In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature______________________________

Date__________________________________

University of Washington

Abstract

Obtaining Responsiveness in Resource-Variable Environments

by George H. Forman

Chairperson of Supervisory Committee: Professor John Zahorjan
Computer Science and Engineering

This research addresses the problem of building software that can be responsive even in environments with widely variable or unknown resource availability. Application programmers' efforts to ensure responsiveness are rendered increasingly difficult by a number of trends in modern computing environments that increase service variability, including:

1. wide scale sharing of resources, such as distributed file systems,

2. wireless networking for mobile computers, and

3. sporadic inclusion of multimedia in documents.

The confluence of these trends calls for broadly applicable programming support to create applications that ensure good responsiveness in the face of widely variable service times at little or no increase in programming effort relative to the existing technology (which cannot provide this level of responsiveness). The work described here provides the design of such support, and furnishes a proof of concept that it

is effective where (a) portions of the application can be decomposed into concurrent tasks, and (b) the application can produce and present its results incrementally, i.e., can trade the quality of the response against the delay required to provide it, such as with multi-resolution techniques.

The key to this approach is the runtime construction of a global dependence graph that relates tasks to each other through their input and output dependencies. This graph is constructed by an application-independent system with minimal programmer effort, and is used at runtime to provide useful services, including:

1. adjusting resource allocation along the branches of the graph to improve response time for those tasks on which the user is currently focussed,

2. pruning branches of the graph that no longer represent useful work, thus freeing resources for the computations of interest,

3. managing multiple results of improving quality and their storage behind the abstraction of program variables, and

4. synchronizing the execution of threads on new input quality versions according to programmer-specified semantics.

The combined effect is that it is easier to write applications that provide good responsiveness across a wide and unpredictable range of service variability.

TABLE OF CONTENTS

List of Figures v

List of Tables vii

Chapter 1: Introduction 1

1.1 An Example of Responsiveness Despite Lagging Resources 2

1.2 Synopsis 3

1.3 Contributions 5

Chapter 2: Service Variability 7

2.1 Wide Range of Service Variability 7

2.1.1 Mobile Computing $\Longrightarrow$ Variable Network Bandwidth 8

2.1.2 Variable Computational Performance 12

2.1.3 Variable User Congestion 13

2.1.4 Variable Data Magnitude 14

2.1.5 Compounding 15

2.2 Volatility of Service Variability 15

2.2.1 An Experiment to Characterize Volatility 16

2.2.2 Presentation of Results 18

Chapter 3: Existing Approaches to Obtaining Responsiveness 26

3.1 Approaches to Obtaining Responsiveness Despite Volatility 29

3.1.1 Feedback Before Completion— Incremental Quality 29

3.1.2 Asynchronous Operation 30
3.1.3 Resource Allocation 31
3.1.4 Cancellation 32
3.2 Existing Support 33
3.3 Summary of the Motivation 35

Chapter 4: The Petra-Flow Framework 36
4.1 Key Ideas 36
4.2 Central Components 40
4.3 Hypothetical Example: Slide Browser Application 42
4.4 Uses of the Derived Global View 48
4.4.1 Resource Allocation 49
4.4.2 Incremental Quality Results 51
4.4.3 Obsolete branch pruning 55
4.5 Priority-Mediated Resources 57
4.6 Summary 62

Chapter 5: A Prototype Petra-Flow Implementation 63
5.1 Language and Operating System 64
5.2 Annotations for Asynchronous Tasks 65
5.3 Runtime Library 68
5.3.1 Versioned Variables 68
5.3.2 Priorities 70
5.3.3 Reference Counter 71
5.3.4 Task Services 72
5.4 Debugging Support 72
5.5 Conclusion 75

Chapter 6: Evaluation **77**
6.1 Applications . . . 78
6.1.1 Database Front-End . . . 78
6.1.2 Mandelbrot Fractal Explorer . . . 83
6.1.3 Photo Album Web Browser . . . 89
6.1.4 Testing Tool for Prioritized Network Traffic . . . 94
6.2 Static Analysis . . . 95
6.2.1 Coding Effort . . . 95
6.2.2 Comparative Survey of Other Programs . . . 96
6.2.3 Storage Overhead . . . 100
6.3 Runtime Analysis . . . 103
6.3.1 Macroscopic Runtime Overhead . . . 103
6.3.2 Micro-Benchmark Comparison with Raw Pthreads . . . 105
6.3.3 Evaluation of Techniques Supported by Petra-Flow . . . 107
6.4 Simulation Analysis of Priority-Mediated Locks . . . 114
6.4.1 Workload and Parameterization . . . 114
6.4.2 Statistics and Stopping Criteria . . . 116
6.4.3 Results . . . 116

Chapter 7: Conclusion **119**
7.1 Generalizations . . . 120
7.2 Related Work . . . 122
7.2.1 Asynchronous Programming . . . 122
7.2.2 Resource Allocation . . . 123
7.2.3 Responsiveness . . . 125
7.3 Future Work . . . 126
7.4 Conclusion . . . 127

Bibliography **128**

LIST OF FIGURES

1.1 *User Responsiveness: Netscape vs. Mosaic during Web page download.* 3

2.1 *Cost vs. Range of 1–3 Mbit/s, 2.4 GHz Wireless Products.* 9

2.2 *The timing measurements taken for each request.* 16

2.3 *Raw Timing Measurements of the Web* 19

2.4 *Histogram of Connection Establishment Times* ($T1$) *of All Datasets.* . . 21

2.5 *Coefficient of Variation of* Ta *for Batches from Dataset* `yahoo`. 21

2.6 *Coefficient of Variation of Batches Averaged Across Time.* 22

2.7 *Standard Deviation of Batches Averaged Across Time.* 23

2.8 *CDF of* Ta *Divided by Median* Ta *of its Batch.* 24

3.1 *Static Sizing: Qualitative Graph of Quality vs. Response Time.* 26

4.1 *Pseudo-Code for Hypothetical Slide Browser Application.* 42

4.2 *Dependence Graph Transitions Showing Variable Splitting.* 43

4.3 *Pseudo-Code for Enhanced Slide Browser.* 45

4.4 *Dependence Graph Transitions Showing Priority Up-Flow.* 46

4.5 *Summing Priority Values vs. Unioning Priority Token-Sets.* 49

4.6 *Detailed Petra-Flow Graph.* 56

4.7 *Priority-Mediated Locks: Deferment Avoids Priority Inversion.* 59

4.8 *Priority-Mediated Locks: Legend to Diagrams.* 59

4.9 *Priority-Mediated Locks: Priority Shift Can Unblock a Request.* 60

4.10 *Priority-Mediated Locks: Retained Interest Avoids Priority Inversion.* 61

4.11 *Priority-Mediated Locks: Resilient Under Dynamic Priority Changes.* 61

5.1 *Petra-Flow Pre-processing and Compilation Process.* 63
5.2 *Petra-Flow Producer-Consumer Example.* 69
5.3 *Producer-Consumer Dependence Graph Snapshot Viewed with Dotty.* . . 73

6.1 *Snapshot of the Database Front-End.* 79
6.2 *Example Dependence Graph for Database Front-End.* 83
6.3 *Program Structure of* `pDatabase`. 84
6.4 *Snapshot of the Mandelbrot Fractal Explorer.* 85
6.5 *Example Dependence Graph for Fractal Explorer.* 87
6.6 *Program Structure of* `pFractals`. 88
6.7 *Snapshot of the Photo Album Web Browser.* 89
6.8 *Example Dependence Graph for Photo Album Browser.* 91
6.9 *Program Structure of* `pAlbum`. 92
6.10 *Snapshot of the Testing Tool for Prioritized Network Traffic.* 94
6.11 *Incremental Quality Results: Quality vs. Response Time.* 108
6.12 *Prioritization: Quality vs. Response Time.* 109
6.13 *Cancellation: Quality vs. Response Time.* 111
6.14 *Slowdown vs. Polling Granularity.* 113
6.15 *Simulation Results: Priority Inversion vs. Waste.* 117
6.16 *Simulation Results: Priority Inversion vs. Competing Threads.* 118

7.1 *Mobile Computing Models in Terms of Where the Wireless Network Cuts the System.* 122

LIST OF TABLES

2.1 *Mobile (Wireless) and Non-Mobile Network Technologies* 11

2.2 *Server, Request and Data Volume Returned for each Dataset.* 17

5.1 *Choice of Synchronization Semantics.* 65

6.1 *Coding Effort of Petra-Flow Applications.* 96

6.2 *Comparison of Interactive Mandelbrot Fractal Generator Programs.* . . 97

6.3 *Storage Sizes.* 100

6.4 *Macroscopic Measurement of Overhead (as percentage).* 104

6.5 *Micro-Benchmark Comparison to Raw Pthreads (microseconds).* 105

ACKNOWLEDGMENTS

I would like to thank John Zahorjan, my advisor, not only for his attention to detail in honing my writing and my research, but also for making an effort to teach me the worldly ways of the field. I also appreciate the investment of time and attention from the other committee members, Brian Bershad, Alan Borning, Blake Hannaford, and David Notkin.

This department has been a great place for graduate studies for me, entirely to the credit of the individual people it's comprised of. Among those I wish to thank for making my time here enjoyable and challenging are Robert Bedichek, Nancy Burr, Sung Choi, E Chris, Frankye Jones, David "Pardo" Keppel, Calvin Lin, Erik Lundberg, Dylan McNamee, Gail Murphy, Brian Pinkerton, Larry Ruzzo, Stefan Savage, Alan Shaw, Gun Sirer, Larry Snyder, Geoff Voelker, and Terri Watson.

Finally, I am blessed with a wonderful wife, Deirdre, whose love and support through it all gave me stamina and preserved the fun in my work.

Chapter 1

INTRODUCTION

Application programs are beginning to run up against a new impediment to providing users with satisfactory performance: several trends in modern computing environments are resulting in greatly variable service delivery. Since today's highly interactive applications depend on timely response from the underlying resources, if resource response time varies dynamically, responsiveness to the user suffers. Although traditional programming methods often deal adequately with high *mean* service time, they fall short in coping with high *variance*, choosing either to ignore the potential periods of high resource availability or to suffer through periods of low availability with inadequate responsiveness [34]. While there are several techniques for providing users with responsive behavior in the presence of service delays, programming applications using these techniques involves significant extra effort beyond implementing the base application. *The goal of this research is to develop broadly applicable support to aid programmers in constructing and executing interactive applications that exhibit good responsiveness even in environments where available system resources are highly variable or unknown at the time of writing.*

To help clarify the high-level purpose of this research, consider a similar service: automatic garbage collection. Although programmers can certainly implement their own memory management by hand using known techniques, it is valuable to have automatic facilities that provide this service. By providing reusable, application-independent support for memory management, we can lighten the burdens of many

programmers in writing and maintaining this tedious, error-prone aspect of their codes, meanwhile instilling greater confidence that their applications will not exhibit memory leaks or dangling pointers.

In a like manner, this dissertation develops the design of reusable, application-independent programmer support that yields similar benefits in the domain of insulating the user from the widely variable delays inherent in modern computing environments. The notion of a resource-variable environment, and what can be done to provide responsiveness to the user in such an environment, will be introduced through an example.

1.1 An Example of Responsiveness Despite Lagging Resources

Demand for features that help ensure responsiveness will grow in the near future as applications are used in increasingly variable environments. A common example of such an environment is the World Wide Web, where the response time of a request can be hard to predict and highly variable; even the rate at which data is returned can vary widely during the transfer.

To illustrate what is meant here by good responsiveness despite slow resources, we demonstrate one of the striking differences between two World Wide Web browsers, Netscape and Mosaic. Figure 1.1 shows a snapshot of each browser in the process of downloading the same Web page. In Netscape we see almost all the content of the page, but in Mosaic we see nothing at all.[1] What is the difference? Surely Netscape cannot make the Internet run faster. Netscape employs certain features to make the user interface more responsive. Among other things, it displays the Web page incrementally as it arrives, showing inlined images in coarse resolution initially and

---

[1] At monitor resolution, one can easily see that the Netscape image quality is rough and "blocky." The lack of detail is less noticeable in this reduced reproduction.

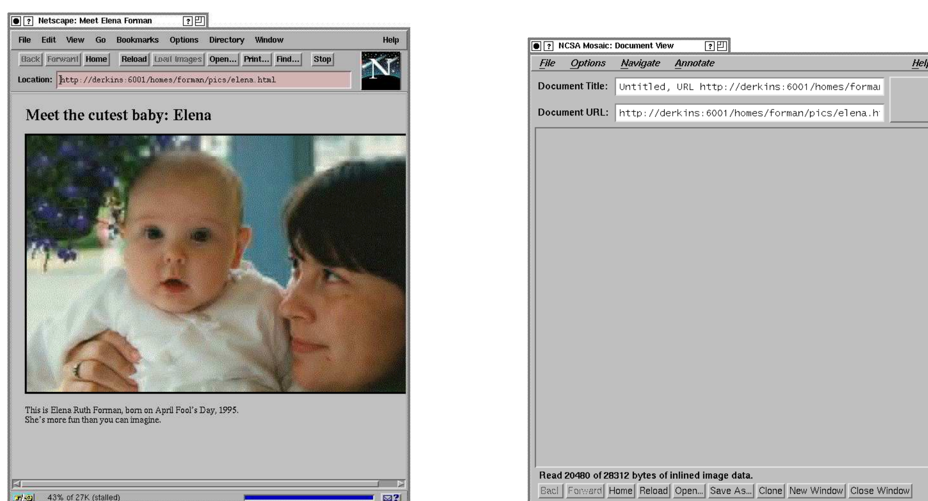

Figure 1.1: *User Responsiveness: Netscape vs. Mosaic during Web page download.*

improving them when higher resolution data becomes available.[2] In the figure, less than half of the image data has been received. On the other hand, Mosaic displays none of the image or accompanying text until all downloading has completed. The difference in responsiveness between these two browsers is particularly apparent to users when there are many inlined images, but with only a few initially visible at the top of the Web page.

1.2 Synopsis

Incremental processing and display of information is only one of a variety of techniques that can improve responsiveness when resources lag. Further techniques are surveyed in Chapter 3. The programming effort required to implement such "responsiveness-enhancing" features is dramatically reduced by a new framework introduced in Chapter 4, the principal innovative contribution of this research. In brief, it

[2] This feature requires that the image be delivered in a progressive-resolution format, such as "interlaced GIF" or "progressive JPEG." Such encoding is currently performed statically, before the Web server delivers the image.

consists of three fundamental programming abstractions that facilitate the expression of asynchronous, application-level units of work, including their synchronization and resource requirements. The use of these abstractions produces at runtime a dependence graph among the concurrent tasks that describes their data flow and synchronization structure, as well as the resource conflicts among them. Using the global view manifested by the dependence graph, an application-independent library can provide a number of services to enhance the responsiveness of the application, such as dynamic, intelligent resource allocation among the tasks, and an analogy to garbage collection that automatically eliminates branches of the dependence graph that become obsolete due to overriding user actions.

