, this effect is certain to be small: Radiator's memory demand is predictable with low error (1.3%) even with a single generic predictor across multiple scenes, and any error induced by modifications to a single scene will be smaller than this. Since memory demand is highly predictable, I am really evaluating the accuracy of memory *supply* prediction, and its effect on Radiator.

I used a square-wave background memory load switching between 0 and 48 MB every 200 s; the entire experiment takes 1200 s, i.e. 3 cycles of load variation. These large time periods are determined by the large time constants of the underlying operating system. The Linux VM's page aging algorithm updates the age of each memory page once every 32 s, and any interesting VM behaviour can only observed after several iterations over the entire list of pages.

As with **memtest**, I used a latency constraint violation callback to abort operations suffering from excessive memory contention, i.e. thrashing. A low callback threshold limits the time and resource consumed by an aborted operation. However, too low a threshold will cause unnecessary aborts, due to underestimates of CPU demand rather than memory contention. The CPU demand prediction error for radiosity is well under 50% in the common case (Section 7.4.2): thus, any misprediction of latency by a factor of 2 or greater, is almost certainly due to memory contention. Hence I set the latency callback threshold to 20 s: twice the target latency.

When Radiator receives a callback, it must clean up the state created by the aborted operation before it can begin a new one. In such a complex application, the cleanup requires many memory accesses, which aggravates the memory contention. A cheap yet clean abort mechanism would require considerable changes to the source code. Instead, I chose the simpler approach of killing and restarting the Radiator process on abort. This is a feasible approach for Radiator because it only has *soft state*, which can be regenerated by reading the input data file and re-executing the radiosity computation.

Figure 8.14 shows the results of one experimental run of Radiator and **memload**. When there is no load, Radiator's fidelity is close to 0.08, corresponding to a memory demand of 55 MB. This is significantly less than the memory available in the system: Radiator's fidelity is limited by the 10 s latency constraint: it is CPU-bound rather than memory-bound.

When the background load increases to 48 MB, Radiator becomes memory-bound, and must adapt its fidelity downwards. We see the expected pattern:

- The operation in flight when the load increases suffers heavy memory contention, and is aborted after 20 s.
- Radiator then adapts its memory demand down to 35 MB.
- It corrects the overshoot, and converges on 40 MB$\pm$5 MB: note that the oscillation is larger than in the no-load case.
- When the load drops, Radiator rapidly adapts upwards back to the 55 MB level, at which point it is CPU-bound again.

I ran the same experiment with Radiator in a non-adaptive configuration, i.e., with a fixed fidelity and with latency callbacks disabled. In the adaptive case, the optimal fidelity was 0.09 in the absence of memory load, and 0.02 when loaded: thus, I used 0.05 as a good compromise value. Figure 8.15 shows the performance of non-adaptive Radiator. We see that during the unloaded or CPU-bound phases, there is no memory contention, and latency is around 5 s, well short of the 9 s optimum: the static fidelity is too conservative in this case. When there is load, the application is memory-bound, and the same fidelity is now too aggressive: the application suffers heavy memory contention and latencies well above 20 s. If I had enabled the latency callbacks in the non-adaptive case, almost every

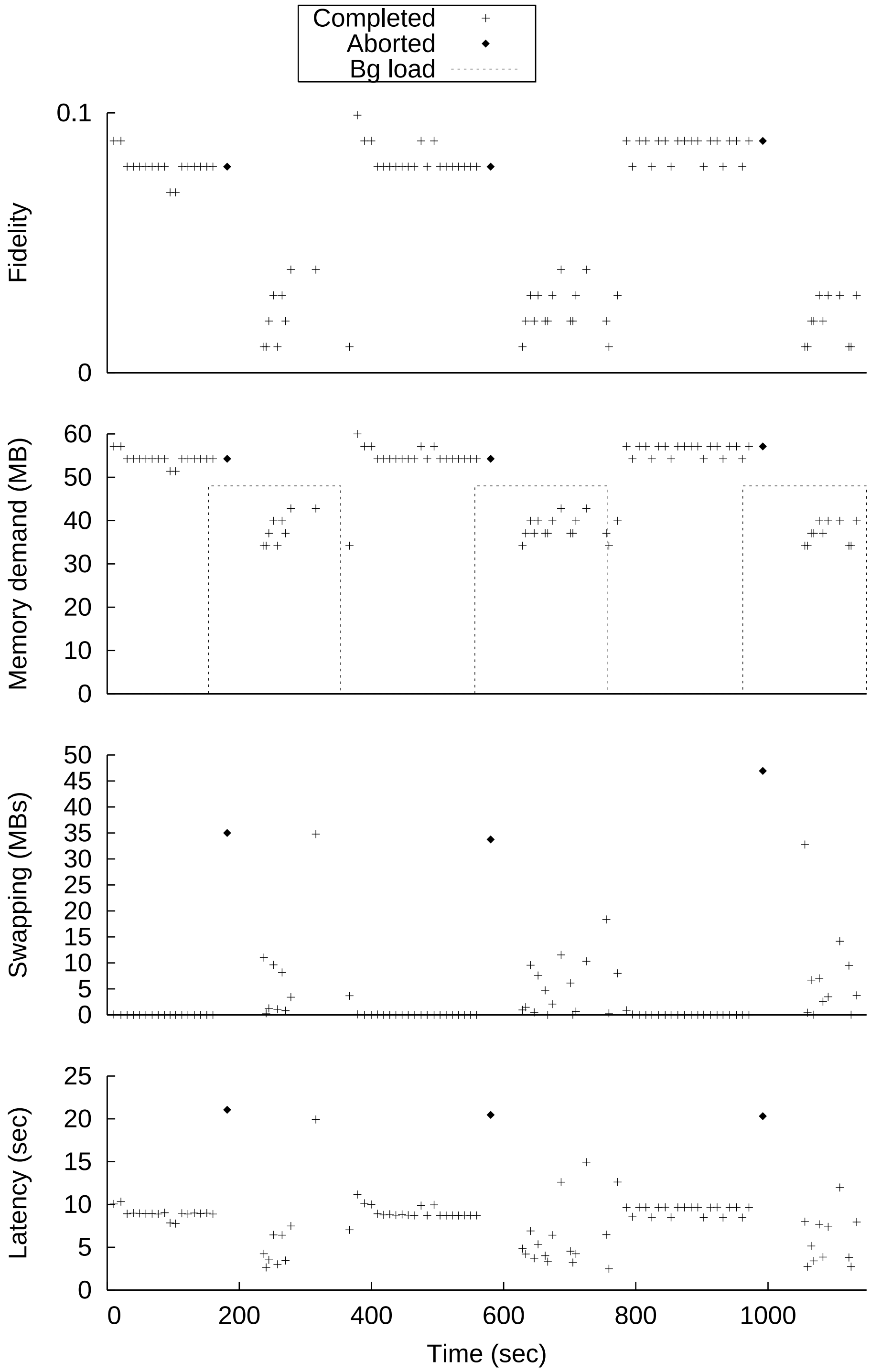


Figure 8.14: Adaptation in Radiator under varying memory load

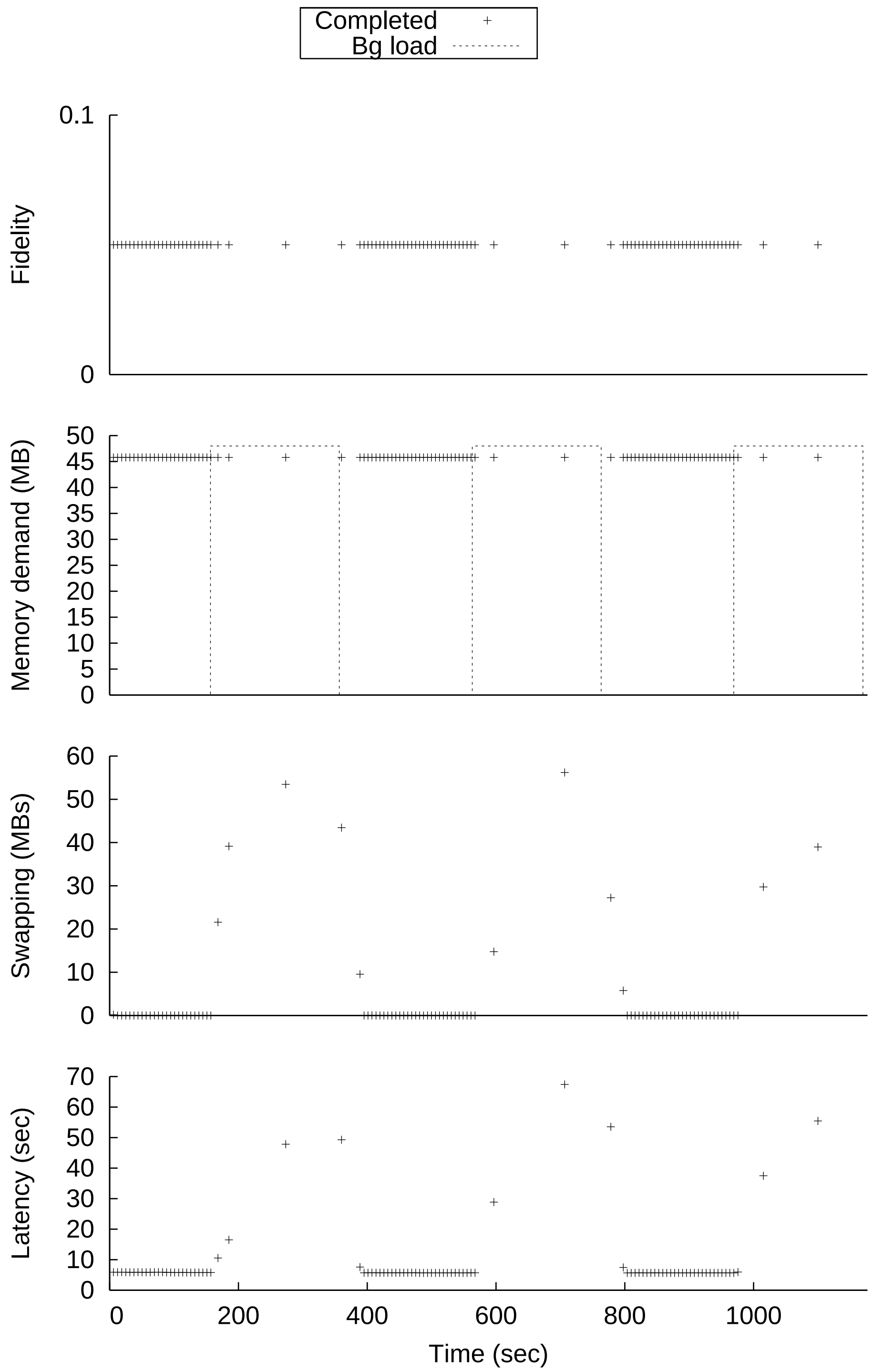


Figure 8.15: Radiator with no adaptation under varying memory load

operation executed under load would have been aborted; in the adaptive case, only the operation that had already commenced when the load increased, is aborted. Note also that in the non-adaptive case, the total latency of operations executed under load is significantly lower than the length of the corresponding load period: memory contention prolongs the inter-operation intervals (which are normally extremely short) as well as the operations themselves.

Radiator thus alternates beween two phases: CPU-bound and memory-bound. In order to evaluate the accuracy of adaptation, I consider the two phases separately. When Radiator is CPU-bound, we expect latency to be close to L_{opt} (9 s). When memory-bound we expect memory demand to be close to M_{opt} (39 MB for this experiment), and latency to be less than L_{opt}.

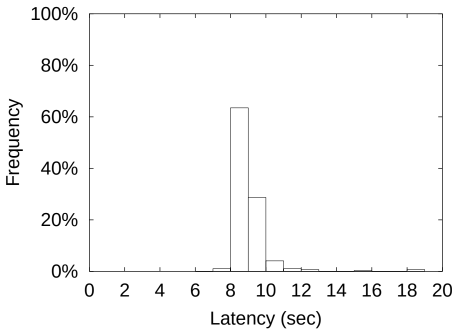
Figures 8.16(a) and 8.16(b) show the latency distribution during the CPU-bound phases, for the adaptive and non-adaptive configurations respectively. In the former case latency is usually around 9 s; in the latter, almost always 5 s. Figure 8.16(c) shows the deviation of these latencies from the optimal. When adaptive, latency is close to optimal (usually within 10%); when not adaptive, it is consistently at 40% under the optimal 9 s, indicating an unnecessary sacrifice of fidelity (0.05 rather than 0.09). Figure 8.16(d) shows the corresponding values of f_{20} and E_{90}; it also shows the average number of successful operations per experimental run, and the number of aborted operations in the adaptive case. The non-adaptive case completed a larger number of operations due to its lower fidelity and correspondingly lower latency. There were no aborts in the adaptive case when CPU-bound, as there was no memory contention.