Chapter 5 then describes a prototype specification and implementation of this framework. To be concrete, the prototype specification provides the application programmer with a set of abstractions via minimal annotations to procedure headings and several C++ template classes [89]. The construction and maintenance of the runtime dependence graph is done behind these abstractions. The implementation consists of a runtime library and a preprocessor for the annotations that generates procedure stubs to maintain the dependence graph, implement the synchronization semantics chosen by the application programmer, and manage the multi-threading. The runtime library uses the dependence graph to provide responsiveness-enhancing services to the application.

The ideas behind the framework are general; the particular choice of annotations, target language and kernel threads system is not crucial. However, object-orientation does simplify the use of certain framework abstractions.

Chapter 6 validates and evaluates this new framework partly via the construction and measurement of multiple applications using it. Finally, Chapter 7 describes related work and avenues for further exploration.

The conclusion of this chapter enumerates the contributions of this research, while Chapter 2 illuminates the growing importance of this area, expounding on the great

variability of modern environments. Whereas the initial motivation for this research was provided by the inherent variability in a specific environment, *mobile computing*, as discussed in the next chapter, broad variability in service times is experienced in other, more commonplace environments, making this research widely applicable, consequential, and useful for today's applications and environments.

1.3 Contributions

This research makes the following contributions:

- It identifies a trend in modern computing environments toward widely variable and dynamically changing service delivery, and catalogs the fundamental sources of this trend.

- It characterizes via experiment the variability in service delivery on today's World Wide Web, and is the first published study to measure its *short term* volatility in service delivery.

- It recognizes the effect of this variability as a growing problem for application programmers and an upcoming research area: that of designing support for producing applications that are consistently responsive to the user despite widely variable or unknown resource availability.

- It proposes an innovative framework to fulfill this need for a broad class of applications.

- It demonstrates the viability of this framework through the construction of a working prototype and measurements of multiple test applications built using the prototype.

- It perceives the opportunity for optimization presented by the capacity of users to get ahead of resources when resources lag, and suggests the use of a dynamic dependence graph of on-going tasks as a means for global optimization for user responsiveness.

- Finally, it contributes a new form of priority-arbitrated lock that helps avoid priority inversion in common situations that arise in incremental processing.

Chapter 2

SERVICE VARIABILITY

Responsiveness to the user appears to have a profound influence on the usability of an application [75, 25]. The importance of a fast response can even exceed that of a complete response. For example, when riffling through an online document, a snappy display can be more important than accurately portraying every detail.

Good responsiveness is characterized by low *mean* response time as well as low *variance*. Low variance makes the interface more predictable and helps users calibrate their expectations. The importance of this is highlighted by an early study that showed that users prefer lower variance even when they must endure a longer mean response time [65].

Unfortunately, the trend in modern computing environments is toward greater variance in service delivery, the topic of this chapter. The first section describes the wide range of variation, whereas the second section characterizes the dynamic nature of this instability.

2.1 Wide Range of Service Variability

In this section we enumerate four fundamental sources of variability in modern environments, focusing on their range of variation: network bandwidth, computational performance, user congestion, and data magnitude. These stem from corresponding trends in modern computing: mobile and wide-area networking, hardware platform diversity, shared resources, and multimedia.

2.1.1 Mobile Computing $\Longrightarrow$ Variable Network Bandwidth

Advances in wireless networking technology have engendered a new paradigm of computing, called *mobile computing*, in which users carrying portable devices have access to a shared infrastructure independently of their physical location and movement [33, 76]. This provides flexible communication between people and continuous access to networked services. The importance of good responsiveness is paramount to mobile computing, where people carry the devices for the express purpose of having "anywhere anytime" access to remotely available information and to other people (through use as telephones, pagers, and electronic-mail devices). Indeed, user studies conducted by Solomon of Hewlett Packard Research Laboratories confirm that users demand instantaneous response from their mobile devices [88].

The thorn here is that network quality on a mobile computer is less stable and subject to a wider range of variation in performance than traditional networked computers, which are statically connected to a single network. As a mobile computer leaves the broadcast range of one wireless connection point, or *cell*, it switches to another cell to maintain its network connections. The applications using these connections may experience extreme changes in available bandwidth for several reasons:

1. Wireless bandwidth is shared among a dynamically changeable user population. For example, an application might experience a drop in available bandwidth of 30:1 in a classroom as students arrive.

2. Different cells may use transceivers of differing quality, for example, the wireless equipment installed in a meeting room may have greater capacity than that in a hallway. Although such differences in equipment may be intended to compensate for localized usage patterns, rarely can the hardware deployment precisely match the difference in workload, especially since workload often

varies by time of day. Thus, applications experience a shift in communication performance when crossing cells of different capabilities.

3. Different cells may vary in size by orders of magnitude in order to establish wireless coverage over a large area and simultaneously provide high performance in key locations. For example, a single cell might cover a meeting room, a floor of a building, a campus (cellular telephony networks), or several states (satellite coverage). A useful metric for a wireless network cell is its *bandwidth per area (or volume) covered*, because available bandwidth per user decreases as cell size increases [101, 73]. This happens both because larger cells can serve more users and because engineering constraints make high speeds over large distances more expensive. To illustrate this, observe the very rough correlation between cost and indoor/outdoor range in Figure 2.1 where we plot a number of wireless products on the market today (that provide 1–3 Mbit/s bandwidth and operate at 2.4 GHz; this restricts the sample to approximately equivalent technologies). Cell size also affects the variance in user population, larger cells enjoying the

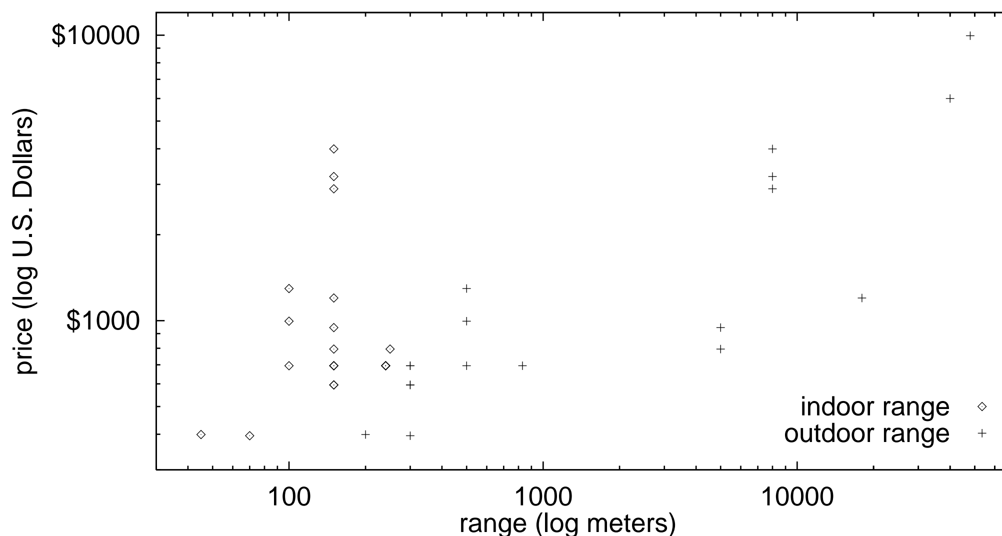

Figure 2.1: *Cost vs. Range of 1–3 Mbit/s, 2.4 GHz Wireless Products.*

statistical properties of larger numbers of users.

4. Applications may experience extreme network variation if the mobile device is also capable of plugging into a wired network at times for a faster or cheaper connection, such as while a user is at his or her desk. Table 2.1 shows the raw bandwidths of a number of mobile (wireless) and non-mobile network technologies. Observe the bandwidth gap between mobile and non-mobile interfaces. There is a discrepancy of two orders of magnitude between high end wired vs. high end mobile wireless technologies. As a vivid illustration of this gap, an application would experience a four order of magnitude drop when unplugging from a wired 155 Mbit/s ATM interface and switching to a wireless 19,200 bit/s cellular digital packet data (CDPD) modem [37]. (Note also that establishing the cellular connection may involve a significant setup delay.[1])

   Although wireless technology will certainly advance, there will continue to be a bandwidth discrepancy compared with wired networking because of the inherent differences in engineering complexity [33, 76, 77].

5. Unlike wired networks, wireless networks are subject to sporadic electromagnetic interference (e.g., microwave ovens radiate in the public 2.4 GHz band) and signal blockage as users move about in the physical environment. Applications may experience such intermittent disconnections as extremely high variance in network latency when, say, a user drives through a tunnel, momentarily obscuring the signal from the cell transceiver.[2]

In summary, applications on mobile computers must be prepared for widely varying

---

[1] CDPD call setup time is a few seconds, versus 20–30 seconds typical with analog cellular modems.

[2] The mobile computing paradigm also admits the possibility of *disconnected operation* for extended periods, say, in areas with no wireless coverage or for savings in connection cost and power consumption in the mobile device. This thesis does not directly address disconnection.

Table 2.1: *Mobile (Wireless) and Non-Mobile Network Technologies Sorted by Raw Bandwidth. (Much of this information is drawn from the "Wireless LAN/MAN Modem Product Directory" at* `http://hydra.carleton.ca/info/wlan.html`*. Other sources include [14, 37, 40, 55].)*

Bandwidth (bits/second)	Coverage, or Indoor/ Outdoor Range (m)	Technology
2400	world	Motorola Iridium satellite service
4800	city	RAM Mobile Data
9600-14.4 K	city	modem over analog cellular telephone
9600-19.2 K	building	AT&T Definity Wireless Business System
9600-28.8 K	400 m	Metricom Ricochet
19.2 K	city	Cellular Digital Packet Data (CDPD)
19.2 K	city	ARDIS Air
115 K, 1-4 M	1 m	IRDA (infrared data standard)
215-860 K	300 m / 10 km	Aironet ARLAN 690-900
230 K	45 m / 9 km	Telesystems ARLAN 450
312 K	n/a	Bell Labs SWAN Project
500 K - 1 M	250 m	IBM Wireless LAN
1 M	6 m	IBM Infrared Wireless LAN
1 M	6 m	Photonics Infrared
1 M	45 m / 200 m	Xircom Netwave
1.6 M	150 m / 300 m	Proxim RangeLAN2
1-2 M	150 m / 5 km	Aironet ARLAN 690-2400
2 M	240 m	NCR WaveLAN
5.7 M	80 m / 2.9 km	Windata FreePort/AirPort II (not mobile)
10 M	(wired)	Ethernet
100 M	(wired)	Fast Ethernet
100 M	(wired)	FDDI
155 M	(wired)	ATM

performance if they involve communication.

Further, an application running on a mobile computer is *likely* to be dependent on the performance of the network and remote servers. This is due to the nature of mobile computers as communicators and information utilities, and also due to a greater dependence on remote resources than is traditional. Because of portability and battery constraints, mobile computers have much more moderate computing resources than stationary computers [33, 76]. Consequently, it is natural to compensate for the lack of on-board resources by employing remote resources, for example, by making use of a remote file system [44] or remote virtual memory paging [78]. This leads to implicit dependence on communication and remote servers, even where none is entailed by the application, making it harder for application programmers to ensure good responsiveness.

2.1.2 Variable Computational Performance

Up to this point we have focussed on the variability caused by one aspect of mobile computing: wireless networking. Mobile computing illustrates another source of uncertainty for application programmers if their applications are to run on both mobile and stationary computers, the difference being that mobile computers have less resources available to them because of their portability and battery constraints. Modern desktop systems often have about twice the computing power of equivalently modern notebook computers, and this difference grows dramatically if we include less-advanced portables [34]. Even if we leave out mobile computers, there is wide variety in performance among the heterogeneous machines in use today. For example, among the various types of desktop workstations in use today here in the Department of Computer Science and Engineering, there is a performance ratio of 25:1 between the high end and low end machines, using SPECint'92 as the performance metric. At the low end, DECstation 3100's are still in use, with a rating of 7.1; at the high end, there are SGI Indigo2's, which are rated at 176. And this performance discrepancy would

more than double with the purchase of certain machines on the market, supposing that the slowest departmental hosts are not instantly jettisoned.

Such broad variation challenges application writers in building software that runs well in diverse environments. A simple approach is to specialize software for particular platforms, but this only works to a limited degree, and is a burden to the consumers, the distributors, and the producers of the software. The hordes swarming to Java these days are attracted particularly because software written in Java should be able to run on diverse platforms without modification [84]. It does not follow that these programs will provide good responsiveness on any but the fastest platforms. The goal of this research is to make such software run *with good responsiveness* on diverse platforms. This is a particularly important property for software that is dynamically downloaded to all manner of hosts in heterogeneous distributed environments, such as with Java Applets or Telescript agents [23, 104].

2.1.3 Variable User Congestion

Service delivery can also vary extensively with user congestion. The impact of this variability is growing with the trend toward wide-area resource sharing and increasingly interactive use of wide-area networking, e.g., the World Wide Web and enterprise intranets. Wide-area resource sharing exposes services to a potentially tremendous user load. If service requests were to arrive independently and under a steady distribution, then it would be feasible to size server resources statically to meet demand with reasonable response time. Evenly distributed requests are atypical, however; workload usually varies substantially with the time of day and the day of the week, and can increase immensely under certain events. For an illustration, while it typically takes about a fifth of a second to download a 20 KB image from a particular Web site, `home.netscape.com`, under high user congestion it sometimes takes up to thirty times as long. And to demonstrate variation around a special event, we repeatedly measured the time to download a 31 KB file from the IRS Web site

www.irs.ustreas.gov. On average, it took 6.9 seconds on April 15th, 1996, when taxes were due, but only 0.86 seconds the following day. (The server was so heavily loaded on April 15th that it was rare even to get a successful TCP connection.) Thus, workload variation and unpredictability makes it uneconomical to endow a service with enough resources so that its response time never deteriorates under load.

2.1.4 Variable Data Magnitude

The final source of service time variability we discuss here is the broad variation in the data magnitude of objects, which proportionally impacts their transfer and processing times. The pervasive multimedia trend is causing some document sizes to grow tremendously. For example, the sporadic inclusion of images in hypertext documents is causing wide differences in the total data size of Web pages, and hence their download times. To exemplify this, we measured for different Web pages the number of bytes of data that fill the browser window when rendered: 850 bytes for a text-only page; 18 KB for a personal home page bearing a small portrait; 67 KB for GNN's corporate page containing a large, clickable image[3]; and 470 KB for a Java Applet showing a six frame animation of the Department's steam powered Turing machine mural[4]. Note the broad range and magnitude of data sizes here; expressed as ratios they are 1:22:80:570, respectively. The discrepancy can easily exceed three orders of magnitude if we include motion video clips; to be quantitative, at the MPEG Movie Archive[5] the average size of a movie clip is 880 KB (the standard deviation being 95% of this figure— again representing great variation).

---

[3] http://www.gnn.com

[4] http://www.cs.washington.edu/education/courses/590s/w96

[5] This movie database resides at http://w3.eeb.ele.tue.nl/mpeg, though it is extensively mirrored throughout the world to provide users with better responsiveness.

2.1.5 Compounding

A final note about the broad range of service delivery time: the extensive variability described in the above subsections can compound, potentially multiplying the cited ratios of variation. Consider the enormous potential for variation in service time when downloading objects of widely diverse sizes across networks of various bandwidths from servers of different hardware capabilities and under varying amounts of user congestion. The bottom line is that the degree of variation is so great that system software cannot hope to hide it from applications. Applications themselves must adapt [66, 77].

2.2 Volatility of Service Variability

While the previous section detailed the *variability* in modern environments, i.e., the wide potential range in service delivery, this section discusses the orthogonal issue of *volatility*: the short term fluctuation in available resources.

The dynamic nature of wireless connectivity certainly introduces a level of dynamic variation not present in statically networked hosts. However, as mobile computing is not yet a well established paradigm, we do not attempt to characterize its volatility here. Instead, we quantify via experiment the volatility of service delivery from the World Wide Web, that is, from remote HTTP servers on the Internet [12]. An extensive range of services are being offered over the Web, including services built upon other Web services, such as shopping agents [24] and the MetaCrawler [28, 81]—they all depend on the lively performance of the Web. A great appeal of the Web is that it provides a ubiquitous, cross-platform user interface with a potentially enormous user population. And with the recent addition of dynamically downloadable Java Applet programs, this interface becomes programmable.

2.2.1 *An Experiment to Characterize Volatility*

While other studies have measured the variability of service delivery of the Web (i.e., with coarse grain measurements in time) [80, 96], our goal in this experiment is to determine its *short term* volatility. To this end, we measure the service delivery of Web requests, or *probes*, repeated in rapid succession and analyze their variation, yielding one *batch*. Figure 2.2 defines the various timing measurements taken for each successive probe. The total elapsed time (Ta) of a request and response is partitioned into ($T1$) establishing the TCP connection, ($T2$) sending the request and receiving the first byte, and ($T3$) reading the remainder of the response. We check the volume of data received to identify and exclude failed probes.

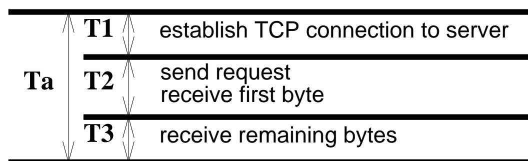

Figure 2.2: *The timing measurements taken for each request.*

Although our method makes multiple requests in rapid succession, it does not impose a large or uncommon load on the Web servers. Popular Web browsers request multiple inline images in parallel, hence our sequential requests are unlikely to exceed the service demands of a genuine user at any given moment. This is important both for public etiquette and so that probe measurements are not overly affected by the impact of our prior probes.

Both to collect enough batches for significance and to observe the variation by time of day and day of the week, we repeated this test every half hour for a period of one week, issuing 20 probes per batch. We selected nine heavily loaded Web servers plus our local departmental server to measure in this manner: totaling three in Europe and seven in North America. Table 2.2 lists for each of the ten server datasets the

type of request made and the number of bytes returned. Three of the datasets use keyword search queries, involving greater load at the server compared with simple file delivery. The different servers were sampled out of phase with one another to avoid influencing each other's measurements at the point of origin, a DEC Alpha 3000/400 running OSF/1.

Table 2.2: *Server, Request and Data Volume Returned for each Dataset.*

Dataset	Server	Request	Bytes returned
fi	www.sti.fi	image file	8205
se	www.kajen.malmo.se	image file	18333
uk	www.cl.cam.ac.uk	HTML file	2970
local	www.cs.washington.edu	data file	16000
ncsa	www.ncsa.uiuc.edu	image file	22719
netscape	home.netscape.com	image file	23301
archive	wuarchive.wustl.edu	text file	31520
sec	www.town.hall.org	keyword search	19280
webcrawler	www.webcrawler.com	keyword search	54586
yahoo	search.yahoo.com	keyword search	35645

We turn now to the statistics we apply here. Both the *mean* and the *median* are useful measures of "typical" response time. Where the distribution of response time has a long tail, the median better describes what is "typical." On the other hand, it inadequately represents the impact of occasional lags in service time on user frustration. Such occasional lagging is detectable when the mean is significantly higher than the median. However, there are better ways to describe variation. The most commonly used measure is the *standard deviation* about the mean. Instead of using this measure, we use the *coefficient of variation (CV)*, because it lends itself to comparison across multiple batches and datasets of different means. The CV is defined as the standard deviation divided by the mean, and is expressed as a percentage; e.g., a 95% CV implies the standard deviation is nearly as large as the mean, indicating great variability. Another statistic that is useful for characterizing

variability is the proportion of service times that exceed the median by some factor, e.g., the proportion of service times that are greater than twice the median.

2.2.2 Presentation of Results

Figures 2.3a–d show the raw timing measurements for four of the datasets. The vertical time scales of these graphs differ by more than an order of magnitude. Notice the service slowdown during the daytime, especially on weekdays.

One can detect a multi-modal distribution of $T1$, the connection establishment time, by the multiple horizontal lines in Figure 2.3a, b, and d. The double row we see in the `webcrawler` dataset is due to caching effects. In datasets `webcrawler` and `ncsa`, the first connection usually took about twice as long to establish as the following nineteen; 40% longer in datasets `netscape` and `yahoo`; and less than 14% longer in the other datasets, respectively.

The multiple bands of $T1$ values seen in Figures 2.3b and d are the effect of timeouts on failed domain name server lookups.[6] In fact, nearly every non-local dataset exhibits these timeouts at about 6 seconds and 30 seconds. Figure 2.4 shows a histogram of all $T1$ measurements of all datasets combined. Observe the two humps. The `uk` dataset alone exhibits an additional hump at 12 seconds.

For each batch, we compute the CV of its twenty probes. Figure 2.5 shows how the CV of the Ta measurements for dataset `yahoo` varies over the week. Corresponding with the service lag experienced during weekdays and daytimes, as shown in Figure 2.3, we see an increase in variation during those same times; recall that by the definition of the CV, this means the standard deviation is growing more than is the mean at these times. In the following analysis, keep in mind that if we were to include only those batches taken when most people use the Web, the computed

[6] The $T1$ measurement of each probe includes the time to resolve the host address with a call to `gethostbyname()`. Somewhat surprisingly, timeout delays were not significantly more likely for the first probe of each batch, but were equally distributed among the twenty probes.

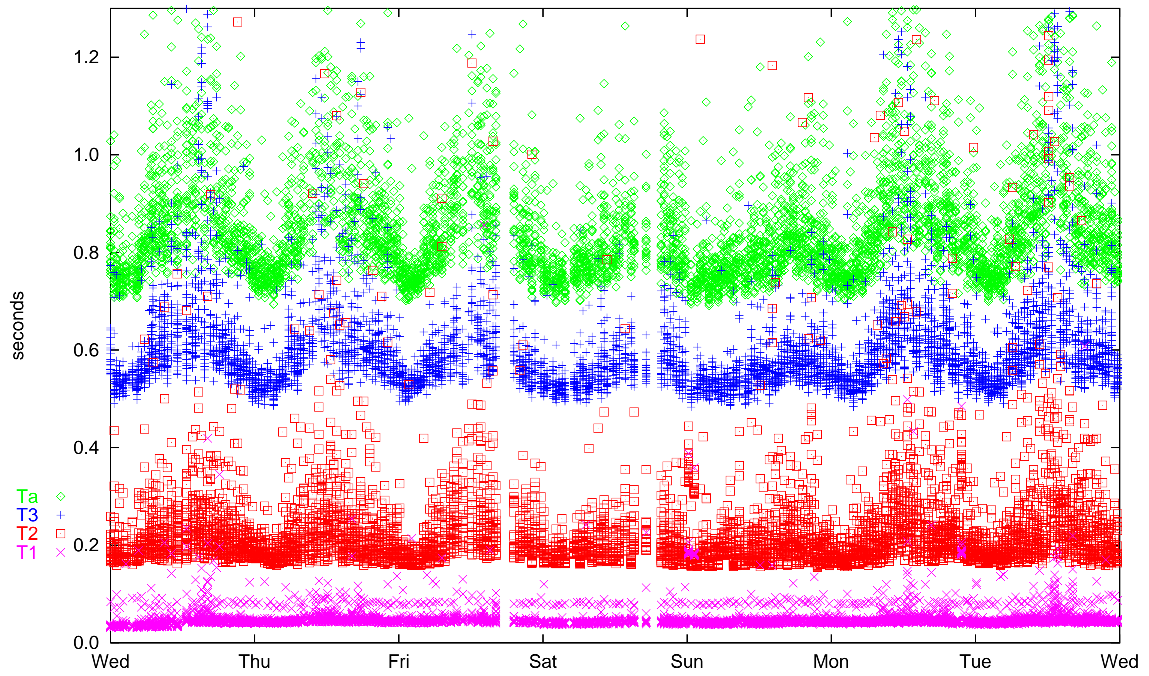

Figure 2.3a: *Raw Timing Measurements of the Web; Dataset* `webcrawler`.

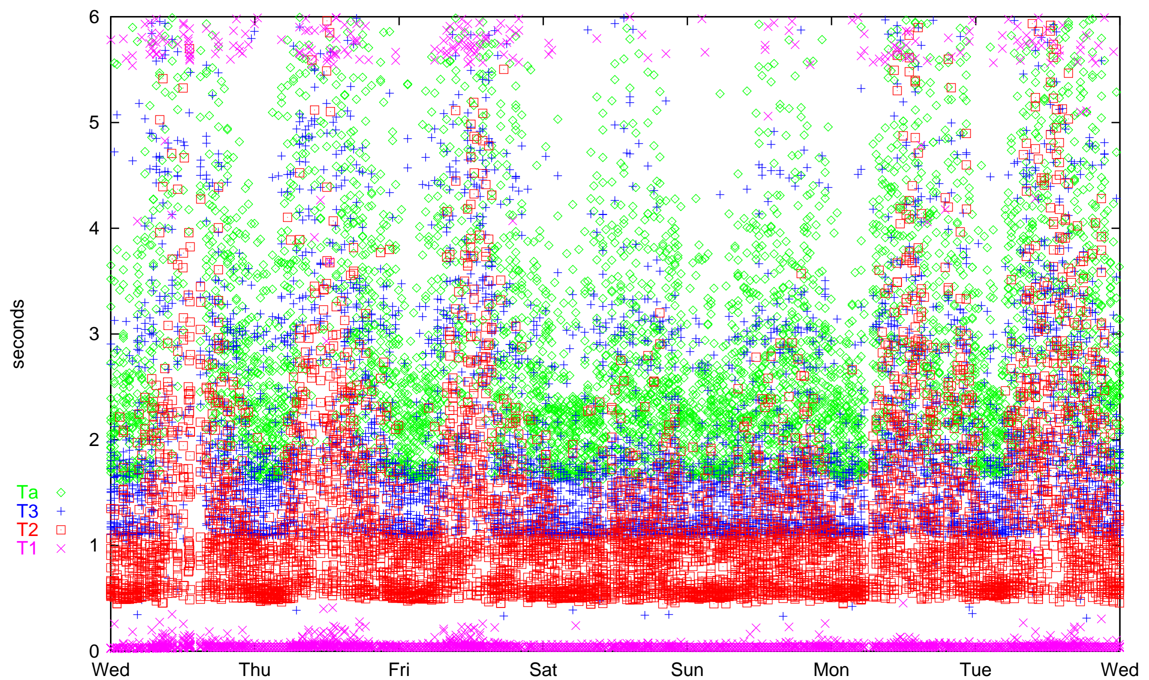

Figure 2.3b: *Dataset* `yahoo`.

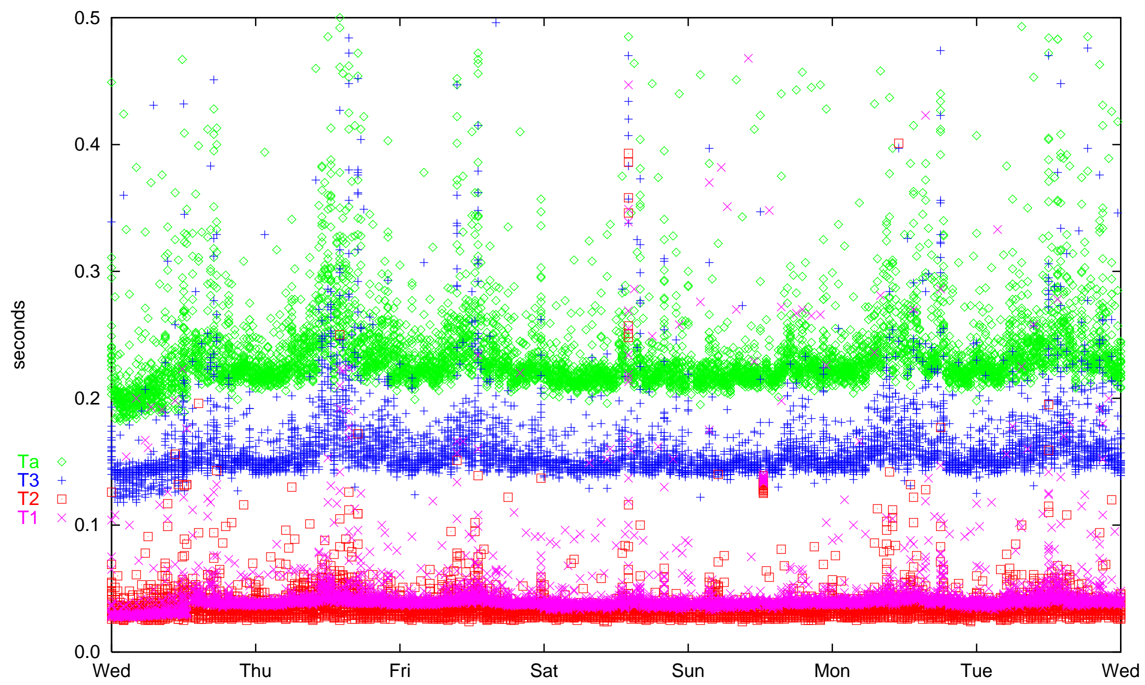

Figure 2.3c: *Dataset* `netscape`.