Figures 8.17(a) and 8.17(b) show the latency distributions during the memory-bound phases. In the adaptive case, latency is usually under 9 s. Sometimes we see higher latencies due to CPU mispredictions, or to the after-effects of a recently concluded memory-bound phases (the system continues to swap for a while even after the background memory load is removed). The graph does not show aborted operations: these have a latency of 20 s each, and there is one for each upward transition in load (i.e. 3 per experimental run). In the non-adaptive case, the fixed fidelity leads to a fixed memory demand of about 48 MB, which exceeds the memory supply and causes memory contention and high latencies.

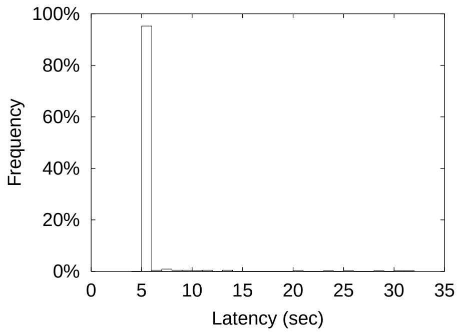
Figure 8.17(c) shows the deviation from M_{opt} in the two configurations. When adaptive, memory demand is usually within 10% of optimal; when not adaptive, it is consistently 25% above optimal. Figure 8.17(d) shows the corresponding f_{20} and E_{90}, and also the number of successful and aborted operations. By adapting memory demand, the adaptive case was able to bound latency, and complete a larger number of operations, at the cost of reduced fidelity, and one aborted operation per upward transition in memory load.

Costs and benefits of aborting

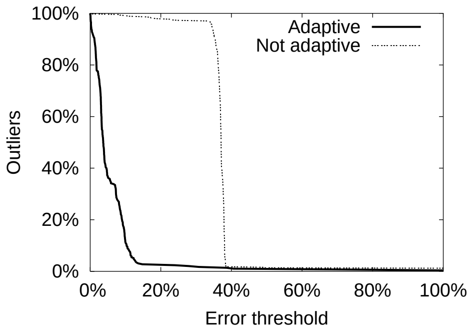
In the preceding experiments, we saw that a sharp increase in memory load during an operation causes severe memory pressure: my solution was to abort the operation after a



(a) Latency distribution: adaptive



(b) Latency distribution: not adaptive

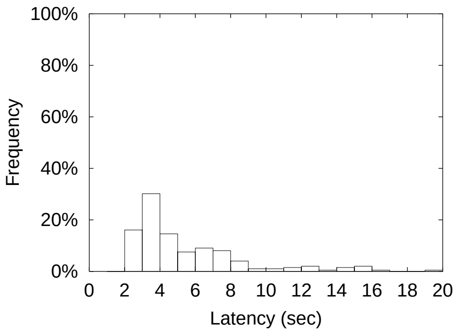


(c) Latency outliers: adaptive and non-adaptive

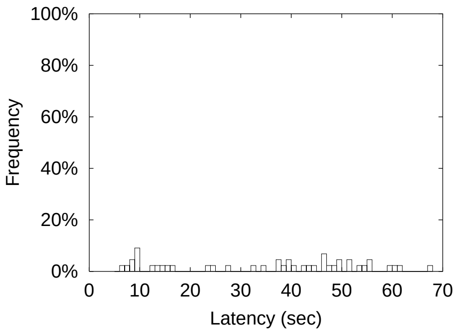
Configuration	Latency deviation		Successful ops	Aborts
	f_{20}	E_{90}		
Adaptive	2.7%	10.3%	58.6	0
Not adaptive	98.0%	38.1%	88.2	n.a.

(d) Bad deviation frequency and common-case deviation

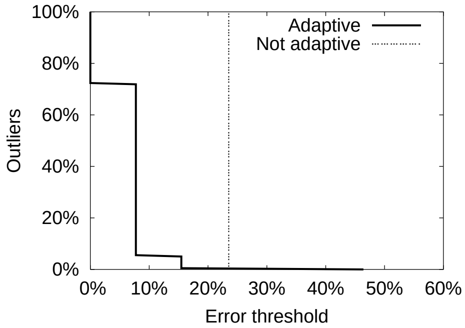
Figure 8.16: Memory adaptation accuracy in Radiator: when unloaded



(a) Latency distribution: adaptive



(b) Latency distribution: not adaptive



(c) Memory outliers: adaptive and non-adaptive

Configuration	Memory deviation		Successful ops	Aborts
	f_{20}	E_{90}		
Adaptive	0.5%	7.7%	39.8	3
Not adaptive	100.0%	23.5%	8.8	n.a.

(d) Bad deviation frequency and common-case deviation

Figure 8.17: Memory adaptation accuracy in Radiator: under load

Mean time to abort (T_{abort})	20.6 s	(0.8 s)
Mean time to recover ($T_{recover}$)	49.6 s	(5.3 s)

The table shows the frequency and the cost of latency-based aborting for Radiator. We show the mean values of 5 experimental runs, with standard deviations in parentheses. Note that aborts only occur when an operation is hit by an upward load transition.

Figure 8.18: Cost of aborting on latency callbacks in Radiator

certain timeout period, and to restart it. However, aborting operations comes at the cost of additional latency: the time taken by the aborted operation is wasted, and the application must also spend some time to recover from the abort and start a new operation. Figure 8.18 shows the cost of aborting for the 5 experimental runs described in the previous section. The mean time to abort T_{abort} is the length of the aborted operation and is almost exactly equal to the latency callback threshold; the mean recovery time $T_{recover}$ is the time to restart the Radiator process, and re-read the input data file.

The total cost of an abort is in the tens of seconds: however, this is usually preferable to continuing execution at high fidelity under memory pressure, which can take hundreds of seconds and severely impact other applications in the meantime.

Abort-and-restart is a general and powerful solution to a commonly observed asymmetry in many adaptive systems: if resource demand is too high compared to resource supply (e.g. if resource supply drops sharply), then program execution slows down, and the wait time until the next adaptive decision point can be arbitrarily long. On the other hand, if resource demand and fidelity are too small, the application will reach the next adaptive decision point quickly, and can rapidly adjust fidelity upwards. E.g., Noble et al. [71] have observed that an adaptive video stream can increase fidelity rapidly when network bandwidth improves; but if bandwidth drops suddenly, adaptation is slow: it must wait for the current segment of video to be fetched at high fidelity before it can degrade subsequent segments.

Aborts can be used to recover not only from resource shortages, but also from bugs and crashes. The importance of clean and cheap abort mechanisms has been frequently observed in a wide range of contexts. E.g. Recoverable Virtual Memory [82] provides a transactional view of memory operations in Unix processes, with the ability to abort or commit a series of modifications to program memory. My observation is that for single-threaded applications with soft state, an efficient abort mechanism can be useful by itself, even without support for persistence or serializability.

Of course, adaptation through aborting comes at a cost. Abort and recovery consume time and resources, which must be balanced against the potential savings. Depending on the implementation, aborts may also have a visual impact on the user. E.g., my simple "kill-and-restart" mechanism causes Radiator's window to disappear, reappear, and remain blank while the input file is being read. Any adaptive policy must balance all these costs against the potential benefits of aborting. The latency callback thresholds used in my prototype provide an extremely simple form of such cost/benefit analysis. More sophisticated methods would dynamically predict the relative costs of aborting vs. non-aborting, in terms of latency as well as other resources and user distraction.

Variation across scenes

The results in the previous experiment were for a single scene: "dragon". How would these results vary from one scene to another? We know that memory demand varies widely from one scene to another, but in all cases is a predictable function of polygon count and fidelity. Thus we expect that while the value of the optimal fidelity f_{opt} will vary from scene to scene, the accuracy of adaptation will be similar.

Figure 8.19 shows the latency and memory deviation for the unloaded (CPU-bound) and loaded (memory-bound) phases respectively. We see that, for all scenes, latency is within 10.3% of optimal 90% of the time when CPU-bound, and memory demand is within 32% of optimal 90% of the time when memory-bound. Figure 8.20 shows the bad deviation frequency and common-case deviation corresponding to the graphs.

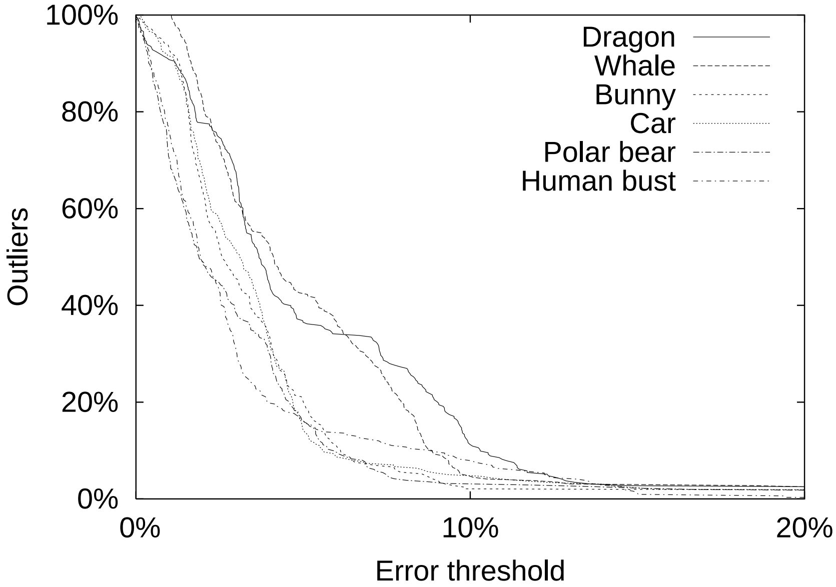
Figure 8.21 shows the distribution of latency for the CPU-bound and memory-bound phases. We see that, when CPU-bound, latency is close to the optimal 9 s. When memory-bound, it is usually below 9 s: i.e., when there is deviation from the optimal memory usage, it is usually on the conservative side, and the system avoids thrashing. We see a wide spread in latency due to variation in CPU demand across scenes: recall that CPU demand is data-specific (Section 7.4.2).

Scaling the experiment

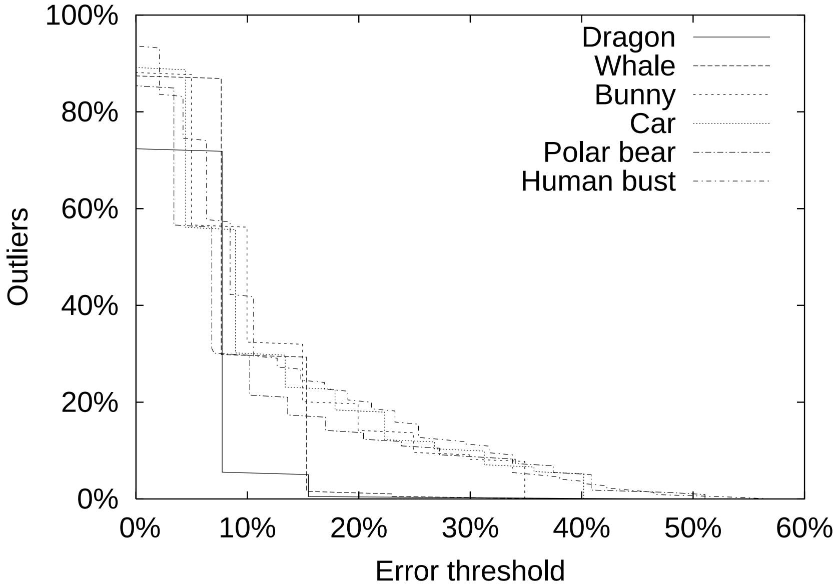
The experiment described at the beginning of this section showed Radiator adapting its fidelity to memory load; however, the highest fidelity ever achieved was 0.1 (Figure 8.14). This is because the limited memory supply on the mobile hardware platform (96 MB) and the large number of polygons in the scene (109K polygons) prevent operation at higher fidelity, even when there is no load.

Mobile devices, however, are growing more powerful every day. Moreover, when we have connectivity to a fast compute server, we might wish to execute the radiosity computation remotely, thus achieving a higher fidelity. To verify that multi-fidelity adaptation is equally effective at high fidelities on a fast server machine, I repeated the baseline experiment — Radiator executing progressive radiosity repeatedly on the "Dragon" scene — on a fast server (2.2 GHz Intel Xeon processor, 512 MB of memory). I used a similar, but higher, background memory load: a square wave pattern alternating between 0 MB and 256 MB.