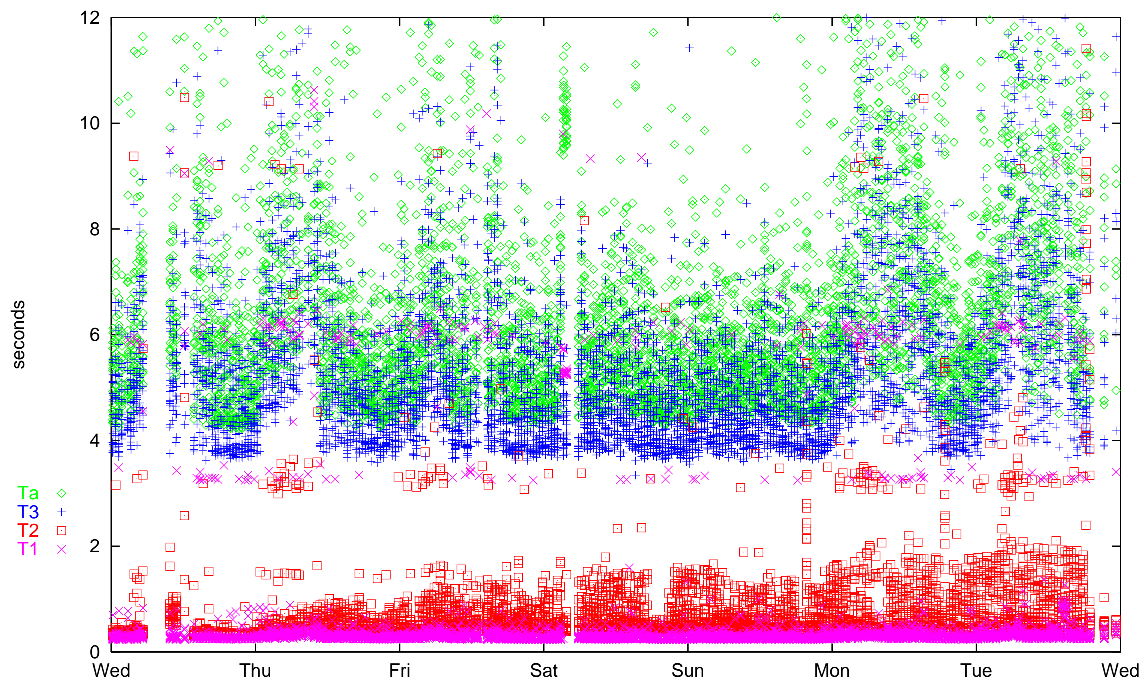

Figure 2.3d: *Dataset* `se`.

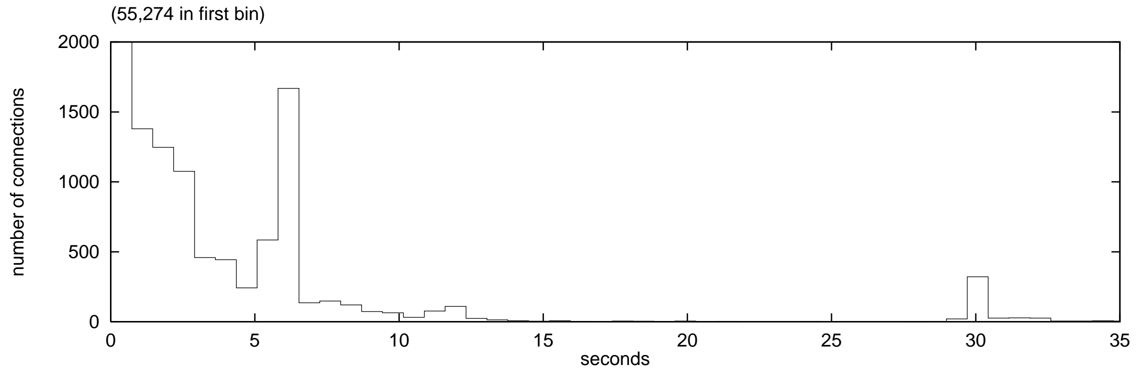

Figure 2.4: *Histogram of Connection Establishment Times ($T1$) of All Datasets.*

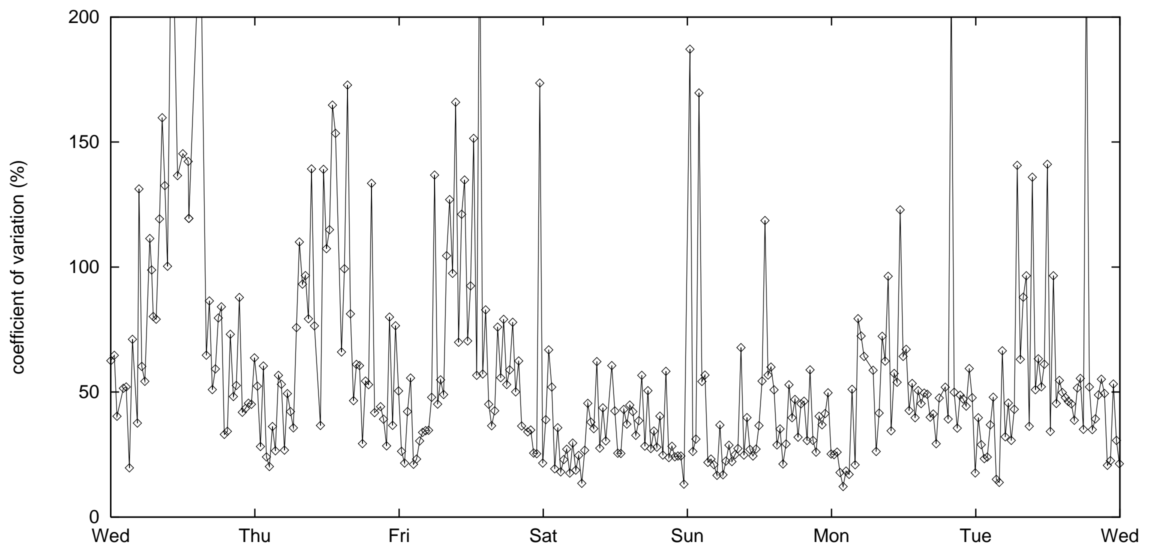

Figure 2.5: *Coefficient of Variation of Ta for Batches from Dataset* `yahoo`.

degree of variation would be greater because of the diurnal and weekday increase in variation.

To summarize over time, we average the CV values of the batches. Figure 2.6 presents this summary for each timing measurement type and dataset. In addition, because the CV normalizes for differences in the mean, we can summarize across the batches of all datasets combined. This is shown in the column labeled `altogether`.

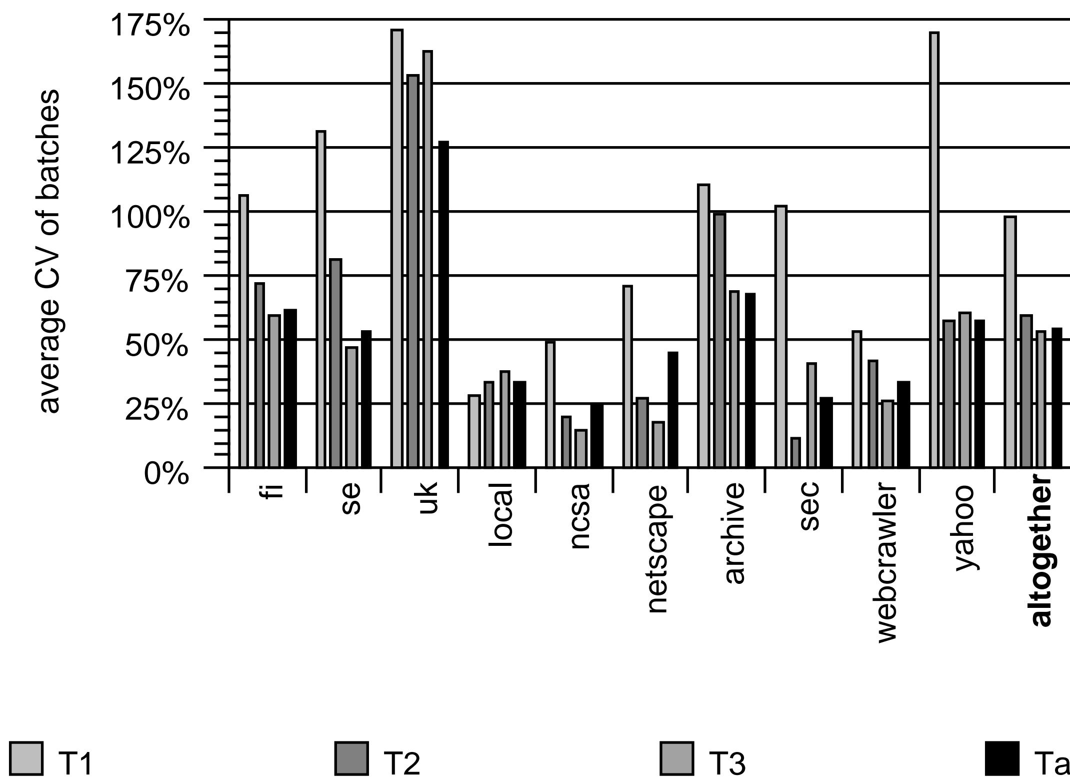

Figure 2.6: *Coefficient of Variation of Batches Averaged Across Time.*

Overall, the average CV of $T2$, $T3$ and Ta is over 50%, while the average CV of $T1$ is nearly 100%. In almost every dataset, the average CV of $T1$ significantly exceeds the other measures. Although it may vary to the greatest degree, it is not necessarily the largest component of service time variation (measured in seconds). For evaluating the relative impact of variation in the different types of measurements, refer to Figure 2.7, which shows the standard deviation of batches averaged across time for each type of measurement and dataset. (The vertical axis is log scale in order to present them together in one graph.) With this view, we see that, for most datasets, the major

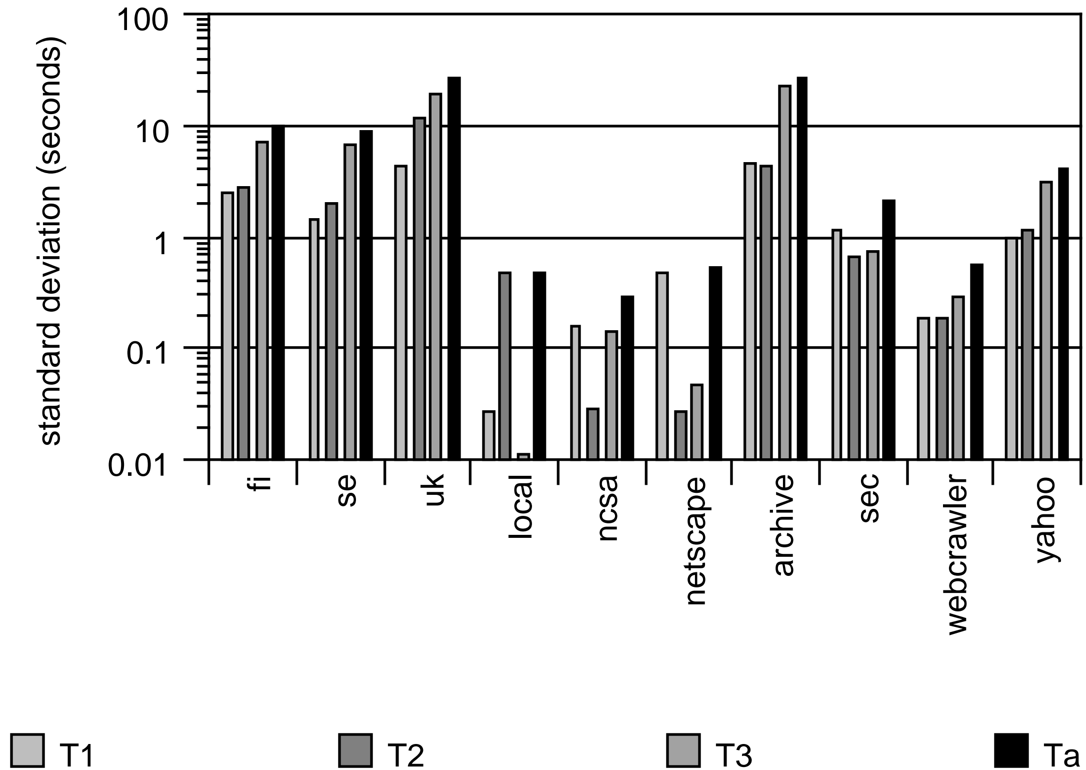

Figure 2.7: *Standard Deviation of Batches Averaged Across Time.*

factors of variation are the initial response ($T2$) and the reply bandwidth ($T3$). Notice the latter shows very little variation in the local case. Only for datasets `netscape` and `sec` does the variation in connection establishment time ($T1$) dwarf the other factors. The datasets for which the overall standard deviation of Ta exceeds one second correspond with the servers that sometimes feel sluggish when browsing the Web.

Another way to characterize variation is by the proportion of requests that significantly exceed the typical response time. For this we divide the timing

measurement Ta of each probe by the median Ta value of its batch, and produce a cumulative distribution function for each dataset, shown in Figure 2.8. To determine

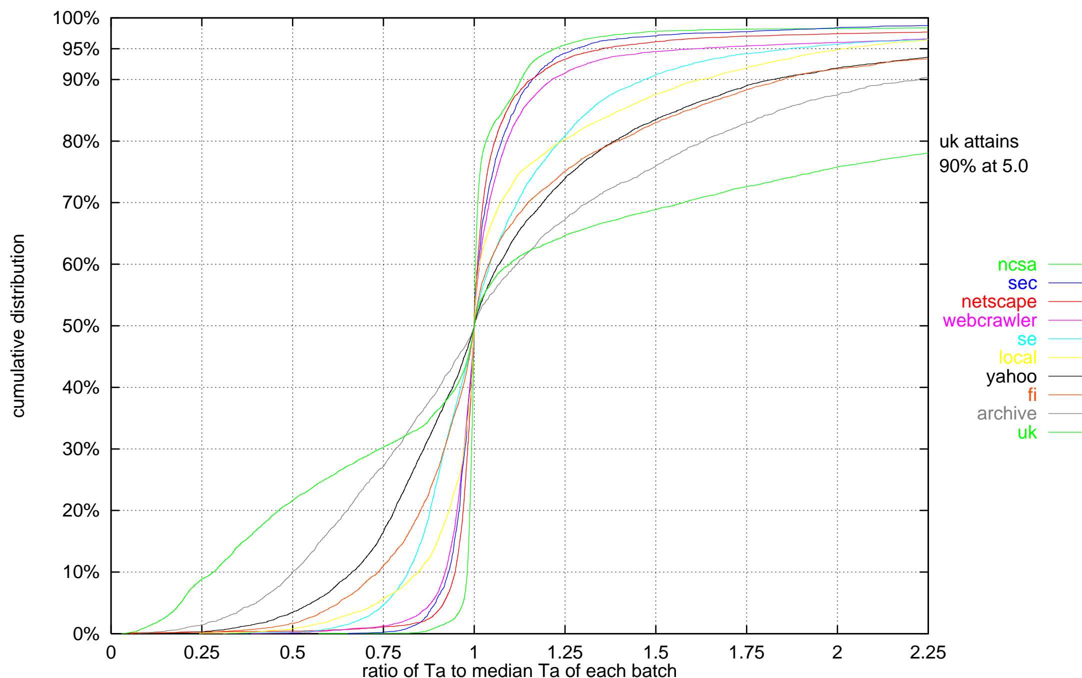

Figure 2.8: *CDF of Ta Divided by Median Ta of its Batch.*

the proportion of probes that exceed the typical response by, say, 75% or more, follow the 1.75 abscissa upwards to the curve for each dataset; the region above the curve represents the proportion. Doing so, we see, for example, that for datasets `yahoo` and `fi` approximately 11% of the probes exceeded the typical response by 75%, and that 20% to 35% of the probes surpassed their medians by 25% or more for six of the datasets. Alternately, one can slice the curves horizontally. For example, following the 95% level across, we see that 5% of the probes exceeded the typical response by 50% or more for all but three of the datasets, and for half the datasets, 5% of the probes took more than twice the typical response time.