Figure 8.22 shows a timeline for one run of this "server" experiment. We see very similar behaviour to the "mobile" case:



(a) Latency outliers: when CPU-bound

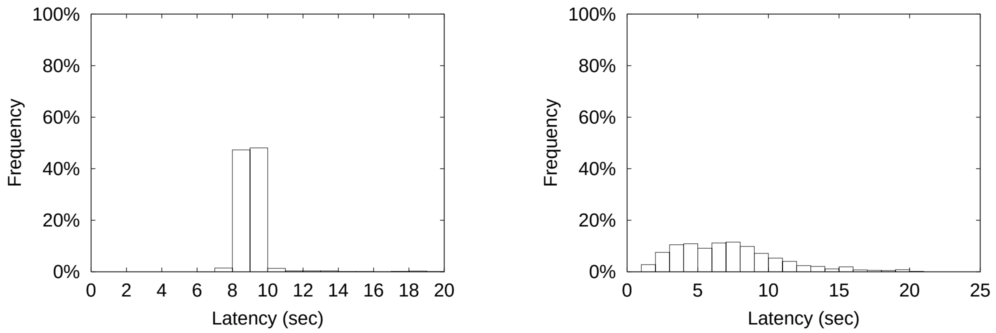


(b) Memory outliers: when memory-bound

Figure 8.19: Memory adaptation accuracy for different scenes in Radiator: outlier graphs

Scene	Latency deviation (when CPU-bound)		Memory deviation (when memory-bound)	
	f_{20}	E_{90}	f_{20}	E_{90}
Dragon	2.7%	10.3%	0.5%	7.7%
Whale	2.7%	9.7%	1.6%	15.3%
Bunny	2.5%	9.2%	14.2%	24.9%
Car	2.4%	8.9%	18.4%	31.3%
Polar bear	2.3%	8.6%	14.2%	27.2%
Human bust	1.9%	8.6%	20.5%	31.7%

Figure 8.20: Memory adaptation accuracy for different scenes in Radiator: bad deviation frequency and common-case deviation



(a) CPU-bound (b) Memory-bound

Figure 8.21: Memory adaptation accuracy for different scenes in Radiator: latency distribution

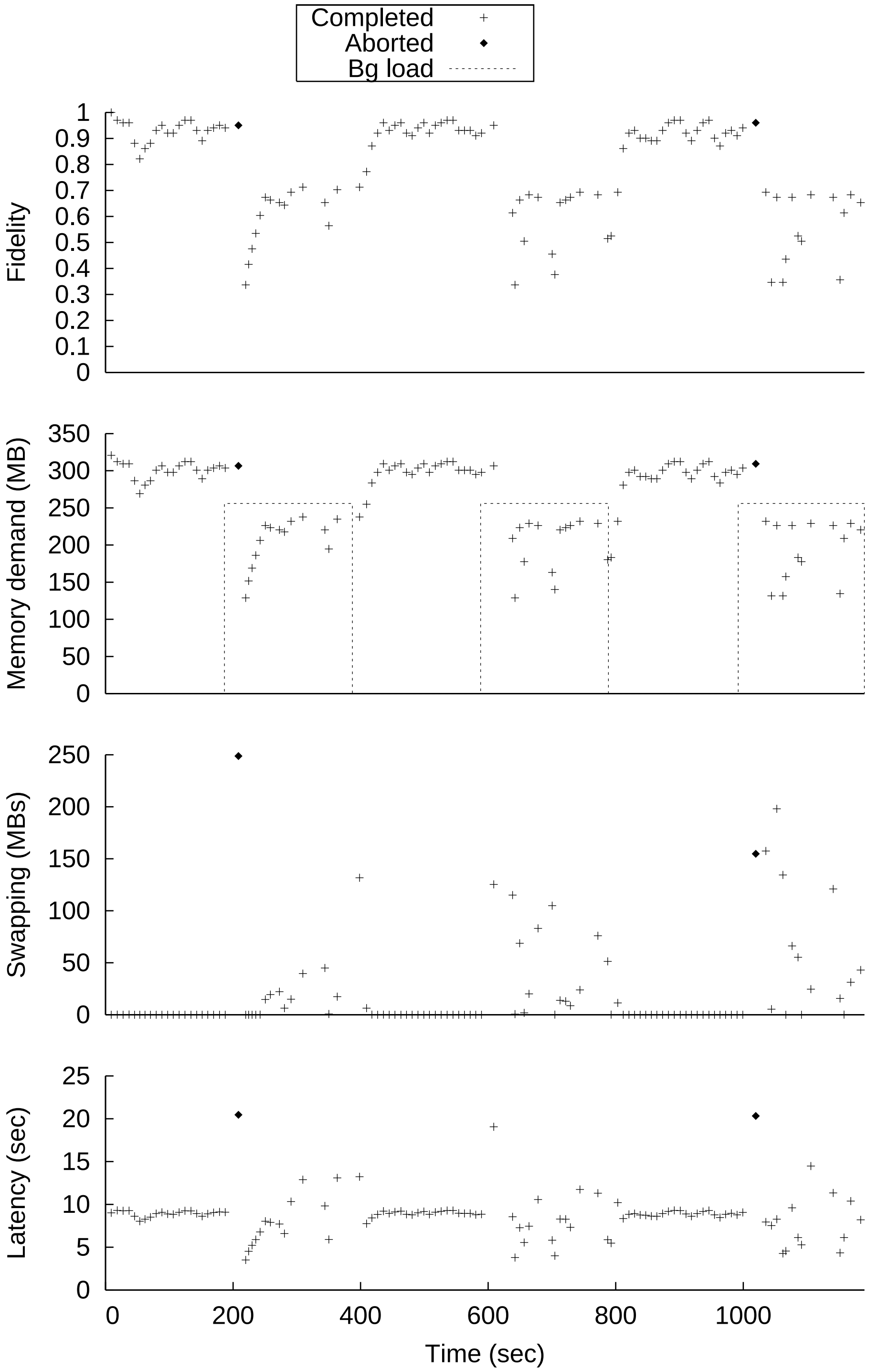
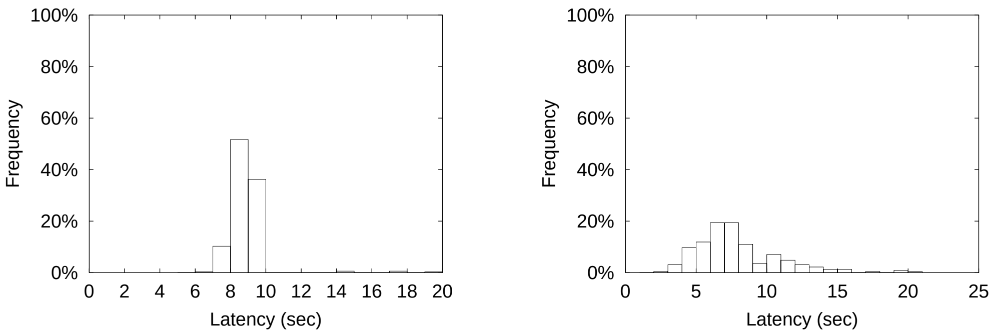


Figure 8.22: Adaptation in Radiator under varying memory load on a fast server



(a) CPU-bound phases (b) Memory-bound phases

Figure 8.23: Latency distribution for Radiator with time-varying memory load on a fast server

Configuration	Memory deviation (when loaded)		Latency deviation (when unloaded)		Probability of abort (on upward load transition)
	f_{20}	E_{90}	f_{20}	E_{90}	
Fast server (Xeon)	1.8%	12.7%	20.3%	32.5%	0.8
Baseline (ThinkPad)	0.5%	7.7%	2.7%	10.3%	1.0

Figure 8.24: Memory adaptation in Radiator on two different platforms

- When there is no memory load, Radiator is CPU-bound, and latency is close to optimal (9 s). On this platform, the optimal latency corresponds to a fidelity slightly under the maximum value, which is 1.0.
- When memory load increases, the operation in flight is sometimes aborted; subsequent operations have very low memory demand, and gradually ramp up towards the optimal value.
- When the memory load decreases, Radiator ramps up its fidelity until it is CPU-bound again.

Figure 8.23 shows the distribution of operation latency during the CPU-bound and memory-bound phases, aggregated over 5 runs of the experiment. We see again that the latency when CPU-bound is close to the target (9 s); when memory-bound, it is usually under the target. Figure 8.24 shows the deviation of memory demand and latency from optimal, and the probability of abort for this configuration, as well as the baseline configuration. We see that though deviation is slightly higher, latency is still usually within 13% of optimal when CPU-bound. When memory-bound, memory usage is usually within 33% of optimal; from the histogram we can see that errors are almost always on the conservative side, i.e., the system avoids thrashing.

8.4 Concurrent applications — architect scenario

In the preceding sections, we saw how multi-fidelity adaptation benefits a single application competing with a time-varying but non-adaptive load, by reducing variation in interactive latency and avoiding memory contention. In this section I evaluate the effect of adaptation on two interactive applications running concurrently and competing for resources. My aim is to evaluate the benefit of adaptation by comparing the adaptive (fidelity is dynamically chosen by the runtime system) and non-adaptive (fidelity is statically determined and fixed at runtime) approaches on the same workload.

8.4.1 Experimental setup

The experimental setup emulates the following scenario:

An architect is showing the plans for a building to her client, who is using GLVU *to "walk through" a model of the building. During the walkthrough, the client suggests changes in the placement of windows or indoor lighting; each time, the architect makes these changes and runs the Radiator program to see the effect of the modifications. While Radiator is running, the client continues with the walkthrough; when the new scene has been processed, it is substituted for the old in* GLVU. *Thus* GLVU *runs continuously in the foreground; Radiator runs sporadically in the background, whenever the architect wishes to recompute the lighting effects.*

The core functionality — rendering and radiosity — required to implement this scenario is present in GLVU and Radiator. However, there are aspects of the scenario that these applications do not support:

- GLVU cannot read the annotated scenes produced by Radiator, and so I could not feed the radiosity-annotated scene into GLVU.
- Radiator does not offer any tools for scene editing: thus I was constrained to run radiosity on the same scene repeatedly, since there was no way to modify it on the fly.

The experimental setup attempts to mimic the architect scenario subject to these limitations. I ran the "Notre Dame" benchmark on GLVU: to mimic the continually-running virtual walkthrough, I ran GLVU continuously on the user path trace, with a 1 s latency constraint. The recomputation of lighting effects was represented by sporadically running Radiator in the background, also on the "Notre Dame" scene. Each Radiator operation ran a progressive radiosity computation with a 10 s latency constraint. Between operations, Radiator idled for a "think time" chosen randomly from the range 0–10 s.

I ran this experiment 5 times on each of 4 configurations:

1. *Both adapt*: both GLVU and Radiator use multi-fidelity adaptation to match fidelity

to their latency constraint.

2. GLVU *adapts*: GLVU alone adapts, Radiator uses a static policy of setting the fidelity to 0.05 for all operations.
3. *Radiator adapts*: Radiator alone adapts, GLVU uses a static policy of setting the fidelity to 0.5
4. *Neither adapts*: GLVU and Radiator used fixed fidelities of 0.5 and 0.05 respectively

The static fidelity values used here were chosen to represent reasonable hand-optimized values. I.e., GLVU's latency for the "Notre Dame" scene at a fidelity of 0.5 is about 1 s when there is no CPU load. The latency will, however vary over time even without CPU load: we have already seen that GLVU's CPU demand at a given fidelity varies with camera position. A fidelity of 0.05 for Radiator gives us a latency of around 6 s when GLVU is not running, and avoids memory contention when GLVU is running. While these values might not be the optimal static values for any particular benchmark, I believe they represent the best that we can reasonably expect from a static approach. In Section 8.4.2 I will show that benchmark-specific fidelity tuning can achieve the desired mean performance for any workload, but that performance variability can only be reduced by dynamic fidelity adaptation.

This experimental setup avoids memory contention between GLVU and Radiator: thus, both applications are adapting based on the CPU resource alone. This also avoids the need for aborts in the case of Radiator: though I had enabled latency callbacks of 20 s for Radiator when adaptive, they were never triggered.

8.4.2 Experimental results

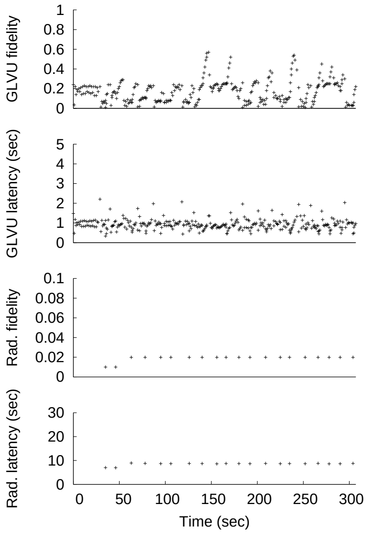
Figure 8.25 shows one representative run of the architect scenario, in each of the four configurations. I show GLVU's fidelity and latency, and Radiator's fidelity and latency over time. Note that when GLVU adapts, the latency is usually around 1 s; when it does not adapt, latency is highly variable. Multi-fidelity adaptation substitutes variability in fidelity for variability in latency, by adapting to changes in CPU demand as GLVU moves through the user trace, and to changes in CPU supply caused by the sporadic execution of Radiator.

Radiator's latency does not vary much, even when it is not adaptive. This is because:

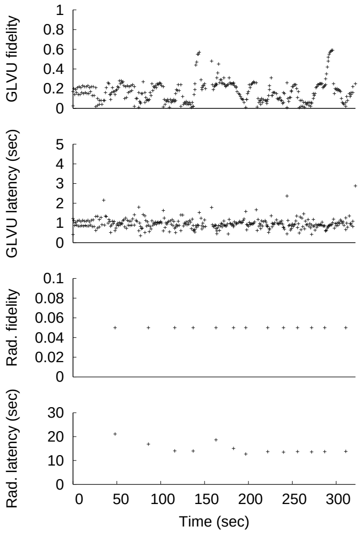
- Radiator's CPU demand does not change over time, since it executes the same computation on the same input data.
- At the time scale of 10 s, the background load caused by GLVU is constant, and does not vary from one radiosity computation to the next.

However, the statically chosen fidelity converges to a latency of 14 s, whereas the adaptive configuration converges to the optimal 9 s.

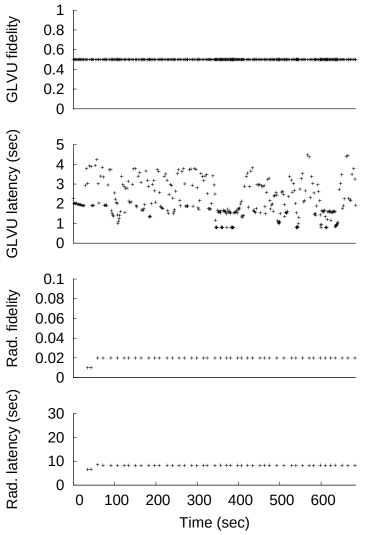
Figures 8.26 and 8.27 show the latency distributions for GLVU and Radiator from 5 runs of the experiment in each configuration. As we expect, we see GLVU's latency clustered around 1 s when adaptive, and highly variable when not. Radiator is consistently at 9 s



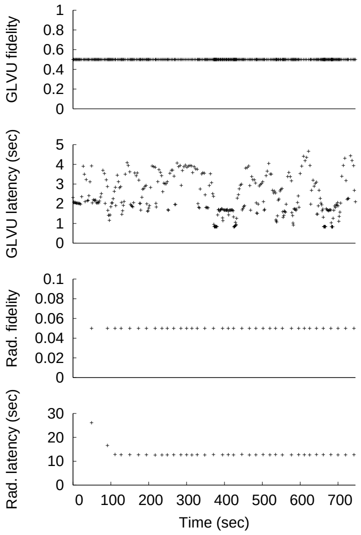
(a) Both adapt



(b) GLVU adapts



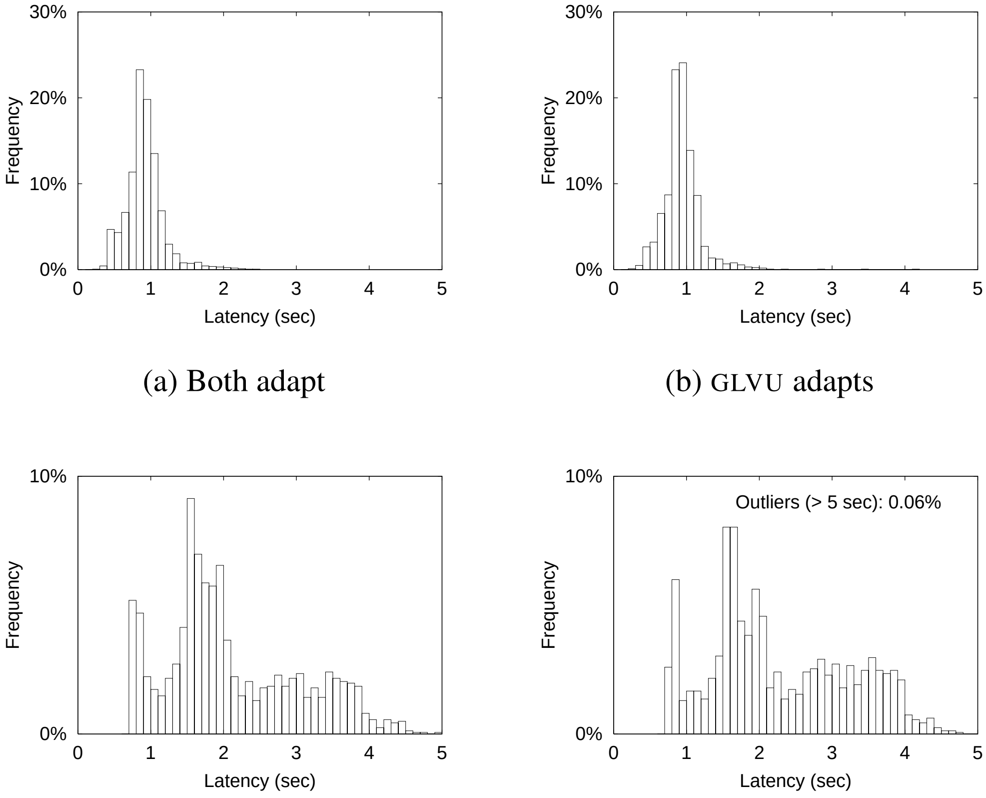
(c) Radiator adapts



(d) Neither adapts

Note that when GLVU is adaptive, there are fewer points on the Radiator timeline: this is because the entire experiment takes much less time when GLVU is adaptive, i.e., it runs through the user trace much faster.

Figure 8.25: GLVU and Radiator in architect scenario



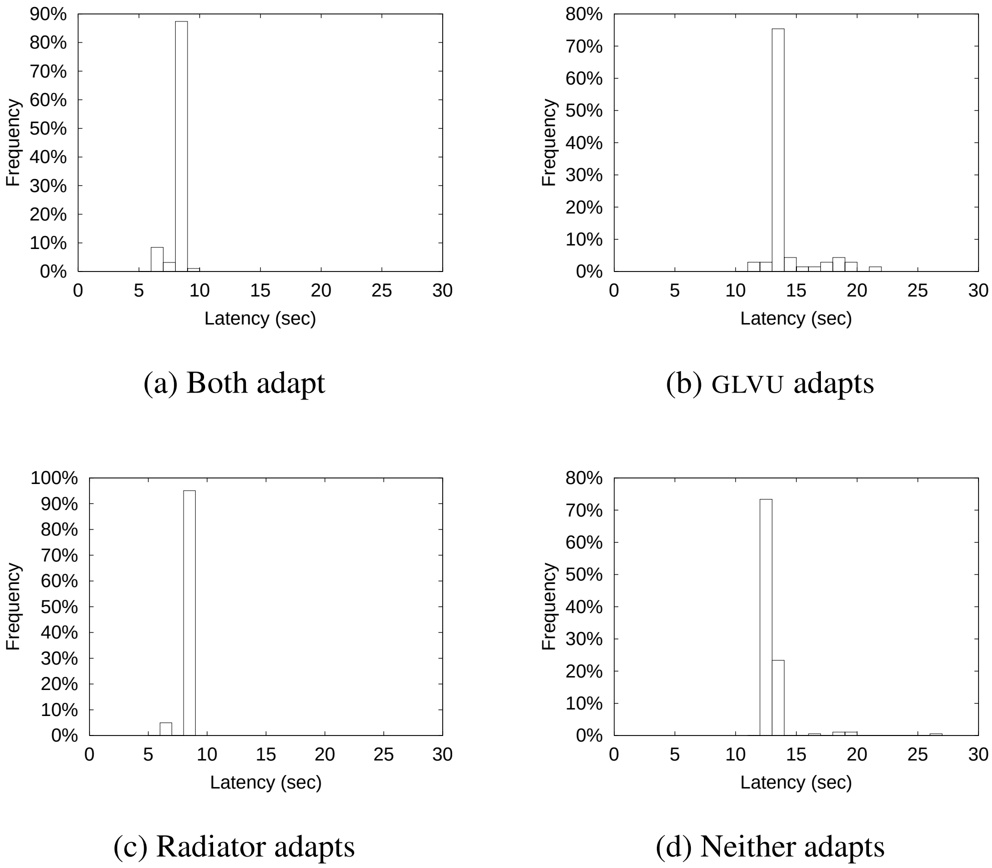
(a) Both adapt

(b) GLVU adapts

(c) Radiator adapts

(d) Neither adapts

Figure 8.26: Latency distributions for GLVU: architect scenario



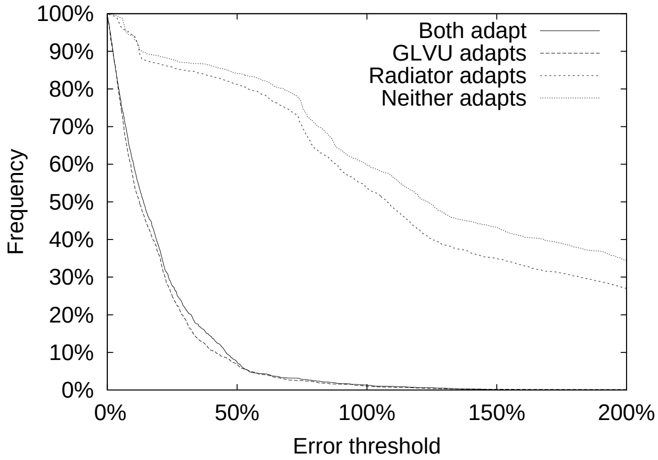
(a) Both adapt

(b) GLVU adapts

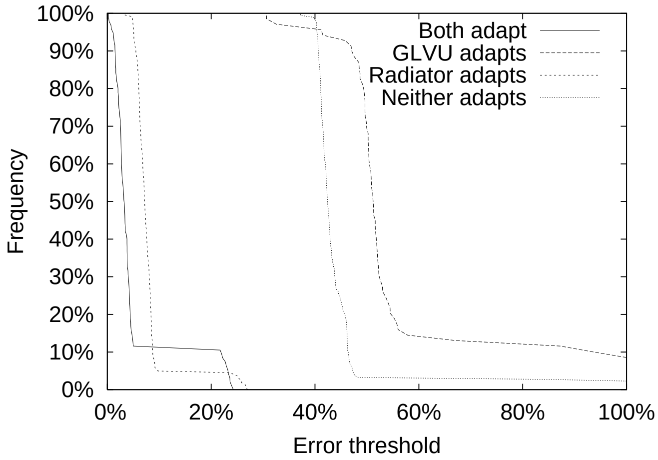
(c) Radiator adapts

(d) Neither adapts

Figure 8.27: Latency distributions for Radiator: architect scenario



(a) Latency outlier distribution: GLVU



(b) Latency outlier distribution: Radiator

Adaptation		GLVU		Radiator	
GLVU	Radiator	f_{20}	E_{90}	f_{20}	E_{90}
Adaptive	Adaptive	38%	46%	12%	22%
Adaptive	Not adaptive	36%	42%	100%	99%
Not adaptive	Adaptive	87%	296%	5%	9%
Not adaptive	Not adaptive	89%	311%	100%	46%

(c) Bad deviation frequency and common-case deviation

Figure 8.28: Concurrent GLVU and Radiator: latency deviation

when adaptive, and 14 s when not. The adaptive case has a small number of outliers on the low side, corresponding to an initial ramp-up of fidelity to the steady-state value. The non-adaptive case has a few outliers on the high side, again corresponding to the first few operations,which are affected by the paging activity caused by GLVU and Radiator starting up simultaneously.

Figure 8.28 shows the deviation of GLVU's and Radiator's latency from the optimal values: 0.9 s and 9 s respectively. We see that both applications see significant benefit from adaptation. We also see that each is each is able to adapt effectively whether the other adapts or not. I.e., GLVU, when adaptive, has the same behaviour whether Radiator adapts or not. Similarly an adaptive Radiator is able to mask the effect of GLVU's adaptation, or lack of it.

When GLVU is *not* adaptive, there is some dependence on Radiator's behaviour: GLVU performs slightly better when Radiator is adaptive. This is because adaptation shortens Radiator's "on" periods, while the think times or "off" periods are unaffected: the "on" periods are the ones that cause high latency in GLVU. Radiator, when not adaptive, performs slightly better when GLVU is *not* adaptive. This is simply because when GLVU is not adaptive, the walkthrough takes significantly longer: thus the outliers caused by the initial page-in activity have a smaller weight compared to the larger number of operations in steady state.

Variance

So far, I have focussed on the ability of adaptive applications to keep latency close to a pre-determined *optimal* value L_{opt}. However, for many interactive applications, *variance* in latency is also important [62]: i.e., deviation from the *expected*, rather than the optimal value. In other words, user expectations of latency might be based not only on the latency constraint they impose on an application, but also on the runtime behaviour of the application itself. A user might set a latency constraint of 1 s, but realise that they would prefer a constant response time of 2 s to a response time of 1 s 90% of the time and 3 s 10% of the time.

My current prototype does not support setting explicit bounds on latency variance. However, simply by attempting to meet a pre-determined latency constraint, the system reduces variance as a side-effect. Figure 8.29 shows the standard deviation of GLVU's and Radiator's latency for the data from the "architect scenario" experiments. In this case, I measure the deviation of latency from the observed mean value, rather than from a pre-determined optimal value.

We see that adaptation reduces not only the mean latency μ, but also the standard deviation σ, i.e. the variation from the mean. In the case of GLVU, adaptation decreases variation by adjusting fidelity to keep CPU demand steady; for Radiator, by avoiding the initial high-latency outliers during startup. In fact, adaptation reduces not only the absolute variance, but also the coefficient of variation σ/μ: the standard deviation scaled by the mean.