Section Summary

In this section, we have identified volatility— the *short term* variation in service time— as an important aspect of resource variability (whereas the previous section discussed the range of variation independent of time scale). We have presented the results of a study to measure the volatility experienced in one of today's resource-variable environments: the World Wide Web. We characterized volatility by several different measures, including the average coefficient of variation of the batches across time, and the proportion that exceed by some factor the then-current typical response time. We observed that for all but the `local` dataset the coefficient of variation averaged over batches identifies the connection set-up time $T1$ as the most widely fluctuating, however, in absolute terms, the standard deviation of the data transfer time $T3$ was typically the largest component of the total response time variation. Finally, we found that for Ta in six of the ten datasets over 20% of the individual probes exceeded their respective batch medians by 25% or more.

Chapter 3

EXISTING APPROACHES TO OBTAINING RESPONSIVENESS

Given that an application should operate with good user responsiveness even in environments with dynamically variable service delivery, what can be done? In this chapter we briefly discuss existing approaches to address this question. We first describe how traditional practice and its obvious follow-on fail in dynamic, resource-variable environments. In the next section we list what general techniques can be used to improve responsiveness when resources lag. In the last part of this chapter we look at existing facilities for implementing these techniques.

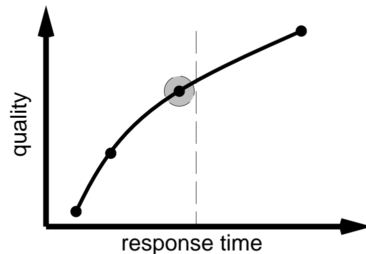

Figure 3.1: *Static Sizing: Qualitative Graph of Quality vs. Response Time.*

In choosing what response to generate for a given manipulation of the user interface, there is often a tradeoff between response time and quality, as illustrated abstractly by the curve in Figure 3.1. The vertical axis represents response quality, or level of functionality provided to the user, e.g., image resolution. The horizontal axis represents elapsed time from user input to completed response. The curve represents the choices available to the programmer in responding to a given user

input; it takes more time to generate a higher quality response. Ideally, programmers have a continuous range of choice, but in practice the choice is often discrete, e.g., in responding to a scrollbar being dragged by the mouse, one might re-display the document in each new scrolled position, or show the scrolled document only when the mouse button is released. The vertical dotted line suggests a perceptual deadline to be met; for the scrolling example, one might be willing to wait up to a quarter second, but not much more.

Static Sizing

Establishing responsive application behavior is straightforward in traditional environments where resources are approximately known in advance and stable during execution. Programmers simply select the best quality response that provides satisfactory response time based on the expected performance of the platform on which the application will run. This *static sizing* method works well traditionally, but is rendered ineffective at ensuring good user responsiveness for environments where runtime resources can be widely variable or are not approximately known in advance [34]. Statically sized applications suffer inadequate responsiveness during periods of low resource availability, and ignore the potential for higher quality responses during periods of resource abundance. Poor responsiveness being the greater problem of the two, programmers sometimes base their sizing on a lowest common denominator platform, sacrificing functionality and higher quality in favor of responsiveness. Even this technique does not ensure responsive behavior if the environment can exhibit delays beyond what was anticipated.

Dynamic Sizing

The obvious follow-on is *dynamic sizing*, where the programmer writes the code necessary for a range of quality levels and also writes code to intelligently select the

appropriate quality level based on resource availability at the time of the particular operation [34, 66]. One could write a different procedure for each quality level [49], or write a single procedure that scales the data size or error tolerance to adjust for different quality levels [35]. The latter is accommodated, for example, by the Kodak photograph format for compact disks, which stores each picture at five different quality levels ranging from 128×192 pixels to 2048×3072 pixels [18].

In theory, this technique ensures an appropriate balance between response time and response quality. In practice, however, it may not meet its goal, and involves significant programming complexity. Notice that it requires writing a new variety of code that forecasts the highest quality level that will meet a fixed response time. In the real world, it can be difficult to obtain an accurate prediction; there may be uncertainty both in the current resources and in the workload that will be generated at a given quality level. Also, for some resources, such as an unfamiliar Web server, predicting service time may require test and measurement, involving additional load on those resources and delay in getting to the real work to be done.

As the level of volatility in service delivery increases, advance measurements cease to reflect the actual conditions experienced when an operation finally executes. Response time suffers if resources drop after the measurement and before the operation completes. This calls for additional code that monitors resources during execution and cancels operations that will take too long and restarts their execution at a lower quality level. This carries additional overhead, and introduces the potential for thrashing.

For highly dynamic, resource-variable environments, we do not envision dynamic sizing to be a suitable use of programmer effort given the existence of other techniques, such as those described in the following section. Nevertheless, dynamic sizing has its place in writing code for environments where resources are stable but cannot be estimated at programming time.

3.1 Approaches to Obtaining Responsiveness Despite Volatility

In this section we highlight existing techniques that can improve user responsiveness when request completion time is not approximately known in advance and potentially much slower than desirable. Here we focus on methods that help cope with variable service time from system resources, as opposed to methods that try to control service time, such as dynamic sizing [34, 35, 66] and reservations [8, 5, 30, 97].

For any specific application, some of the individual techniques described below may not be suitable, but overall they cover a wide variety of domains and can have a large payoff. Hence, application support for including these techniques in one's program would be useful.

3.1.1 Feedback Before Completion— Incremental Quality

Users want responses immediately. At the very least, they demand some minimal feedback to their input [88], else they may end up entering extraneous actions, such as repeatedly clicking a button, until they see a response, sometimes with undesirable consequences. If the complete response to a user action is generated instantly, great. If not, a temporary response must be provided. Progress messages embody this technique in a limited way.

A more sophisticated approach is to produce quickly a low quality response at first, and incrementally improve its quality over time, as resources allow [19]. For example, an application that dynamically downloads city maps could display the major streets first, and later fill in the other streets and improve street curve resolution. By riding along the quality vs. response time curve in this fashion, this powerful technique delivers both quick responsiveness and the highest quality users care to wait for (although at some delay over what would be needed to provide only the high quality response). Users simply proceed as soon as the response quality suffices for their purposes. The interface may also allow the user to influence dynamically which

aspects of the response are improved in quality early on, such as in *computational steering* for scientific visualization [27].

This brings up a comparative shortcoming of the dynamic sizing technique: if the user wants a higher quality response than the one generated, some extra functionality is needed, whereas with incremental quality, the user gets higher quality just by waiting. Also, unlike dynamic sizing, under this technique the programmer need not write any code to predict an appropriate quality level to meet a fixed response time.

The incremental quality technique works well in domains for which the effort at each quality level contributes directly toward higher quality levels. Where this "inclusion" property does not hold, the overhead of duplicated processing at each quality level weighs against the improvement in response time. Fortunately, there has been a great deal of research in cultivating *multi-resolution* representations with this inclusion property in a broad variety of domains [19]: images [93], object graphics [32], three-dimensional models [26, 35], movies [31], sound, etc. The general notion of incremental processing and display can be applied to text documents as well, e.g., Netscape formats and displays HTML documents incrementally as they arrive.

3.1.2 Asynchronous Operation

Traditionally, sequential programming languages provide a "request-response loop" model[1] for human interaction with computers, where each response completes before the next request is processed. This model becomes unsatisfactory in variable-resource environments, however, when delays in one response slow or suspend further human interaction.

It is useful to allow the processing of user requests to proceed asynchronously from one another, especially so that new user input can be processed even when a previous

[1] It is known as the "read-eval-print loop" in the LISP community, and as the "event loop" for graphical user interfaces.

request is delayed, say, by a temporary network disconnection. Asynchronous operation also allows users to tap concurrent processing capabilities of the environment. High latency operations to remote resources can be overlapped and different resources can be used in parallel.

Netscape employs this technique to a degree: for example, when a user follows a hyperlink that takes a long time, he or she can open a new window that operates independently of the first in order to browse another part of the Web.

Asynchrony is beneficial not just at the granularity of whole user requests, but also in executing the sub-tasks of an individual request. Netscape fetches multiple inlined images of a Web page at the same time, for example; the display of each image is thereby not held up by the others.

Another technique for asynchronously decoupling the response time to the user from the response time of the system is to pre-fetch or pre-compute speculatively information that will likely be needed. For example, work by Tait *et al.* [90] attempts to predict future file access patterns so that files can be downloaded in advance of their potential demand.

3.1.3 Resource Allocation

With asynchronous operation, we have multiple tasks executing concurrently, which allows the possibility of intelligent resource allocation among them. If some tasks are more important to user responsiveness than others, they should be given priority. For example, demand-fetching of a document might be given priority use of a wireless network over spooling to the printer.

Dynamic interaction with a user affords an application special opportunities for directing resources to enhance perceived responsiveness. By focusing resources on those tasks that currently occupy the user's attention, we can improve the responsiveness of tasks the user cares about, trading off the responsiveness of peripheral tasks. Resource allocation can be especially beneficial where results are

produced in many different panels of a graphical user interface, being that users focus on only a fraction of them at a time. Cues for determining the user's attention may include mouse location, keyboard focus, window manipulation and visibility, and application-dependent knowledge, such as user commands that indicate focus on particular window panes.

3.1.4 Cancellation

In an environment where asynchronous operations may take a long time to finish, newer user actions can cause outstanding tasks to become obsolete. By detecting these situations and terminating obsolete tasks early, we can conserve system resources, improving responsiveness for the tasks the user cares about. For example, suppose an intelligent agent [24] is searching the Web for particular information and displays a summary incrementally as results arrive. We can reduce contention for computation and network service if we call off the search as soon as, say, the user has seen enough results and decides to search for something else. This optimization opportunity presents itself because of the combination of (1) potentially slow responses to user actions, (2) the ability to respond to new input asynchronously from outstanding operations, and sometimes, (3) interaction with users, who change their minds about what to do. The longer the delays of the resource-variable environment, the more likely it is that outstanding operations may be superseded as users move ahead or change their minds.

Cancellation could almost be viewed as a special case of resource allocation where the canceled task has infinitely low priority to suspend it from execution, preventing it from consuming more resources. However, a suspended task may still be holding resources, such as memory and network buffer space. Thus, resource allocation alone does not obviate the need for cancellation.

3.2 Existing Support

Having cataloged several important techniques for providing good responsiveness, we turn now to the rudimentary level of support afforded programmers by existing facilities.

There is no particular support for applying the incremental quality technique to application features, besides the existence of a few standards for multi-resolution formats [16].

Asynchronous operation is supported by three different facilities: event callbacks (as in user interface widgets [108, 109]), polling (as in the Smalltalk user interface [107]), and threads (as in Microsoft Windows NT [20] and multi-threaded X Windows implementations [79]). Event callbacks, as provided by most windowing systems that manage the event loop, let the programmer designate a procedure to be run when a particular event occurs. For example, instead of consuming an input file with a simple loop (which could virtually hang application responsiveness if file access were slow), the programmer specifies the body of the loop as a procedure to execute whenever data becomes available for the file; after opening the file and specifying the callback, control is returned to the windowing system. Even if the file is delivered slowly, the application continues to be responsive to new user actions. Programmers must be vigilant to code every potentially slow I/O interaction with a callback. This incurs a cost in programming effort, since programming with callbacks is more cumbersome than the straightforward sequential programming style [63].

Callbacks help cover I/O delays and untangle interleaved user dialog actions, but they are not a panacea. Incoming events are noticed in the main event loop, but *during* the execution of a callback routine the application is unresponsive to new input. Hence, for asynchronous operation of the interface during CPU intensive tasks, callbacks alone are not adequate.

In such a case, the traditional implementation technique is to sprinkle the event

processing routine with polling calls[2] that temporarily dive back to the event loop to process any accumulated events. The principle problems with this approach are that it clutters the code, and the polling calls need to be placed appropriately. If polling is too infrequent, users experience unresponsive behavior; if too frequent, polling overhead slows the application. For a widely variable environment, one cannot determine statically where in the code to poll to manage this tradeoff well.

This brings us to threads, which overcome the problems mentioned above for event callbacks and polling, and allow cleaner code structuring than either [59, 87]. Threads provide an appropriate vehicle for overcoming both computation and I/O delays[3], and are the recommended facility for implementing asynchronous operation on modern, thread-capable operating systems [47].

Threads are a powerful, but primitive, facility. The interface typically consists of calls for forking threads, joining with threads, changing thread priorities and perhaps their scheduling policies, and possibly, terminating threads cleanly via cancel signals/exceptions. The facilities for adjusting thread priority and canceling threads provide a basis for implementing resource allocation and task cancellation, respectively.

Multi-threaded programming, despite its advantages, has been difficult historically [17, 68]. The extra burden of managing threads and data synchronization adds substantially to the programming complexity needed for the basic application.

---

[2] One polls with `PeekMessage` in Microsoft Windows, `update` in Tcl/Tk, `XmUpdateDisplay` in Motif, and `XtAppPending` in raw X Windows, for example.

[3] *Kernel* threads, as opposed to *user-level* threads, allow the threads of an application to continue to run even when one of them blocks on a system call. User-level threads with *scheduler activations* [6] effectively provide the non-blocking benefits of kernel threads; for our purposes, we do not distinguish between the two.

3.3 Summary of the Motivation

In this chapter we outlined important techniques for attaining responsive behavior when service from the environment may be highly variable or not accurately estimable. Although they are known techniques, they are seldom encountered in today's applications, because (1) their programming cost is large, and (2) computing environments have typically been stable enough for the static sizing method to deliver applications with satisfactory responsiveness. But the trend toward greater variability in the environment is rapidly increasing the pressure on programmers to deploy these "responsiveness enhancing" features. The shift from Mosaic to Netscape is not arbitrary: Web browsers are one of the first highly interactive applications to capitalize on the Internet, a much less stable environment than traditional stand-alone or local-area networked computing.

This highlights the former issue above, programming cost, which is incurred for each feature of each application to which these techniques are applied. The support provided by existing facilities is, in a nutshell, negligible compared to what is offered by the framework introduced in the following chapter. The benefits of this research multiply, as the framework is reusable across a broad range of application domains.

Chapter 4

THE PETRA-FLOW FRAMEWORK

In this chapter we introduce a novel framework, which we call *Petra-Flow*, to support programmers in constructing and executing applications that have good responsiveness in modern computing environments exhibiting highly variable service. We begin by presenting the key ideas independently of any particular embodiment, and in the second section give an overview of the fundamental programming components. Section 4.3 explains the practical use of Petra-Flow in an extensive example, applying it to a hypothetical slide browser program. The subsequent section furnishes particulars about the services Petra-Flow provides. Finally, the last section describes in detail one of the fundamental components of Petra-Flow, a new kind of software lock devised to improve responsiveness for important tasks.

A prototype implementation of the Petra-Flow framework is presented in the next chapter.

4.1 Key Ideas

Imagine that somehow we are able to split the execution of an interactive application into two conceptual components: one determines what needs to be done (the user interface, primarily) and hands work requests over to the second component for processing. With this asynchronous decoupling of the user interface responses from work request completions, the user interface can better keep up with the user, even if the execution component falls behind in delivering its final (full-quality) results. (We assume that during such a resource shortage the response time of

keeping the user interface current without full application-specific content is less than that of displaying it with full content. This holds in many domains where the service bottleneck is remote or the processing of user objects is large relative to the manipulation of graphical user interface widgets.)

The backlog of specified-but-uncompleted work presents an opportunity for runtime optimization to make the execution component more responsive to the user, e.g., determining whether portions of the outstanding work will go unused, and eliminating them before they are executed.

As a means for optimization, we use runtime analysis of the data dependencies among the pieces of outstanding work, or *tasks*. A task is essentially a procedure call that executes asynchronously; the declaration of a task is like that of a procedure, but it also specifies input and output relationships to parameters and global variables that it reads or writes. At runtime we maintain a global dependence graph of the outstanding tasks, significant program variables and their values. The dependency arcs reflect the input-output relationships. While there are many details to be explained later, the key notion is that by maintaining this graph, we have a *global view* of the dependencies among outstanding tasks. Shortly we will give a detailed example and explanation of the global dependence graph, but first we motivate its construction by showing that it gives us a great deal of leverage for runtime optimizations:

1. We can expose concurrency by allowing those tasks that do not violate any dependencies in the graph to run in parallel. When a task completes, we can start each of its dependent tasks, unless they have other unsatisfied dependencies.

2. We can use the dependence graph to apply runtime analogs of compiler optimizations:

   (a) An analog of *variable splitting* or *variable renaming* [70, 45] eliminates

anti-dependencies (write-after-read) and output dependencies (write-after-write) in the graph, enabling out-of-order execution and thereby greater concurrency.

(b) An analog of *dead code elimination* [4] conserves resource consumption by the early termination of tasks whose results no longer affect the final computation. A branch of the dependence graph is *obsolete*, and can be pruned, if no program output or variable is dependent on it, such as happens when a user action overrides prior actions. Whereas traditional dead code elimination occurs at compile time and eliminates lines of code, this optimization occurs at runtime and eliminates task invocations. Pruning tasks that are currently executing is delicate but feasible.

3. With a simple indication of which program variables are most important to obtain results for quickly (e.g., where user attention is focused), we can employ the dependence graph to determine which tasks they depend on and grant those tasks priority for (ideally, all) resources. This serves as a basis for user-centric global resource allocation. The development of a data-oriented model for specifying time sensitivity (priority) is a natural transition from a focus on processing to a focus on the products of that processing [15]. Were priorities assigned to tasks instead of variables, as is common with most threads packages, the dependence graph could be used in the same way to propagate that priority.

4. The dependence graph also supplies the underpinnings for the support of incremental quality, a powerful technique for improving responsiveness, as discussed in the previous chapter. Whereas in traditional programming a function produces a single result and returns, we permit tasks to submit a low quality result quickly and continue to work on higher quality results. Meanwhile, the low quality result propagates down along the arcs of the

dependence graph, executing the dependent tasks it feeds. For example, imagine a low resolution version of an image quickly traversing down the graph of outstanding tasks, while a full resolution version continues to be downloaded. In this way, the user gets a low quality result processed through the backlog of work swiftly and a high quality result eventually. Referring back to our conceptual split of the application, this helps the execution component keep up with the user, although at a lower quality of result.

5. Finally, by deriving a global view of the tasks and their dependencies at runtime, we are in a good position to provide programmers with debugging information.

The Petra-Flow framework enables the user interface to operate asynchronously from work completion. With traditional programs that process requests synchronously, users are held up by service delays. The common technique of having user input buffered by the system, known as *type-ahead* or *mouse-ahead*, can help to a limited degree, but because it involves no interpretation of the buffered input (other than perhaps detecting signals to flush the buffer or interrupt the process), it is inferior to asynchronous operation of the user interface, especially with incremental quality results, resource allocation and task pruning. For example, if a user enters several actions, some of which override earlier actions in the series, a traditional interface with buffered input would dutifully execute each in turn, whereas an asynchronous interface would automatically elide the obsolete ones. With buffered input, the user may choose to flush the buffer in order to purge the obsolete commands, but then the buffered commands that are not obsolete are deleted as well, and need to be re-entered. Further, users may have a hard time deciding whether to flush the buffer in these circumstances, because there is a race condition between the user flushing the buffer and the program consuming the obsolete commands.

Let us emphasize that these opportunities for optimization arise from the potential for users to outpace resources. By exposing concurrency, prioritizing

tasks, eliminating obsolete branches, and presenting incremental quality results, users should be able to work faster when resources lag. This may allow users to outpace resources even more, in turn, yielding further opportunity for optimization.

Notes About the Dependence Graph

Note that the dependence graph is shaped at runtime by the course of user events, as opposed to those programming models where the programmer develops a *static* dependence graph when writing an application, such as AVS [95], IBM Visualization Data Explorer [1], IRIX Explorer [86], and Khoros [74]. The size of the graph is determined by how much the user gets ahead of resources. Hence, it depends on the speed of the user, the workload he or she generates, and the system resources available to accomplish it. The graph is naturally acyclic, because it represents the unfolding of computation over time as driven by the user; a work request in the past will never depend on an as-yet-uninstantiated work request.

The granularity of the graph is constrained by runtime overhead. One can conceive of statement-level granularity for a high-level interpreted script language, such as Mathematica or shell scripts, but larger granularity is needed for performant languages, such as C. While it might someday be possible for a compiler to select an appropriate granularity for such languages without programmer intervention, in this thesis we expect the programmer to specify the granularity manually.

4.2 Central Components

Here we describe the central components of the Petra-Flow framework, which makes practical the ideas set forth above. The components are: asynchronous tasks, versioned variables, and priority-mediated resources.

- An *asynchronous task* describes an application-level unit of work that may be executed asynchronously. It specifies its dependencies in terms of parameters

and any global variables it reads or writes. When the task is invoked, these dependency arcs are instantiated in the global dependence graph. For the support of incremental quality, a task may over time submit multiple results of improving quality for its outputs, unlike a traditional procedure call, which returns only a single result.

- A *versioned variable* is a potential attachment point for dependencies in the global dependence graph. This abstraction provides two levels of versioning: *values* and *quality versions*. The former supports variable splitting for different assigned values. The latter supports the incremental quality versions generated for a single assigned value. Every program variable could potentially be versioned, but for overhead. In Petra-Flow, the programmer manually specifies this attribute for appropriate program variables so that it is only applied where useful.

- Asynchronous operation is not permitted with many common facilities, for example, software libraries that were not written to be thread-safe. For systems and resources that must be used under mutual exclusion, one traditionally uses a simple lock. In practice it turns out this defeats the resource allocation mechanism, and so we provide the abstraction of *priority-mediated resources*: these allow re-use of existing code bases that are not thread-safe without sacrificing the global perspective gained on resource allocation. This component is discussed in detail in Section 4.5.

Together, the use of these components describe the data-flow and synchronization structure of a program, along with its exclusive resource needs.

4.3 Hypothetical Example: Slide Browser Application

We illustrate the Petra-Flow framework and its benefits by applying it to a hypothetical application. Consider a slide browser program that shows one slide of a sequence at a time, and has three operations at the user interface: (1) advance to the next slide, (2) print the current slide, and (3) mutate the current slide somehow, say, by histogram equalization of the image colors. Figure 4.1 shows pseudo-code for this application. In the initialization before the event loop, the first slide is loaded from `URL[1]` (suggesting remote storage of the slides on the Web) into the variable `C`, which stands for the current slide. From then on, the event loop processes user actions. When the user presses the mouse button, it loads `C` with the next slide. When the user types "`R`", the current slide is recolored and the result stored back into `C`. When the user types "`P`", the current slide is sent to the printer.

```
C = Load(URL[i=1])
event loop
  case next:    C = Load(URL[i++])
  case recolor: C = Recolor(C)
  case print:   Print(C)
end;
```

Figure 4.1: *Pseudo-Code for Hypothetical Slide Browser Application.*

With traditional programming methods, this application stops responding to the user if any of the actions takes a long time to execute, e.g., if the recolor function is invoked on a much slower machine than it was designed for, or if slides are delivered over a low bandwidth wireless network. This limitation can be overcome with asynchronous operation and incremental quality results. Using Petra-Flow, we mark each of the three procedures as asynchronous tasks, and indicate at the declaration

of C that it is to be a versioned variable.[1] The main application thread merely describes what needs to be done by asynchronous task invocations, rather than being responsible for actually accomplishing each task. (Asynchronous tasks may also be employed within individual tasks.)

We now examine a hypothetical execution of this hypothetical application built with Petra-Flow. Let us presume that the environment is currently resource-poor. The initialization step launches a task to load the first slide and store its value to the variable C. Suppose the user prints this slide and then wants to move on quickly to see the next slide, even though the load and print tasks take a while to complete. Figure 4.2 illustrates the dependence graph through each of these transitions. The initialization launches a load task (represented as an oval) that works at producing a value (represented as a rectangle) for the versioned variable C (represented as a diamond). (For now, think of a value as an allocated block of memory. Later we reveal another level of indirection for managing the actual memory allocations of incremental quality results.) When the user requests that the slide be printed, a print task is attached to the graph, reading its input from the value of C.

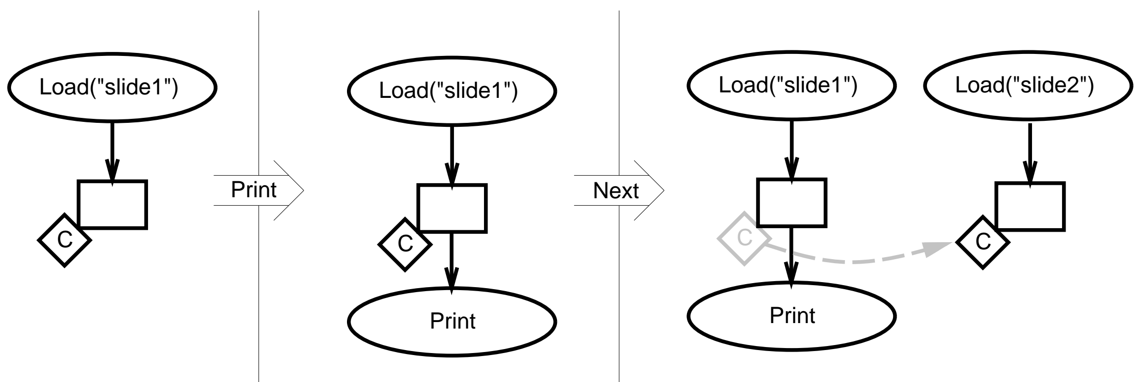

Figure 4.2: *Dependence Graph Transitions Showing Variable Splitting.*

[1] The particular notation for this is immaterial at this point. The syntax for the prototype implementation is given in Chapter 5.

When the user decides to advance to the next slide, the mouse event leads to the execution of the statement `C = Load(URL[i++])`, which conflicts with the value of `C` needed by the print task. Ordinarily this new task would need to wait until the print task completes and, say, releases a read lock on `C`, but with versioned variables as supported by Petra-Flow, a separate region of memory can be used for the new value of `C`, allowing the new load task to write to it without interfering with the existing load-print chain of tasks. The rightmost dependence graph shows the situation after executing the statement above. Note that the variable `C` is associated with a new memory allocation to which the new load task writes. Just as in traditional sequential programming, `C` refers hereafter to the new value, e.g., a newly launched print task would print this slide. As illustrated by this example, asynchrony and variable splitting in combination with incremental quality results allow the user interface keep up with the user despite slow completion of prior actions.

Resource Allocation

Now, suppose we want to augment this application to hide latency by pre-fetching the next slide while the viewer appreciates the current slide. The pre-fetch could start either after or concurrent with loading the current slide. The advantage of concurrent pre-fetching is that startup delays can be overlapped, and parallelism in the environment can be exploited, e.g., if slides reside on different servers. But we do not want the pre-fetch to interfere with the download of the current slide by contending for resources, so we need to communicate the relative priorities of these requests to the execution environment. If these (incremental) download requests are long lived, we will need to adjust their priorities dynamically as the user advances to this pre-fetched slide and beyond.

Petra-Flow supports this dynamic priority management atop raw operating system facilities for setting priority. We extend the above example with pre-fetching to illustrate this. Figure 4.3 shows the enhanced pseudo-code. Pre-fetching disturbs

```
C = Load(URL[i=1])
P = Load(URL[i=2])
C's priority = high
event loop
  case next:    C = P
                P = Load(URL[i++])
  case recolor: C = Recolor(C)
  case print:   Print(C)
end;
```

Figure 4.3: *Pseudo-Code for Enhanced Slide Browser. (Modifications for pre-fetching are italicized.)*

the code of the prior example little: in initialization we launch an extra load task that writes into P, the pre-fetched slide; and when advancing to the next slide, we assign[2] the current slide variable C to the value of the pre-fetched slide P, then begin pre-fetching the subsequent slide.