GLVU's strategy	Radiator's strategy	GLVU's latency			Radiator's latency		
		μ	σ	σ/μ	μ	σ	σ/μ
Adaptive	Adaptive	0.9 s	0.3 s	0.29	8.5 s	0.6 s	0.07
Adaptive	Static	0.9 s	0.3 s	0.28	14.2 s	1.9 s	0.13
Static	Adaptive	2.1 s	0.9 s	0.44	8.3 s	0.4 s	0.05
Static	Static	2.3 s	1.0 s	0.43	13.1 s	1.4 s	0.10
Optimal static	Optimal static	1.0 s	0.44 s	0.46	8.4 s	0.3 s	0.03

The table shows the mean μ, standard deviation σ, and coefficient of variation σ/μ for the latency of GLVU and Radiator, over 5 runs of each configuration of the architect scenario. The optimal static configuration used a fidelity of 0.17 for GLVU and 0.019 for Radiator.

Figure 8.29: Concurrent GLVU and Radiator: latency variability

Benchmark-specific tuning

Multi-fidelity adaptation reduces mean latency by degrading fidelity to the appropriate level for the resource conditions; it also reduces variation in latency by compensating for dynamic changes in resource supply and demand. The first of these benefits can be achieved through *benchmark-specific tuning*: choosing a static fidelity that achieves the desired mean latency, given a particular workload and background load pattern. Such tuning is unwieldy and impractical: often, we do not know the runtime conditions that an application will experience. Moreover benchmark-specific tuning can bring mean latency on target, but cannot regulate dynamic variation.

To validate this claim, I repeated the "architect scenario" experiment in the *optimal static configuration*: fidelity for both applications is fixed, but the fixed value is chosen to keep mean latency on target for this workload. To compute the optimal static fidelity, I used the mean fidelity chosen by the fully-adaptive configuration over 5 runs of the experiment: 0.17 for GLVU and 0.019 for Radiator.

The last line of Figure 8.29 shows the mean latency and variance for this *optimal static* case. Now mean latency is close to target, as in the adaptive case. However, GLVU's latency is still as variable as in the non-adaptive case. For Radiator, the optimal static strategy is in fact the best one, since its resource supply and demand are constant during the experiment. The adaptive case deviates slightly from optimal during the first few operations, before converging to the correct value. The optimal static strategy does not suffer from this problem, and has slightly lower variability.

8.5 Summary

In this chapter I evaluated multi-fidelity adaptation with both synthetic and real applications. I showed that the system responded to sharp changes in CPU load within seconds, and to changes in memory load within tens of seconds. With adaptation, latency was usually within 43% of GLVU's latency constraint when competing with a synthetic background CPU load, compared to 494% at full fidelity. I compared these results across multiple input scenes, latency constraints, and background load patterns. I showed that larger scenes showed slightly larger deviation; that load variation on the same time-scale as adaptation had the most destructive effect on adaptation accuracy; and that larger peak-to-trough transitions in load caused less accurate adaptation.

The system can maintain latency constraints as small as 0.5 s, but not lower:

- to reduce CPU demand below 0.5 s on my test platform, fidelity would have to be decreased to an unusable level;
- even so, the large scheduler quantum would cause large deviations from the latency constraint.

Moore's Law will soon solve the first problem, while the second simply requires more fine-grained scheduling. Modern processors can easily support scheduling quanta of 10 ms with little overhead; I hope that future versions of the Linux kernel will support smaller quanta, and enable us to achieve the desired "seamless" interactive response time of 100 ms [12] for virtual walkthrough.

In the case of Radiator, constraining memory usage is key to good performance. I also showed that, with adaptation, Radiator's memory usage (with a synthetic background memory load) is within 10% of optimal in the common case, compared to 38% when using a static, conservatively chosen fidelity. I noted that effective memory adaptation requires a clean and cheap abort mechanism for in-flight operations that are thrashing the system by consuming too much memory. My simple abort mechanism — killing and restarting the application process — costs us an additional 70 s per aborted radiosity operation.

I also evaluated the adaptation of GLVU and Radiator when they ran concurrently and competed with each other. I showed that adaptation aided both applications in meeting their latency constraint, independently of the other application's behaviour. Finally, I showed that adaptation helps not only to maintain latency constraints, but also to reduce variability in latency despite dynamic resource variation. Benchmark-specific tuning can provide the former benefit but not the latter.

Chapter 9

Related Work

The villany you teach me I will execute, and it shall go hard but I will better the instruction.

(The Merchant of Venice, III.i)

The idea of adapting fidelity to gain performance is an old one: however, to the best of my knowledge, the research presented here is the first to propose a general computational model for multi-fidelity adaptation. I believe that the resource model, the operation-based API, and the history-based prediction of resource demand as a function of fidelity, are also novel. Of course, I derived inspiration and borrowed many ideas from other researchers, and hope that this work will influence future research into mobile and adaptive systems. This chapter presents selected research relevant to one or more of the issues addressed in this dissertation.

Section 9.1 describes work that addresses similar problems, or uses similar solutions, to those described in this dissertation. In each case, I briefly summarize the cited work, contrast it with mine, and list the novel contributions of my work.

Section 9.2 presents a broader view of adaptive software systems: its purpose is not to enumerate all such systems, but to map out the design space with a few examples. The design space is decomposed along several axes; for each axis, I indicate the choices I made in designing the multi-fidelity runtime system, and describe a few examples of differing design choices.

In an interactive system, the true measure of success is user happiness. Agile and accurate adaptation is of little use if the system optimizes the wrong utility function: for example minimizing latency when the user wished to maximize fidelity. Section 9.3 discusses some research that could help us derive good utility functions: by quantifying the notion of application "quality", and by inferring user preferences dynamically from context.

9.1 Related Work

9.1.1 Mobile application adaptation

The idea of fidelity adaptation in resource-constrained mobile environments is not new. Fox [37] has shown that *dynamic distillation* of web content can improve web download times by degrading the fidelity of downloaded content to match the network bandwidth and the capabilities of the viewing device. Noble [71] showed that fidelity adaptation in the Odyssey system enables a variety of applications to maintain performance despite variations in network bandwidth

My work differs previous research in mobile application adaptation by

- including many different kinds of adaptation under a broad unifying concept: the *multi-fidelity computation*.
- demonstrating the importance of *history-based resource demand prediction* to effective fidelity adaptation
- showing through empirical evaluation that multi-fidelity adaptation is well suited to the needs of interactive applications.

More recently, de Lara [20] has shown that *component-based adaptation* can enable adaptation in applications even without source code access. This research is complementary to mine: while I have assumed source code access in my application case studies, the general ideas of multi-fidelity computation and history-based demand prediction are also applicable to binary-only situations.

9.1.2 Odyssey

The largest contribution to the concept, design and implementation of my system comes from Odyssey [71]. Odyssey demonstrated the importance of adapting to network bandwidth in mobile scenarios, and proposed the first generic API for application-aware adaptation. The multi-fidelity runtime builds on Odyssey, and uses Odyssey's monitoring and prediction code for its network supply predictor.

Concurrently with my research, Jason Flinn also worked on the Odyssey infrastructure; he extended it to support energy as a resource and battery lifetime as a performance metric. Thus, at the conceptual level, his contribution was the demonstration that application-level fidelity adaptation and remote execution can help in improving battery lifetimes [34, 33]. My contribution is the broad concept of multi-fidelity computation, the API and runtime system architecture that supports these computations, and the demonstration that together they can improve interactive response times.

At the implementation level, Flinn built the supply predictor and demand monitor for energy, the remote execution engine, and the file cache predictor [32]; I built the solver, and the supply predictors and demand monitors for CPU and memory.

9.1.3 QoS and real-time operating systems

QoS-aware operating systems and middleware [67] focus on dividing resources across applications to satisfy some notion of fairness. Thus, each application receives a promised *rate* of processing or network bandwidth, or sometimes a fixed *share* of available bandwidth.

In these systems, applications are expected to specify their resource needs, and adapt correctly to their runtime resource allocation. While this may be a straightforward process for multimedia applications such as video, we have seen in this dissertation that complex, interactive applications require an additional step: mapping application-specific parameters to resource demand. In some QoS architectures, the need for such an explicit mapping is realized: e.g., the "application resource profile" in Q-RAM [57]. However, to the best of my knowledge, this is the first work to develop and validate a general method — *history-based demand prediction* — to generate such mappings or profiles

Thus, while QoS research focusses on fair *allocation* across applications, my research enables effective and automatic *adaptation* within applications. The two techniques are complementary.

A second point of difference is in the target application domain: most QoS systems focus on streaming multimedia applications such as video. These applications can be characterized as streams of periodic operations, each of fixed or bounded resource demand: the system's objective is then to make sure that all active streams can be sustained without excessive delay or jitter. Such a model does not fit interactive applications, where we care about the response time of individual operations rather than the aggregate throughput and jitter of a stream. Further, interactive 1applications are not periodic — operations are triggered by user requests — nor are the operations always of fixed size – runtime parameters can cause wide variations in per-operation resource demand.

Some real-time operating systems allow a broader workload definition, although they are still focussed on multimedia applications. E.g., rate-based execution [52] allows task streams that are not perfectly periodic, but that have some average task rate and period. However, *task size* is still fixed: the system assumes a static worst-case resource demand estimate. Abeni [2, 3] describes a system where both task size and interarrival time are represented as probability distributions rather than fixed values. However, the probability distributions are statically derived, and thus resource demand estimates are not made dependent on application runtime parameters.

In summary, my research differs from that in QoS and real-time operating systems in three ways:

- the presence of explicit, history-based demand prediction

- the use of adaptation rather than allocation to achieve performance goals
- the focus on interactive rather than streaming applications

9.1.4 Resource prediction

Resource *supply* prediction — i.e. load prediction — is found in some form or the other in every adaptive systems. Typically, prediction is implicit in these systems: offered load or other system indicators are used as control signals that drive load balancing [58], scheduling [27], process migration [28, 45], or resource allocation [89].

A few systems use *explicit* resource prediction. The Running Time Advisor [25] predicts the running time of a job on any host in a distributed system, given its nominal execution time: i.e., it predicts resource supply and performance, but the application must specify the resource demand. The RTA is based on Dinda et al.'s work on host load prediction [23] using a variety of time series models. For example, autoregressive linear models [10] take the variance as well as the mean of recent observations into account. Such techniques could improve the accuracy of my relatively simple supply predictors (Section 5.5).

Resource demand prediction is comparatively rare. Some load-balancing systems [58, 45] estimate the CPU demand (execution time) of a job from an analytically or empirically derived probability distribution. However, they do not make use of any dynamic, instance-specific information such as the job's runtime parameters: at most, they use the job's current lifetime to predict its future execution time. For regular, loop-based codes, static compiler analysis has been used to automatically partition programs into subtasks of known resource demand [87].

Demand prediction as a function of runtime parameters occurs in two systems that I know of. Abdelzaher's "automated profiling system for QoS" [1] dynamically estimates the CPU utilization of a multimedia stream as a linear function of its task (frame) rate and average task size in bytes. The technique used — linear estimation with forgetting — is identical to Recursive Least Squares (Section 6.3.1). The predictions of utilization are then used for QoS admissions control or allocation.

More closely related to my work is the PUNCH system [53]. PUNCH uses machine learning techniques to predict the CPU demand of a program run as a function of application-specific runtime parameters. This is very similar to history-based demand prediction, with more sophisticated (and expensive) learning algorithms than linear estimation. The demand predictions are not used for application adaptation, however, but for load balancing in a grid computing framework.

Thus, while there is a large amount of work on load prediction and some on demand prediction, I believe my system is the first that has all of the following properties:

- Demand is predicted from empirical data, both offline and online.
- Demand is predicted as a function of the application's runtime parameters, both tun-

able and nontunable.
- Demand predictions are used to drive adaptation of the tunable parameters.

9.2 Design Space

To more easily describe the large design space for adaptive software systems, I have decomposed it into five axes, not necessarily orthogonal: *adaptation vs. allocation*, *centralized vs. decentralized*, *prediction vs. feedback-control*, *application-aware vs. application-transparent*, and *gray-box vs. in-kernel*. Sections 9.2.1–9.2.5 deal with each of these in turn.

9.2.1 Adaptation or Allocation?

Variation in application resource supply can be addressed at two levels: the OS can change the *allocation* of shared resources across competing applications, and applications can *adapt* their behaviour to their current allocation. The two are complementary, and a really effective system will use a combination of the two techniques.

My approach is an “adaptation-only” one: the system *predicts* each application’s resource share, but does not attempt to modify it. This has a number of practical benefits (Section 5.1): it facilitates a gray-box, user-level implementation, it assures non-adaptive legacy applications that their resource share will not be affected, and it does not require guarantees from the underlying resources (which are difficult to obtain, for example, from wireless networks). This best-effort, adaptation-only philosophy is shared by other mobile adaptive systems [37, 71].

A large body of work, which I refer to collectively as “Adaptive QoS”, focusses on *allocation*: sharing resources across applications to maximize overall utility. Given an allocation, each application is expected to adapt its tunable parameters to maximize its own utility. I believe that adaptation and allocation are complementary. My resource model and history-based prediction approach can map resource allocations into utility; these application utility functions could be used by a QoS-aware allocator to maximize global utility. David Petrou and I have proposed an architecture for combined adaptation and allocation, based on *goodness hints* [74], where

- applications provide demand predictors, and users their utility functions, to the system as hints.
- the system performs a two-level optimization: application tunable parameters are adjusted to maximize application utility given a resource allocation, and resource allocations are adjusted to maximize global utility.