The third initialization statement exhibits a unique capability of Petra-Flow: it asserts that the current slide variable C is of great importance, and so anything it may depend on during the course of execution should be granted high priority. The global view afforded by the dependence graph puts the Petra-Flow framework in a position to endow the appropriate tasks with priority throughout execution automatically. Note that this is achieved here by writing a single line of code at initialization, rather than having to add statements to deal with priority throughout the event loop. This programming ease is facilitated by Petra-Flow's unique notion of assigning priority to *variables*, in contrast with threads packages, which assign priorities to threads.

We now demonstrate this mechanism via a hypothetical execution in which the user recolors and prints the first slide, then moves on to the next. Our purpose is to

[2] The assignment is by reference so that C inherits any ongoing load task for P. Note that C is unaffected when P is later re-assigned, because a new allocation is used rather than writing over the old value.

show how Petra-Flow automatically favors the tasks the user is waiting for.

Figure 4.4 shows how the dependence graph evolves throughout this execution. The initialization statements launch two load tasks— the second for pre-fetching slide two. The priority ascribed to `C` is conveyed up to the load task for the current slide, as illustrated by the bold lines. When the user recolors the slide, the variable `C` is

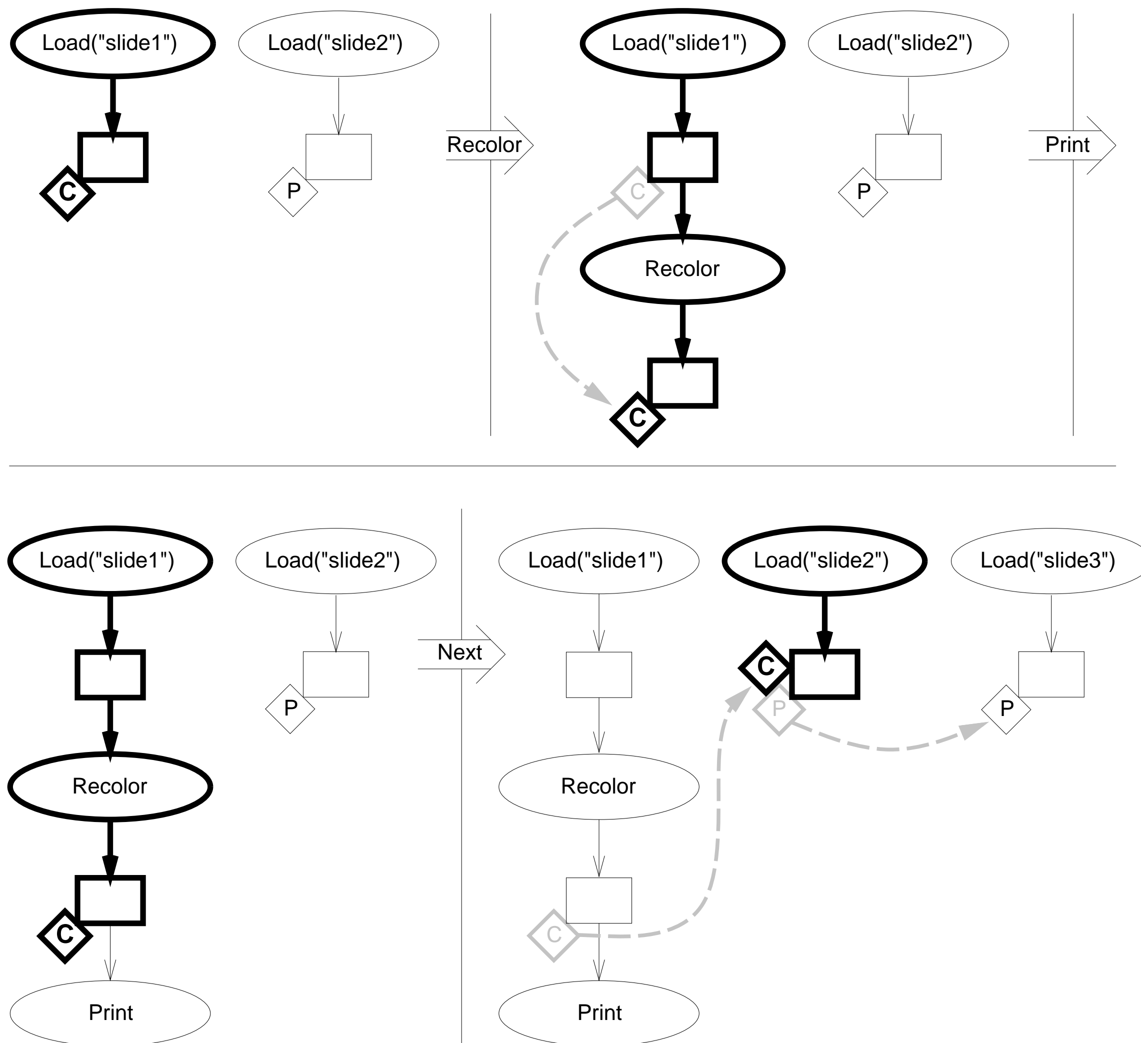

Figure 4.4: *Dependence Graph Transitions Showing Priority Up-Flow.*

split to follow the user through the statement `C = Recolor(C);` in the second graph

of the figure, the new recolor task reads from the old value of C and writes to a new value. The priority associated with the variable C flows up the dependence graph, granting priority to just those tasks involved with (incrementally) generating the current slide. The resource demands of the pre-fetch load task for network bandwidth and computation (say, for image decompression) are subordinate to those of the other two tasks. This helps to achieve better responsiveness for the results that most directly affect the user.

When the user prints the slide, the print task is attached to the current value of C. The print task is not granted priority because C does not depend on it.

When the user advances to the next slide, the variable C is reassigned to the value of P. Petra-Flow rescinds the priority boost given to the first load task and the recolor task, and then grants priority to the existing task that is pre-fetching the second slide. The last statement of the event handling code launches a new load task to pre-fetch the third slide, and reassigns the variable P to the newly allocated value.

In this way, the pre-fetching task and the load-recolor-print chain of tasks are implicitly run in the background at lower priority than the tasks involved with generating the current slide for the user. It is implicit because no mention of priority is needed in the event-handling code nor in the asynchronous tasks themselves.

Cancellation

In addition to managing priorities, Petra-Flow can conserve resources by pruning branches of the dependence graph that become obsolete. For example, had the user not printed[3] the first slide, when he or she moved on to the next slide there would be

[3] Some tasks, such as the print task in this example, must be specially marked to indicate they have important side-effects and therefore cannot be pruned even though they do not appear to write to any variable in the dependence graph. We will suppose that such tasks write to a (non-existent) "side-effect" variable to avoid having to develop and discuss special cases for side-effects.

no present or future consumer[4] for the recolored slide and only an obsolete consumer for the raw version of this slide. In this modified scenario, when the user advances the slide and the variable `C` is reassigned, Petra-Flow determines from an analysis of the dependence graph that the load-recolor branch is obsolete and prunes it by first canceling the tasks and then deallocating the anonymous values in the branch. This reduces contention for the network, the processor, and memory.

Summary

In review, this section demonstrated the application of the Petra-Flow framework to a hypothetical slide browser program. It showed how asynchronous tasks (producing incremental quality results) and versioned variables enable the state of the user interface to keep up with the user, despite slow completion of work orders when resources are scarce. It also showed how the runtime dependence graph can be used by Petra-Flow to automate resource allocation and obsolete branch pruning with little programming overhead. Together, these facilities ease the addition of program features for pre-fetching and backgrounded work while avoiding resource contention with primary tasks.

4.4 Uses of the Derived Global View

Again, one of the key ideas in this work is that by the use of a few fundamental components, we can derive at runtime a global view of the dependencies among application-level units of work. This structure can be put to a number of uses, principally resource allocation, management of multi-resolution results, and obsolete branch pruning. In this section we address some of the details of these services.

---

[4] There can be no future consumer attached to this version of `C` because it is no longer nameable.

4.4.1 Resource Allocation

Given information about which program results are time-sensitive from the perspective of end-user responsiveness, Petra-Flow can control resource allocation among the tasks to favor those on the critical path, i.e., those on which a time-sensitive result depends. Here we have chosen the notion of priorities to express relative importance. Whereas the slide browser example demonstrated the use of just high and low priorities, Petra-Flow can support more general representations, e.g., a numerical value, or a set of tokens, each with an associated priority.

Given a producer task that feeds multiple consumer tasks of different priorities, what priority should the producer inherit? Some straightforward choices are to take the sum or the maximum of the priority values, or if using the token-sets representation, the union. The summation option supports the notion that a task should be given higher priority if it is on multiple critical paths (refer to Figure 4.5a), however, it suffers a pitfall[5]: if there are multiple paths in the dependence graph from the producer task to a prioritized result value, the summation effectively multiplies the priority of the source by the number of paths (Figure 4.5b). By using the token-

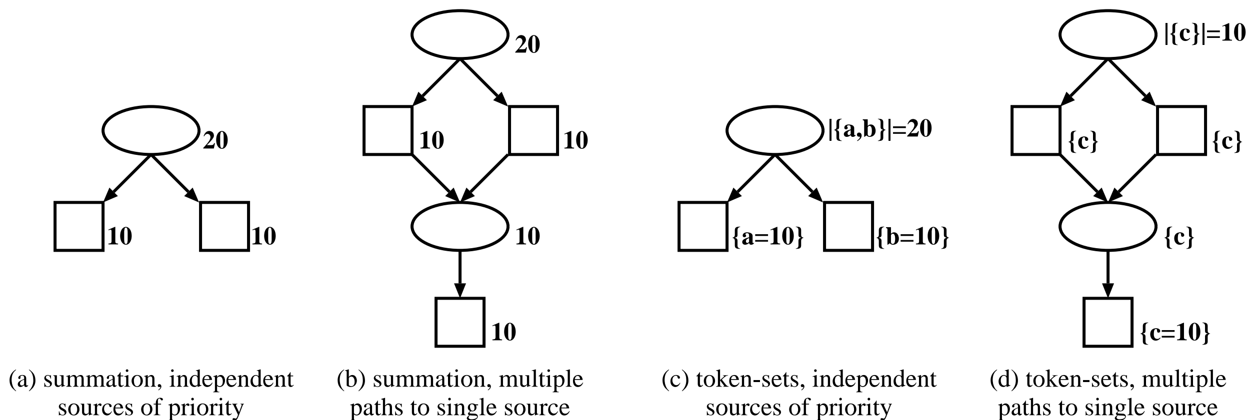

Figure 4.5: *Summing Priority Values vs. Unioning Priority Token-Sets.*

---

[5] This is related to the problem of *correlated evidence* in probabilistic reasoning[71, page 8].

set representation and set union to combine priorities, duplicate paths do not affect the priority of upstream producers (Figure 4.5d). In the end, we take either the sum or the maximum of the token values to convert a set to a numerical priority for the operating system interface. In the figure, we used the sum.

The operation of Petra-Flow is independent of the priority mechanism, and so the application programmer is allowed to control representation and policy choices. The operating system may also permit the programmer to select the effect of the priorities on the underlying resources, e.g., priority-weighted time-slicing vs. strict priority scheduling of the processor. The global dependence graph provides a framework for propagating priorities to the appropriate tasks; we are dependent on the operating system for the last step of translating priorities into improved service.

Resource allocation should be broader than processor scheduling alone. It should affect any critical resource, and naturally has the greatest effect at the system bottleneck. In mobile communications, the bandwidth bottleneck is the wireless hop between the mobile device and the base transceiver connected to the wired network infrastructure. This is good news, because it would be much easier for the network infrastructure to provide prioritized use of the local wireless cell than of Internet-wide networks. We look forward to widespread availability of prioritized service in the future— there is certainly much work on the related problem of providing *quality of service* support for isochronous streaming [8, 21, 30, 92, 97].

Finally, note that task priorities are updated dynamically under three kinds of events: (1) the structure of the graph changes by the addition or completion of a task, (2) a prioritized variable is (re-)assigned to a different value in the dependence graph, as demonstrated in the slide browser example, or (3) the priority of a variable is changed dynamically. The latter is associated with shifts in user attention, whereas the others suggest state changes in the application. The application is responsible for translating user input events, such as mouse motion and window repositioning, into priority shifts on program variables; Tcl/Tk's event binding mechanism makes

this easy to program. Petra-Flow then propagates these priority shifts to all affected tasks.

4.4.2 Incremental Quality Results

The global dependence graph is also used to support multi-resolution results. A task at the top of the graph can quickly produce a low quality result that passes down through the "data-flow" graph to the user; meanwhile, the task can continue to work toward higher quality. This is achieved via multi-threading. Given a few annotations on asynchronous task declarations, Petra-Flow generates for the programmer the necessary, low-level code to manage threads and data-flow synchronization, as well as the auxiliary code to maintain the dependence graph and debugging information.

As new quality versions arrive at the inputs of an intermediate task in the data-flow graph, Petra-Flow determines whether to execute the task on the new set of quality version inputs or whether to ignore them, according to the synchronization semantics chosen by the programmer. For instance, in the slide browser application, the print task should be run only on the final version of the slide image, whereas the recolor task should be run whenever a better quality version arrives (and if multiple versions arrive while the task is executing, we might like to skip to the best quality version available next time the task executes). For each graph-managed input to a task, the programmer selects under what conditions the task should be executed. Below we list some possibilities available to the programmer. (For the time being, consider only a single task input; later we will describe the execution semantics for tasks with multiple inputs.)

FINAL: A task input with these semantics executes only on the final quality version.

Petra-Flow can detect which quality version is the final one either by an indication from the producer task when it is submitted, or retroactively when

the producer task terminates, i.e., when it is removed from the dependence graph.

ANY: An input with these semantics runs on any available quality version. If there is a choice, it will choose the best available at the time. If the upstream producer has not yet generated any version whatsoever, it will execute once when the initial quality version is submitted. Once the task has executed, there is no need to run it again on higher quality versions.

These semantics are useful for reading meta-data about a value that does not improve with quality versions, e.g., the comment header fields of a multi-resolution image.

EVERY: An input with these semantics executes on every quality version in turn.

This choice of semantics is the only one that requires versioned values to retain the full a sequence of quality versions. If these semantics are not needed, only the best quality version for each value need be kept. This optimization may be used wholesale if these semantics are not used anywhere within a program, or with an analysis of the source program, may be applied on a per-variable basis where no present *or future* consumer will ask for these semantics.

SKIP: These semantics are like EVERY, except that intermediate quality versions are skipped if multiple versions are submitted during the execution of the task.

Skipping the backlog of quality versions helps the program keep up with the user despite slow resources.

PREEMPT: These semantics are like SKIP, but with preemption if a better quality version arrives while executing.