9.2.2 Centralized or decentralized?

My approach to finding optimal adaptive parameter values is to use a *centralized solver* or optimizer (Section 5.3). Lee [57] uses a similar approach, and describes optimization algorithms that efficiently maximize utility: i.e., find optimal or near-optimal parameter values under certain conditions. Incorporating these algorithms into my solver could potentially improve its efficiency, generality, robustness and theoretical soundness.

Stratford and Mortier [90] advocate *market models* as an alternative to centralized solvers. Here the system prices each resource according to the observed demand. Each application is given a fixed number of credits to spend, and must allocate them across different resources. Neugebauer and McAuley [68] have developed a market-based CPU allocator based on *congestion pricing*. These microeconomic approaches generalize proportional-share schemes such as *lottery scheduling* [93], which allocates resources strictly in proportion to credit allotment, and uses a different currency for each resource.

Market models, while decentralized and possibly more robust than solvers, impose an additional burden on the application programmer. Instead of utility functions, they must now derive *bidding strategies* that allocate application credits across resources, given the price of each resource. I believe good bidding strategies are harder to create than good utility functions, and know of no automatic way to derive the former from the latter.

9.2.3 Prediction or feedback-control?

In Section 5.3.2, I discussed two different ways of doing adaptation:

- *feedback-control*: iteratively adjusting fidelity based on recent performance.
- *predict-and-search*: searching for the optimal value across the entire space of fidelities, based on predictors of performance as a function of fidelity.

A good example of the first approach is the feedback-driven CPU allocator of Steere et al. [89]. The system continuously monitors each application's throughput or "progress", and adjusts its CPU share to maintain a steady rate of progress. For some classes of applications, the system can transparently measure progress by monitoring the application-kernel interface: the fill level of a socket buffer can tell us whether the producer and consumer rates are mismatched. Lutfiyya et al.[59] describe a feedback-driven approach based on *probes*: embedded code modules that measure application-specific metrics and trigger *alarms* when these metrics exceed specified bounds. The system reacts by consulting a pre-specified rule base: e.g., "if the input buffer is full, increase the CPU priority of the consumer process".

By contrast, the research presented in this dissertation uses the predict-and-search approach, which is based on two broad types of predictors:

- *temporal*: using past values of a time-varying signal y to predict future values. These answer "what next" questions ("what will CPU load be over the next 1 s?"), and I use them for resource supply prediction. I.e., I assume that an application's resource supply in the near future is independent of its adaptive parameters, but can be predicted from the recent past.

- *parametric*: using past values of y and associated parameter values $\langle x_1, x_2, \ldots \rangle$, to predict y *as a function of* $\langle x_1, x_2, \ldots \rangle$. Parametric schemes answer "what if" questions ("what would CPU demand be at fidelity 0.7?") I use these for resource demand prediction.

9.2.4 Application-aware or application-transparent?

In multi-fidelity adaptation, as in Odyssey, applications are aware and in control of the adaptive process: the system provides support for making fidelity decisions, but leaves the decisions to the application. This awareness comes at a cost: the application's behaviour must be modified, which usually involves source code modifications. The multi-fidelity API is designed to minimize this cost: however, source code modification still presents a barrier to modifying legacy applications.

Application-transparent adaptation does not require us to modify the applications, but is restricted in its scope and power. Such adaptation usually happens at the system level, i.e., to adjust system-wide parameters rather than application fidelity. Perhaps the most famous example is TCP congestion control [51], which is invisible to applications using the standard socket interface. Similarly, Coda [55] adapts to mask the effect of low bandwidth or disconnection from file servers, but presents a standard file system interface to applications.

How can we achieve application adaptation without source code modification? Transparent adaptation of *data fidelity* can often be achieved through *proxy techniques*. By placing a proxy on the data path between a server and a client, we can degrade data fidelity without modifying either. Fox et al. [37] use this technique to degrade web content, as does the multi-fidelity web browser (Section 7.5). Component-based adaptation [20] allows even finer-grained control, by intercepting the data flow between different components of a single application.

Computational fidelity adaptation, however, often involves changing state variables and execution paths within a program. Here we need techniques to insert code into applications without modifying their source code. Such techniques are typically used to insert profiling, prefetching, or other performance optimizations; they could potentially be used to insert fidelity adaptation code as well. Binary rewriting tools [88, 77, 17] can transform applications without source code access. Compiler-based approaches [65] require read-only access to source code. Sometimes, programming language features can help to reduce the cost of source code modification, though not eliminate it entirely: Chang et al. [16] use language extensions to export parallelizability information to the system, which then chooses adaptive parameter values and the degree of parallelism at runtime.

9.2.5 Gray-box or in-kernel?

The multi-fidelity runtime system extends OS functionality with services not provided by the Linux kernel. I chose to build it at user-level to ease deployment and debugging. However, to predict application resource supply, the system need knowledge of the kernel's resource allocation decisions. It *infers* this information by translating load statistics into predictions of resource allocation.

Arpaci-Dusseau and Arpaci-Dusseau [6] use the term "gray-box" for such approaches, which combine knowledge of kernel internals with observations made at user level. They describe gray-box techniques for inferring buffer cache contents, disk file layout, and memory availability by using *active probes*: inferring resource state by attempting to use the resource. My approach uses *passive observations*: while less powerful, they have lower overheads and avoid modification of the state being read (the Heisenberg effect). Gray-box techniques can potentially modify or control kernel behaviour as well as predicting it: Chang et. al [15] enforce resource allocations at user level by adjusting process priorities and page protections in Windows NT.

Gray-box techniques ease development and deployability of third-party OS extensions. When these are not overriding concerns, there are advantages to carefully designed kernel modifications. For example, one problem with the multi-fidelity runtime system is that it does not properly account for resources spent by the kernel or a shared server (e.g. the X server) on an application's behalf. New kernel abstractions such as *resource containers* [7] could solve this problem.

More generally, Seltzer and Small [86] advocate *self-monitoring* and *self-adapting* OS kernels. They recommend recording traces and logs of a large number of kernel variables and events, and using off-line as well as on-line analysis to correlate these with application performance. These analyses would be used for adaptations such as modifying prefetch strategies or dynamic code recompilation. This is very similar to my philosophy of "measure,log,learn,adapt" (Chapter 4), which I use at user level for application adaptation.

9.3 Inferring user preferences

The end-goal of application adaptation is to maximize user utility. This requires us to quantify the effect of fidelity and performance on utility. In this dissertation, I assumed that utility functions are provided explicitly by the application or the user, which raises two questions:

- How can application programmers derive good default utility functions?
- How can utility functions be specialized to individual users' needs at runtime?

To create good utility functions, we must quantify the effect on users of different levels of fidelity. This has been widely studied for common media applications such as video.

Webster [95] describes how objective measurements of video quality can be correlated with subjective ratings of video clips by users. Chandra and Ellis [14] recommend using the JPEG Quality Factor (JQF) to represent the user-perceived quality of a JPEG-compressed image: for the adaptive web browser (Section 7.5), this would make output quality identical to fidelity. Wilson and Sasse [97] measured the correlation between video frame rate and biometric indicators of stress such as heart rate. Detailed studies such as these are lacking for adaptive, interactive applications such as augmented reality. This is an important area for future research: for effective adaptation, we must know the user-perceived effect of any change in polygon count, textures, etc.

Similarly, the effect of latency on user satisfaction needs to be quantified. There is a consensus among HCI researchers [62, 12, 69, 13] that there are three important levels of interactive response: "perceptual processing" (100 ms), "immediate response" (1 s), and "unit task" (10 s). However, this does not tell us how to classify interactive operations into these three categories, nor how much fidelity we should sacrifice in aiming for a lower latency constraint. Often, an operation is simply too expensive to meet a lower latency target, forcing us to "demote" it to a slower category. E.g., to achieve "immediate response" in Radiator, we would have to reduce fidelity to an absurd level, which is why I use the 10 s "unit task" constraint instead. Similarly, my intuition is that the 100 ms "perceptual" response time is the appropriate one for GLVU; however, the scheduler quantum of 200 ms makes this an impossible goal under load, and so I use the 1 s "immediate response" target.

A well-chosen utility function can only provide a good default: user utility will still vary across users and task contexts. In the initial stages of design, our architect (Section 1.1) would want quick responses for rapid prototyping. Later, she might prefer a high level of detail and accuracy, even at the cost of higher latency. An appropriately designed interface can allow users to directly adjust utility functions, but at the cost of increasing user distraction. Alternatively, the system can infer the user's preferences from observations about their context; however, such inferences are often uncertain and unreliable, and can lead to bad adaptive decisions.

Clearly, we need some combination of the two approaches. Horvitz [47] describes such a combination, which he terms a *mixed-initiative* approach. A mixed-initiative system allows users to directly manipulate system settings; it also adjusts these settings automatically by making inferences about the user. The costs of explicit user intervention are continually balanced against the risks of automated decision-making, and the user is always allowed to override or correct bad decisions by the system.

Chapter 10

Conclusion

> *To write history is so difficult that most historians are forced to make concessions to the technique of legend.*
>
> *(Erich Auerbach,* Mimesis, *ch. I)*

Wearable, interactive computing requires small lightweight devices, pervasive wireless connectivity, and interactive applications written for mobile rather than desktop users. In the last decade we have seen rapid advances along all three fronts. Soon we will have the mobile hardware, the wireless networks, and the application software: what we will need is operating system software to mediate effectively between them.

This dissertation has shown that any such system must support application adaptation, and has proposed a novel framework for such adaptation, based on *multi-fidelity computation* and a *supply-demand resource model*. It sets forth a modular system design based on this framework, and describes its key component technologies: history-based prediction, passive resource monitoring, and the gradient-descent solver. Through a prototype implementation and evaluation, the dissertation shows the system significantly improves the latency response of interactive applications.

In this chapter, I summarize the research contributions made by this dissertation (Section 10.1); describe the current status of the system and ongoing research (Section 10.2); and explore some directions for future research (Section 10.3).

10.1 Contributions

10.1.1 Conceptual Contributions

The primary conceptual contribution of this dissertation is *multi-fidelity computation*. It generalizes the classical notion of algorithm, where the output specification is fixed but resource consumption depends on the input data. With multi-fidelity computations, both output quality and resource consumption are allowed to vary, and we can dynamically explore the tradeoffs between them. In developing this concept I also expanded previous definitions of fidelity. I introduced the notion of *computational fidelity* as a complement to data fidelity; more generally, I use the term fidelity to cover all runtime tunable parameters that affect output quality and/or resource demand.

The second conceptual contribution is the *resource model*, which lets us abstract application behaviour as resource demand, system state as resource supply, and performance as a cost function on supply and demand. For the performance metric of greatest interest to us — latency — I showed that simple cost functions are sufficient for many applications.

A third contribution is my *history-based methodology* for predicting an application's resource demand as a function of its tunable parameters. This bridges the gap between quantities relevant to the application (i.e., fidelity metrics) and those understood by the system: resource supply and demand. It allows us to make predictions from empirical observations instead of relying exclusively on analytic models, and to specialize predictive models to time- or data-dependent variation in application behaviour.

Fourth, my API design shows how we can make significant changes in application behaviour with only small changes to their source code, by isolating most of the new functionality to *configuration files* and *hint modules*. Once an application has been modified to use the multi-fidelity API, third parties without access to source code can continue to experiment with different resource demand predictors and utility functions.

Fifth, I have shown that a *predictive, gray-box* approach to resource management is feasible and effective. In other words, we can substantially mitigate the ill effects of resource variability without changing the OS's resource allocation policies or modifying its kernel source code.

Finally, my evaluation strategy highlights a useful new performance measure: the *deviation plot*. With fidelity adaptation, there is hardly ever a free lunch: latency can be improved only by sacrificing fidelity, and vice versa. Thus the true test of an adaptive system is to measure how closely it tracks user policy, i.e., maximizes utility under resource constraints. When user policy is expressed as a latency constraint, the latency deviation plot provides us with a simple yet informative measure of the system's success in staying on target.

10.1.2 Artifacts

This dissertation has produced one substantial artifact: the multi-fidelity runtime system. The system is modular and contains several components, each with value independent of the system as a whole:

- The resource supply and latency predictors estimate the resource availability and performance of an application in the near future.

- The logger transparently collects logs of application resource demand to be used for performance debugging as well as constructing resource demand predictors.
- The solver can optimize utility across multiple dimensions of performance, resource demand, and output quality.
- The applications, which can now adapt to a variety of resource scenarios and performance requirements.

10.1.3 Evaluation Results

The experimental evaluation of this work has contributed several important findings for application adaptation. I showed that, with a carefully designed API, the cost of application modification can be made small if not zero. I also showed that history-based techniques can effectively predict application resource demand, especially with *online updates* and *data-specific prediction*.

My evaluation shows that the system overhead per operation ranges from 1 ms to 20 ms. This overhead is not negligible, but acceptable for operations of 1 s or longer. The overhead can be reduced by more efficient ways of reading kernel load statistics, and by a better optimized solver.

I have shown that *gray-box* techniques can accurately predict resource supply: i.e., that kernel modification is not necessary. My approach is an intermediate point in the design space: it avoids kernel modification but not kernel dependence. My memory supply predictor depends on the Linux VM and the statistics exported by it; if these change in a later kernel, the predictor must change also.

I showed that the agility and accuracy of resource supply prediction are close to the limits imposed by the underlying OS. The CPU supply predictor is accurate over time periods of 1 s or greater, and its agility is on the same time scale. The memory supply predictor's time scale is in the tens of seconds: it is limited by the rate at which VM load statistics are updated. Adapting *downwards* to a spike in memory load can be prolonged for another reason: applications *thrash* when there is memory contention, slowing them down and delaying their next adaptation point. Thus, a cheap *abort-and-restart* mechanism is essential for memory-intensive applications.

I showed that effective adaptation depends on both supply and demand prediction: both contribute to meeting latency goals. I also showed that, with adaptation, applications are much more likely to meet these goals. I also showed that the benefit of adaptation for each application is independent of the adaptive behaviour of other, concurrently running applications. Finally, I showed that adaptation reduces *variance* in latency in addition to bringing mean latency closer to the target value.

10.2 Current status and ongoing work

Work on the multifidelity prototype is ongoing, and other researchers have added capabilities beyond those described in the dissertation so far.

Jason Flinn, Rajesh Balan, SoYoung Park and Tadashi Okoshi have ported the system to the StrongARM-based Itsy [8] and iPAQ hand-held computers. The Spectra remote execution engine [33] allows multi-fidelity computations to execute wholly or partly on remote servers; the latest version of the GLVU virtual walkthrough system uses Spectra services to do *remote rendering*. I.e., when appropriate, the 3-D rendering is done on a fast server and the resulting image shipped back to the client. This capability was implemented by Shaheen Gandhi, Tadashi Okoshi, Tridib Chakravarty, and Ningning Hu.

Research in multi-fidelity adaptation is ongoing as part of the *Chroma* project, whose goal is providing middleware for easy and automatic adaptation and resource management in mobile applications. Current work in Chroma is focussed on *automatic stub generation*: given an Application Configuration File, Chroma automatically generates "adaptation stubs" to be linked with the application. These stubs reduce the burden of modifying applications, by providing a high-level interface tailored to the application's needs. They also insulate application programmers from changes in the underlying runtime system and API: the stub generator, and not the application, will track these changes.

Chroma is part of the larger *Aura* [80] project, which addresses the challenges of pervasive computing simultaneously at several software layers. In Aura, a *task layer* called *Prism* manages user tasks and the groups of applications that participate in these tasks. The *adaptation layer* (Chroma) translates between application needs and system resources. Lower layers deal with physical variables such as location and resource availability.

10.3 Future Work

This work has opened up several directions for future research. Some of these are modest, incremental improvements to the system; others are broader and more ambitious in scope. Here I explore different areas of improvement, in each case progressing from short-term goals to long-term ambitions.

10.3.1 The Application Programming Interface

The multi-fidelity API is both simple and general. My experience with this API has convinced me that we should retain the simplicity, but *specialize* the API to the needs of each application. E.g., applications should be able to refer to tunable and non-tunable parameters by name, instead of passing them to the system as anonymous arrays of values. The problem of translating between a high-level application-specific interface and the generic multi-fidelity API is being addressed by the Chroma stub generator described in the previous section.

The hint mechanism, though simple and efficient, is not very secure. Binary modules are loaded into the runtime system's address space, where a malicious or buggy module could wreak havoc. One possible solution to this problem is to use a restricted language that is interpreted at runtime. A slightly less secure but more efficient approach would be for a stub generator to compile the restricted code into trusted binary modules.

There are several limitations to be addressed in the multi-fidelity programming model, especially the current simple notion of *interactive operations*. E.g., the API does not currently support multiple concurrent, communicating operations, where the fidelity and performance of one might depend on that of another, even in the absence of resource contention. Another desirable extension is support for *streaming* media applications such as video. Here the low-level "operations" (e.g. render a frame) are not driven by user actions, but are performed periodically. Further, the primary performance metrics are not per-operation measurements such as latency, but aggregate statistics such as frame rate and jitter. We need to expand the notion of operation, and correspondingly the API, to support adaptive streaming media.

10.3.2 The Solver

The multi-fidelity solver works well when the number of tunable parameters is small and the utility functions are simple and well-behaved. We do not know how its performance or accuracy scale with the number of parameters, nor what kind of utility functions might cause pathological behaviour. To measure the solver's robustness and scalability, we need a more comprehensive evaluation. To improve these properties, we probably also need a theoretically sound approach to the optimization problem. Lee's solver [57] is provably efficient, and the constraints on the utility functions are well-defined. However, one of these constraints is very restrictive: all runtime parameters must have an *ordered* set of values, and both resource demand and utility must increase monotonically as we progress through this set. Often, we cannot achieve such an ordering. E.g., for Radiator, the progressive radiosity algorithm has a higher CPU demand but lower memory demand than hierarchical radiosity, making it impossible to find a total order on these two choices. It would be both interesting and useful to adapt Lee's solver to the multi-fidelity runtime system, and to find ways to relax the monotonicity requirement.

10.3.3 Resource monitoring

My resource monitors are simple and demonstrate the feasibility of a gray-box approach to resource supply prediction. They are agile and accurate when the number of applications is small. In the short to medium term, I would like a more comprehensive evaluation with a wider range of applications and load patterns. Here I indicate a few obvious improvements to the resource monitors; a detailed evaluation will probably suggest several more.

Currently, the system monitors memory demand by measuring VM statistics at the beginning and the end of each operation (Section 5.5.2). This approach is limited by the tendency of VM state to build up over time. In particular, it suffers from a *high watermark effect*: if the current operation uses less memory than a previous one, then its working set will be smaller than its resident set, and we will overestimate the former. This effect could be avoided by adding fine-grained information about the application's memory allocation, e.g., by instrumenting $malloc$ and $free$.

The CPU supply predictor assumes that scheduling is approximately round-robin. This is only true for processes with equal priority: we need to extend the predictor to predict scheduler behaviour for processes with different priorities.

My current prototype lacks supply and demand monitoring for an important system resource: the disk. While disk capacity is easily modelled as a space-shared resource (Chapter 2), disk *bandwidth* is a more complex matter: it is time-shared, but it does not fit the GPS model very well. Disk "supply" depends not only on bandwidth share, but also on seek and rotational positions. Similarly, "demand" is characterized not only by read/write byte counts, but also by the position of those bytes on the disk. An accurate resource model for disk bandwidth must model these additional dimensions to supply and demand, as also the effect of any memory caches inside the storage device.

The ultimate goal of resource management is to combine *adaptation* with *allocation*, such that both system-level and application-level decisions are simultaneously optimized to maximize user utility. I.e., a single decision-making process should determine both the sharing of resources across applications, and the tunable parameters of each application. Section 9.2.1 discussed various allocation-based strategies, and how they might be combined with multi-fidelity adaptation. There are several challenges that any such combined solution must address:

- The mobile client's OS does not have control over resources such as network bandwidth or remote server cycles; thus, we must reconcile ourselves to a predictive approach for these resources, or solve the problem of distributed resource allocation.
- It is not clear how to combine individual application utilities into a single "user utility". Applications run at different time scales: how are we to compare the utility of 10 GLVU operations against that of 1 speech recognition of the same duration. Further, applications compete not only for system resources, but also for *user attention*. This can cause strange interference effects: e.g., the user might desire high-quality video if that is the only application running, but will not notice a degradation in video quality if they are occupied with another task.
- If allocation is tied to adaptation, how can we prevent ill-behaved applications from hogging resources, perhaps by generating bogus utility functions? I.e., how do we detect when applications lie about their resource needs?

10.3.4 Resource demand prediction

This work has shown that history-based techniques — logging and learning — can accurately predict an application's resource demand as a function of its tunable parameters. There is enormous scope for future research in this area. I have used very simple statistical models such as least-squares regression; it would be worthwhile to study other machine learning (ML) algorithms such as nearest neighbour [63, pp. 231–236], and to quantify their accuracy as well as their performance overheads.

Converting resource logs into predictors still requires manual intervention to choose an ML technique and a parametric model. To automate the process further, I propose a *toolkit* approach: a library of many parametric models, over which a "meta-learner" can iterate to find the one most appropriate for the given data.

Building such a toolkit would be an excellent opportunity for collaboration between ML and systems researchers. ML algorithms will help systems builders characterize and predict the behaviour of large systems; logs from such systems will provide real-world test cases for these algorithms.

10.3.5 User Interaction

Section 9.3 discussed the problem of knowing user preferences. We wish users to be able to set their preferences (e.g., desired response time) dynamically. To this end, we have started building a *multi-fidelity control interface* that allows users to adjust application utility functions: to change latency constraints, desired battery lifetimes, or the relative weights of different quality metrics. Building such an interface is a challenge: it must be rich enough to gather information about user preferences along multiple dimensions, but simple and unobtrusive enough to avoid excessive user distraction.

Ideally, the system would automatically infer the user's preferences from their context, avoiding user distraction altogether. E.g., the system would know that users watching a sports video prefer high frame rates to high quality images; users on a video tour of a museum want precisely the opposite. Current work on the *task layer* of Aura [80] is focussed on this issue: identifying the user's higher-level tasks, inferring the appropriate application settings and tradeoffs, and translating them into utility functions for the runtime system to optimize?

Whether we solicit user preferences or infer them, we must always be aware of the attendant cost: increased *user distraction*. In the former case, there is the cognitive burden of manipulating a user interface. In the latter case, there is the risk of erroneous inferences causing unpredictable and unstable behaviour. *User attention* is often the most valuable resource in any computing system: thus, all system decisions need to be measured against this yardstick in addition to traditional metrics of performance. *Quantifying* user distraction is thus a research project of immense importance, presenting formidable challenges and promising enormous benefits.

10.3.6 Augmented Reality

GLVU and Radiator only partially fulfil the vision of augmented reality contained in the architect scenario. My research concentrates on the resource management aspect of augmented reality; to make the vision come true, other hard problems must be solved. E.g., in addition to 3-D rendering, augmented reality needs location sensing techniques to know the user's position and orientation and machine vision algorithms to recognize real-world objects in the user's field of view; all these must run concurrently and cooperate to provide a good user experience in spite of resource variability. Many of these component technologies are at the cutting edge of research in different fields; this is a perfect moment to begin cross-cutting research collaboration between HCI, image processing, and systems researchers to build a usable augmented reality system.

10.4 Summary

The primary focus of this dissertation was crisp interactive response in resource-intensive, mobile applications. I showed that multi-fidelity computation is essential to dynamic regulation of resource demand and thus of performance. I also showed that system support for supply and demand prediction is necessary to support multi-fidelity computation, and that it is sufficient — with some limitations — to achieve bounded interactive response times without unnecessarily sacrificing fidelity. Of the many directions for future research, probably the most important is that of combining prediction-based adaptation with QoS-based resource allocation.

Bibliography

[1] Tarek F. Abdelzaher. An automated profiling subsystem for QoS-aware services. In *Proceedings of the Sixth IEEE Real-Time Technology and Applications Symposium (RTAS '00)*, pages 208–217, Washington, DC, June 2000.

[2] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS '98)*, pages 4–13, Madrid, Spain, December 1998.

[3] Luca Abeni and Giorgio Buttazzo. QoS guarantee using probabilistic deadlines. In *Proceedings of the 11th IEEE Euromicro Conference on Real-Time Systems*, pages 242–249, York, UK, June 1999.

[4] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. Congressional samples for approximate answering of group-by queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*, pages 487–498, Dallas, TX, May 2000.

[5] K. Appel and W. Haken. Every planar map is four colorable. *American Mathematical Society Bulletin*, 82(5):711–712, 1976.

[6] Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, pages 43–56, Chateau Lake Louise, Banff, Canada, October 2001.

[7] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 45–58, New Orleans, LA, February 1999. USENIX.

[8] Joel F. Bartlett, Lawrence S. Brakmo, Keith I. Farkas, William R. Hamburgen, Timothy Mann, Marc A. Viredaz, Carl A. Waldspurger, and Deborah A. Wallach. *The Itsy Pocket Computer*. Compaq Western Research Laboratory, Palo Alto, CA, October 2000. WRL Technical Note 2000.6.

[9] Ludwig van Beethoven. String quartet no. 16 in F, Op. 135, 1826. Dedicated to Johann Wolfmayer.

[10] George E.P. Box, Gwilym M. Jenkins, and Gregory C. Reinsel. *Time series analysis: forecasting and control*. Prentice Hall, Upper Saddle River, NJ, 3rd edition, 1994.

[11] John Bradley. xv: Interactive image display for the x window system. `ftp://ftp.cis.upenn.edu/pub/xv/docs/xvdocs.pdf`, 1994. (Accessed on July 23 2002).

[12] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Mahwah, NJ, 1983.

[13] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The Information Visualizer, an information workspace. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '91)*, pages 181–188, New Orleans, LA, May 1991.

[14] Surendar Chandra and Carla Schlatter Ellis. JPEG compression metric as a quality aware image transcoding. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems (USITS '99)*, pages 81–92, Boulder, CO, October 1999.

[15] Fangzhe Chang, Ayal Itzkovitz, and Vijay Karamcheti. User-level resource-constrained sandboxing. In *Proceedings of the 4th USENIX Windows Systems Symposium (WSS 2000)*, pages 25–36, Berkeley, CA, August 2000.

[16] Fangzhe Chang, Vijay Karamcheti, and Zvi Kedem. Exploiting application tunability for efficient, predictable resource management in parallel and distributed systems. *Journal of Parallel and Distributed Computing*, 60(11):1420–1445, November 2000.

[17] Fay Chang and Garth A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 1–14, New Orleans, LA, February 1999. USENIX.

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

[19] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Boston, MA, 1990.

[20] Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS-01)*, pages 159–170, Berkeley, CA, March 2001.

[21] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI '88)*, pages 49–54, Saint Paul, MN, August 1988. AAAI Press/MIT Press.

[22] P. J. Denning. The working set model of program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[23] P. Dinda, B. Lowekamp, L. Kallivokas, and D. O'Hallaron. The case for prediction-based best-effort real-time systems. In *Proceedings of the 7th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS '99)*, pages 308–318, San Juan, PR, April 1999. Springer-Verlag.

[24] Peter Dinda and David O'Hallaron. Realistic CPU workloads through host load trace playback. In *Proceedings of the 5th Workshop on Languages Compilers, and Runtime Systems for Scalable Computers (LCR 2000)*, pages 265–280, Rochester, NY, May 2000. Springer-Verlag.

[25] Peter A. Dinda. Online prediction of the running time of tasks. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC '01)*, pages 383–394, San Francisco, CA, August 2001. IEEE Computer Society.

[26] Fred Douglis, Ramon Caceres, Frans Kaashoek, Kai Li, Brian Marsh, Joshua A. Tauber, and D. E. Shaw. Storage alternatives for mobile computers. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 25–37, Monterey, CA, November 1994. USENIX.

[27] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS '96)*, pages 25–36, Philadelphia, PA, May 1996.

[28] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. The limited performance benefits of migrating active processes for load sharing. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS '88)*, pages 63–72, Santa Fe, NM, May 1988.

[29] 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, NY, May 1993.

[30] 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, September 1994.

[31] J. Flinn and M. Satyanarayanan. PowerScope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '99)*, pages 2–10, New Orleans, LA, February 1999.

[32] Jason Flinn. *Extending Mobile Computer Battery Life through Energy-Aware Adaptation*. PhD thesis, Carnegie Mellon University, 2001.

[33] Jason Flinn, Dushyanth Narayanan, and M. Satyanarayanan. Self-tuned remote execution for pervasive computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 61–66, Schloss Elmau, Germany, May 2001. IEEE Computer Society.

[34] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile application. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 48–63, Kiawah Island, SC, December 1999.

[35] G. H. Forman and J. Zahorjan. The challenges of mobile computing. *IEEE Computer*, 27(4):38–47, April 1994.

[36] Armando Fox and Eric A. Brewer. Harvest, yield and scalable tolerant systems. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 174–178, Rio Rico, AZ, March 1999. IEEE Computer Society.

[37] 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 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*, pages 160–170, Cambridge, MA, October 1996. ACM Press.

[38] Alan Frieze, Ravi Kannan, and Santosh Vempala. Fast Monte-Carlo algorithms for finding low-rank approximations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS '98)*, pages 370–378, November 1998.

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

[40] Michael Garland. QSlim 2.0 source code and online documentation. `http://graphics.cs.uiuc.edu/~garland/software/qslim.html`, March 1999. (Accessed on July 23 2002).

[41] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH '97*, pages 209–216, Los Angeles, CA, August 1997. Addison Wesley.

[42] Carl Friedrich Gauss. *Theoria Combinationis Observationum Erroribus Minimum Obnoxiae*. Royal Society of Göttingen, 1821. Reprinted by SIAM Classics in Applied Mathematics, 1995, with English translation by G.W. Stewart.

[43] The Independent JPEG group. `http://www.ijg.org/`. (Accessed on July 23 2002).

[44] Richard Han, Pravin Bhagwat, Richard LaMaire, Todd Mummert, Veronique Peret, and Jim Rubas. Dynamic adaptation in an image transcoding proxy for mobile web browsing. *IEEE Personal Communications*, 5(6):30–44, December 1998.

[45] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS '94)*, pages 13–24, Nashville, TN, May 1994.

[46] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '97)*, pages 171–182, Tucson, AZ, May 1997.

[47] E. Horvitz. Principles of mixed-initiative user interfaces. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*, pages 159–166, Pittsburgh, PA, May 1999.

[48] David Hull, W. Feng, and Jane W. S. Liu. Operating system support for imprecise computation. In *Proceedings of the 1996 AAAI Fall Symposium on Flexible Computation in Intelligent Systems: Results, Issues, and Opportunities*, pages 9–11, Cambridge, MA, November 1996. AAAI Press/MIT Press.

[49] Intel, Microsoft, and Toshiba. Advanced configuration and power interface specification v2.0. `http://www.acpi.info/`, February 1998. (Accessed on July 23 2002).

[50] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric query optimization. *VLDB Journal: Very Large Data Bases*, 6(2):132–151, May 1997.

[51] Van Jacobson. Congestion avoidance and control. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM '88)*, pages 314–329. ACM Press, August 1988. Stanford, CA.

[52] Kevin Jeffay and David Bennett. A rate-based execution abstraction for multimedia computing. In *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 64–75, Durham, NH, April 1995. Springer-Verlag.

[53] Nirav H. Kapadia, José A. B. Fortes, and Carla E. Brodley. Predictive application-performance modeling in a computational grid environment. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC '99)*, pages 47–54, Los Angeles, CA, August 1999.

[54] R. H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1):6–17, 1994.

[55] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions of Computer Systems*, 10(1):3–25, February 1992.

[56] Milan Kundera and Michael Henry Heim (Translator). *The Unbearable Lightness of Being*. Harper & Row, 1984.

[57] Chen Lee, John Lehoczky, Dan Siewiorek, Raghunathan Rajkumar, and Jeff Hansen. A scalable solution to the multi-resource QoS problem. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS '99)*, pages 315–326, Phoenix, AZ, December 1999.

[58] Will E. Leland and Teunis J. Ott. Load-balancing heuristics and process behavior. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS '86)*, pages 54–69, Raleigh, NC, May 1986.

[59] Hanan Lutfiyya, Gary Molenkamp, Michael Katchabaw, and Michael Bauer. Issues in managing soft QoS requirements in distributed systems using a policy-based framework. In *Proceedings of the Workshop on Policies for Distributed Systems and Networks (Policy 2001)*, pages 185–201, Bristol, UK, January 2001. Springer-Verlag.

[60] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '98)*, pages 426–435, June 1998.

[61] Bob Marley. Could You Be Loved (12" Mix). Island Records, 1980. 5:26 min.

[62] Robert B. Miller. Response time in man-computer conversational transactions. *AFIPS Fall Joint Computer Conference Proceedings*, 33:267–277, December 1968.

[63] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

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

[65] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 2nd Symposium*

on Operating Systems Design andImplementation (OSDI '96), pages 3–17, Seattle, WA, October 1996. USENIX.

[66] David J. Musliner, Edmund H. Durfee, and Kang G. Shin. Any-dimension algorithms. In *Proceedings of the 9th IEEE Workshop on Real-Time Operating Systems and Software (RTOSS '92)*, pages 78–81, May 1992.

[67] Klara Nahrstedt, Dongyan Xu, Duangdao Wichadukul, and Baochun Li. QoS-aware middleware for ubiquitous and heterogeneous environments. *IEEE Communications*, 39(11):140–148, November 2001.

[68] Rolf Neugebauer and Derek McAuley. Congestion prices as feedback signals: An approach to QoS management. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 91–96, Kolding, Denmark, September 2000.

[69] Allen Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, 1990.

[70] Sergei Nirenburg, editor. The PANGLOSS Mark III machine translation system. Technical Report CMU-CMT-95-145, Carnegie Mellon University Center for Machine Translation, April 1995.

[71] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 276–287, Saint Malo, France, October 1997.

[72] 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)*, pages 51–62, Cannes, France, September 1997.

[73] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[74] David Petrou and Dushyanth Narayanan. Position summary: Hinting for goodness' sake. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 177–177, Schloss Elmau, Germany, May 2001. IEEE Computer Society.

[75] The Walkthru Project. GLVU source code and online documentation. `http://www.cs.unc.edu/~walk/software/glvu/`, February 2002. (Accessed on July 23 2002).

[76] L. Rabiner and B.-H. Juang. *Fundamentals of Speech Recognition*. Prentice Hall Signal Processing Series. Prentice Hall, Upper Saddle River, NJ, 1993.

[77] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and J. Bradley Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–7, Berkeley, CA, August 1997.

[78] Edward Said. *Representations of the Intellectual*. Vintage Books, 1993.

[79] M. Satyanarayanan. Mobile information access. *IEEE Personal Communications*, 3(1):26–33, February 1996.

[80] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, August 2001.

[81] M. Satyanarayanan, Jason Flinn, and Kevin R. Walker. Visual Proxy: Exploiting OS customizations without application source code. *ACM Operating Systems Review*, 33(3):14–18, July 1999.

[82] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight recoverable virtual memory. In *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP '93)*, pages 146–160, Asheville, NC, December 1993.

[83] M. Satyanarayanan and Dushyanth Narayanan. Multi-fidelity algorithms for interactive mobile applications. *Wireless Networks*, 7:601–607, 2001.

[84] Steve Schlosser, John Griffin, Dave Nagle, and Greg Ganger. Designing computer systems with mems-based storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 1–12, Cambridge, MA, November 2000. ACM Press.

[85] Walter Carruthers Sellar and Robert Julian Yeatman. *1066 and All That: A Memorable History of England*. Methuen & Co. Ltd., London, UK, 1930.

[86] Margo Seltzer and Christopher Small. Self-monitoring and self-adapting systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 124–129, Cape Cod, MA, May 1997. IEEE Computer Society.

[87] Bruce S. Siegell and Peter Steenkiste. Automatic generation of parallel programs with dynamic load balancing. In *Proceedings of the 3rd IEEE International Symposium on High Performance Distributed Computing (HPDC '94)*, pages 166–175, San Francisco, CA, August 1994.

[88] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94)*, pages 196–205, Orlando, FL, June 1994.

[89] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 145–158, New Orleans, LA, February 1999. USENIX.

[90] Neil Stratford and Richard Mortier. An economic approach to adaptive resource management. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 142–147, Rio Rico, AZ, March 1999. IEEE Computer Society.

[91] Gauri Viswanathan. *Masks of Conquest: Literary Study and British Rule in India.* Columbia University Press, New York, NY, 1989.

[92] A. Waibel. Interactive translation of conversational speech. *IEEE Computer*, 29(7):41–48, July 1996.

[93] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 1–11, Monterey, CA, November 1994. USENIX.

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

[95] A. A. Webster, C. T. Jones, M. H. Pinson, S. D. Voran, and S. Wolf. An objective video quality assessment system based on human perception. In *Proceedings of SPIE: Human Vision, Visual Processing, and Digital Display IV*, pages 15–26, February 1993.

[96] Andrew J. Willmott. Radiator source code and online documentation. `http://www.cs.cmu.edu/~ajw/software/`, October 1999. (Accessed on July 23 2002).

[97] G. Wilson and M.A. Sasse. Do users always know what's good for them? Utilising physiological responses to assess media quality. In *Proceedings of the 14th Annual Conference of the British HCI Group (HCI 2000)*, pages 327–339, Sunderland, UK, September 2000. Springer-Verlag.

[98] Peter Young. *Recursive Estimation and Time-Series Analysis*. Springer-Verlag, Heidelberg, Germany, 1984.

[99] Jamie Zawinski. Remote control of Unix Netscape. `http://home.netscape.com/newsref/std/x-remote.html`, December 1994. (Accessed on July 23 2002).

[100] T.C. Zhao and Mark Overmars. Xforms home page. `http://world.std.com/~xforms/`, 1995. (Accessed on July 23 2002).

Appendix A

Unrelated Work

Much excellent research is more or less irrelevant to the focus of this dissertation. Here we briefly survey some unrelated work.

The Four-Colour Theorem [5] is a pioneering example of computer-aided theorem-proving. It hardly influenced our research. Said [78] articulates a vision of the intellectual "as exile and marginal, as amateur, and as the author of a language that tries to speak truth to power", while Viswanathan [91] argues convincingly that the development of English Studies as an academic discipline is closely related to the political imperatives of the British Empire in India. This dissertation does not build on these analyses.

Some questions that have rarely plagued the systems community have been answered, e.g. *"Muss es sein?"* [9, 56]. Other problems, of equally negligible import for the future of our field, remain unsolved. We note that, despite the efforts of many researchers, the problem of "*Could You Be Loved?*" [61] remains open. Last and probably least, Sellar and Yeatman [85] have shown conclusively that history is not what you think it is, it is *what you can remember*. We have not improved on this result in any way.