The implementation of these semantics takes advantage of Petra-Flow's cancellation facility. These semantics carry a risk of bounded starvation: if executions keep getting canceled by the arrival of better inputs, the user might not get a result for a long time.

PARALLEL: These semantics are like EVERY, except that a new thread is launched as soon as each new quality version is submitted, instead of waiting for the existing execution to complete. Multiple executions of a single task on different input quality versions could potentially run in parallel on a workstation with multiprocessing capability.

With these semantics, the framework needs to take care that the outputs of these tasks are managed appropriately. Should an execution submit its output before a prior, lower quality execution does, then either the higher quality result must be delayed until after the lower quality result is generated, or instead the lower quality result may be elided if no current or future downstream consumer uses the semantics choice EVERY or PARALLEL, in which case the lower quality execution may be terminated altogether unless it is producing some other output that is still needed. The latter is preferable to introducing artificial delay.

By providing execution semantics that *restart* a task's code on each set of new inputs, Petra-Flow can take over the management of threads and data-flow synchronization for the programmer. The alternative is to have the programmer write this tedious and error-prone code in a loop over input quality versions. The hardship of writing this code correctly and repeatedly for each task is significant, especially when one considers complex interactions across multiple inputs: For example, a task with four input parameters of semantics FINAL, EVERY, PREEMPT, and ANY, would wait until the final version becomes available for the first parameter, and run on every version of the second parameter, sometimes being preempted and re-starting with a

better quality version for the third parameter; the fourth parameter would receive the best quality version available each time the task is executed.

We now state precisely the execution semantics for tasks with multiple Petra-Flow inputs. The following rules are applied in order at any significant state change, i.e., a new task is installed in the graph, a new quality version is submitted for a task input, a quality version that was submitted earlier is marked as the final version (when its producer task terminates), or a task's thread completes or terminates early from a preemption request:

Precondition: The task cannot be run yet if, for any input, no quality version is available, or its semantics are FINAL and the best available quality version is not (yet) marked final.

Thread Preemption: If the task has a running thread, early termination should be requested if, for any input with semantics PREEMPT, the best available quality version surpasses that on which the thread is running.

Thread Launch: There is "incentive to run" if, for any input:

1. it has semantics FINAL, the final quality version is available, and the task has not yet run on this quality version,
2. it has semantics ANY, a quality version is available, and the task has not yet run at all, or else
3. it has one of the other semantics choices, and the best available quality version surpasses those on which the task has already been run.

A thread is launched if there is "incentive to run" and the task does not already have an executing thread, or else for some input with semantics PARALLEL, its best available quality version surpasses those on which existing threads are running.

Task Removal: The task is finished and removed from the graph if, for all inputs, it has run and completed on the final quality version, or on any version if its semantics are Any.

Whenever a thread is started, the quality version used for each input is the best available, unless its semantics are Every or Parallel, in which case it receives the next higher quality version than its previous execution. For example, a task with two parameters with the semantics Every would be re-run whenever either parameter improved; if a backlog develops for both parameters, they may advance together, so long as no quality version is skipped for either parameter.

4.4.3 Obsolete branch pruning

Petra-Flow also uses the global dependence graph to prune branches that have become obsolete, i.e., have no effect on the application's output other than delaying it. In traditional garbage collection, if a program loses all references to a particular memory allocation, it can be deallocated safely. Likewise, a versioned value can be deallocated when all references to it are dropped. This can happen when a versioned variable that was pointing to it is re-assigned, or a consumer task finishes and is removed from the dependence graph.

We can be more aggressive about deallocation in the Petra-Flow framework than traditional garbage collectors because we have more knowledge about the dependencies. Suppose a producer task is writing to a value that no longer is attached to any versioned variable or consumer task. While a traditional garbage collector in this situation would not have license to act, in Petra-Flow we can prune both the task and the value, conserving on storage, processing and perhaps other resources the task was holding, consuming, or going to consume. In general, a task or value is obsolete unless there exists some path in the dependence graph down to a versioned variable (or a side-effect pseudo-variable for tasks with side-effects). When a branch

of the graph becomes disconnected from any program variable, it can be pruned.

The dependence graph can also be used for automatic pruning at another level as well: individual quality versions and task executions can be managed in exactly the same way. Before we can demonstrate this, we first need to describe the dependence graph at a finer level of detail than is needed for exposition elsewhere.

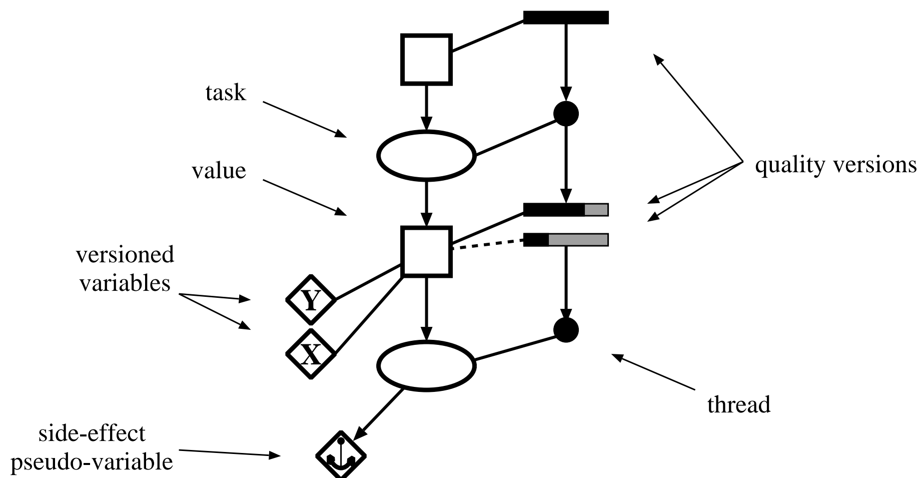

Figure 4.6: *Detailed Petra-Flow Graph.*

Figure 4.6 illustrates an example Petra-Flow graph in fine detail. As before, *tasks* are represented by ovals, which read and write *values*, depicted as squares. The *versioned variables* `X` and `Y`, represented as diamonds, each are linked to their current value, which happens to be held in common. The lower task has side-effects, and so is installed in the graph with an artificial write dependence to a pseudo-variable representing side-effects. This practice ensures that such tasks are not pruned.

In addition to these elements, this detailed graph also shows the individual *threads* (solid circles), and the individual *quality versions* (filled rectangles) they are reading and writing. The lower thread is reading a low quality version while the upper thread is producing a better quality version. Each task is linked to its thread and each value node is linked to its current quality version. The line between the *value* of `X` and

its initial quality version is dotted to suggest that this link exists only if all quality versions must be saved, e.g., to support the synchronization semantics Every. If not, inferior quality versions can be dropped once better versions are available, and so each value node would only point to the most recent quality version.

Having explained the dependence graph in finer detail, the extension of obsolete branch pruning to quality versions and threads is straightforward. The same rule applies as before: a node is obsolete unless it is justified by a dependence path down to a variable. In this figure, none of the quality versions, values, threads, or tasks are obsolete— each is justified by a path to the versioned variable `X` and to the side-effect pseudo-variable. The low quality version near the center of the figure is not obsolete, even without the dotted link to the quality version, due to the path through the thread to the side-effect pseudo-variable.

To prune an obsolete task from the graph, the Petra-Flow framework first cancels any active thread(s) of the task. If there is currently no thread for it (such as while it waits between input quality versions), it can be deleted immediately. Rather than stopping threads dead in their tracks, we cancel them in an orderly fashion so that they can clean up and release any resources they hold. This can be achieved either by (1) setting a flag that the running task polls to discover that it should terminate early, or (2) sending the thread an abort signal, which can be caught by the standard exception handling mechanism to allow cleanup on termination. Both options are available to the programmer. The task can then be removed when all of its threads terminate.

4.5 Priority-Mediated Resources

The remaining component of the Petra-Flow framework is the *priority-mediated resource*. Its purpose is to extend the benefits of resource allocation to existing facilities that are incompatible with asynchronous operation.

Resources that must be used under mutual exclusion, such as legacy software libraries that were not written to be thread-safe, require the use of locks for their correct operation. Yet where prioritized resource allocation and mutual exclusion meet (without the capability for preemption), there arises the classical problem of *priority inversion.* Put generally, priority inversion occurs when a high priority task must wait for a resource that is held by a low priority task. Unfortunately, applications that produce multiple incremental quality results tend to be especially prone to this problem, as they go in and out of resource locks repeatedly to service successive quality versions.

The traditional remedy for priority inversion, *priority inheritance* [82], temporarily grants priority to the task holding the resource so that it can complete quickly, releasing the resource for the blocked high priority task. Unfortunately, this is insufficient in an environment with widely varying resource availability— granting full use of a network that is tremendously slow at the moment is unlikely to get the locking task completed quickly. Note that low priority tasks will tend to have large holding times in this environment during periods of scarce resources and also because their low priority makes them operate slower than high priority tasks.

The approach adopted here is to defer low priority requests in favor of upcoming high priority requests. In this way, we achieve better responsiveness for time-critical actions by trading off the responsiveness of non-critical actions. This is similar to the approach taken by the priority ceiling protocol [83]; see related work in Section 7.2 for a comparison. Naturally, where priority queuing is involved, low priority requests might experience starvation. (This was not found to be a problem, however, for any of the applications we built.)

Tasks that will be wanting a priority-mediated resource state their interest in advance, so that when requests arrive for the resource, the system knows the dynamic priorities of the other tasks that will soon be requesting it. At the time of a request, if no other task of strictly greater priority has an interest, the lock is granted

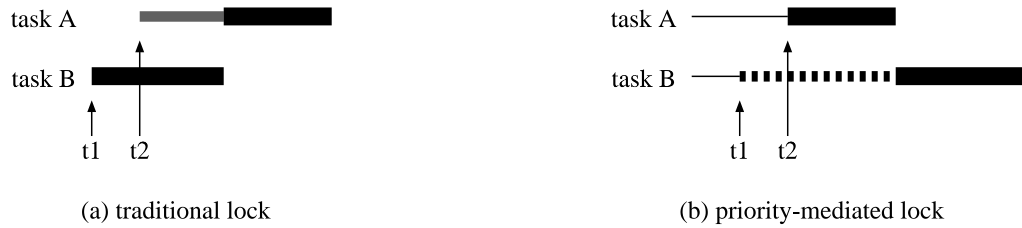

Figure 4.7: *Priority-Mediated Locks: Deferment Avoids Priority Inversion.*

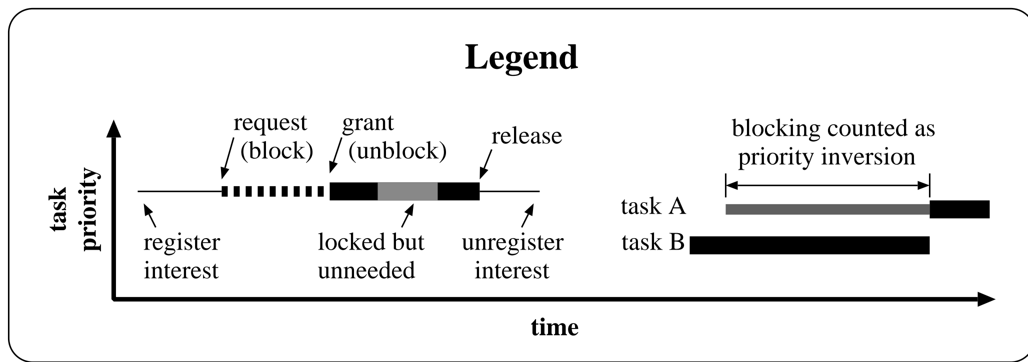

Figure 4.8: *Priority-Mediated Locks: Legend to Diagrams.*

immediately. Otherwise, by choice of the programmer, the request is refused or else queued until higher priority tasks have completed their interest in the resource or priorities change to make the queued request of the highest priority. We name this new type of lock a *priority-mediated lock.*

Figure 4.7 compares the behavior of a traditional lock to that of a priority-mediated lock on an execution in which a low priority task requests the resource at time `t1` shortly before a high priority task requests it at time `t2`.[6] (Even under strict priority scheduling, the low priority task is able to advance whenever the high

[6] Refer to Figure 4.8 for the legend to this and subsequent locking diagrams. The right half of the legend shows an example where task B obtains a lock, then a higher priority task A requests it and blocks. The time until task B releases the lock to the higher priority task is counted as priority inversion, depicted by diagonal dotting of the line that indicates blocking.

priority task blocks, e.g., due to paging or input/output.) With a traditional lock, the first request is granted immediately, causing the high priority task to have to wait for the resource to be released. But with a priority-mediated lock, the resource is effectively reserved for the high priority task by claiming an interest in the resource in advance— the tradeoff being that the completion of the low priority task is delayed. This is not equivalent to having the high priority task reserve the resource by locking it in advance of need. Figure 4.9 demonstrates the difference on an execution in which the relative priorities of the two tasks dynamically reverse. Obtaining a traditional lock in advance of need results in priority inversion, whereas with a priority-mediated lock, the shift in priority results in granting the resource to the blocked task, which now has the highest priority.

Figure 4.9: *Priority-Mediated Locks: Priority Shift Can Unblock a Request.*

Priority-mediated locks exhibit further benefits for repeated requests, as is common with incremental quality processing. When a task releases a resource, it has the option to continue its interest in the resource for upcoming requests. If this option were not available, a low priority task might grab the resource (potentially for a long time) between repeated requests by the high priority task, as illustrated in Figure 4.10. This risk is inevitable with traditional locking on individual requests. On the other hand, this risk can be avoided even without priority-mediated locks by securing a traditional lock once for the series of requests. This, however, bears several comparative disadvantages. If the low priority task were to secure the lock first, the coarser granularity locking would increase the potential for priority inversion.

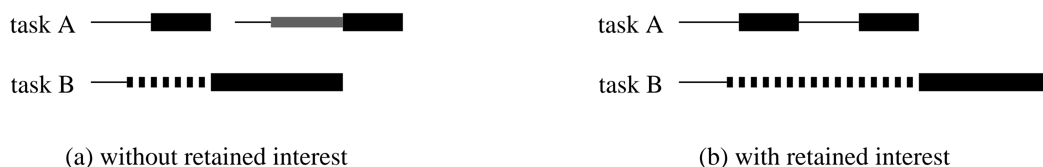

Figure 4.10: *Priority-Mediated Locks: Retained Interest Avoids Priority Inversion.*

The coarser granularity also increases lock contention; the resource could otherwise be multiplexed with other tasks of equally high priority under priority-mediated locking. Further, this scheme is prone to long periods of priority inversion after dynamic priority changes. Figure 4.11 shows the behavior of three locking schemes under a dynamic shift in priorities. Each suffers from priority inversion immediately after the locking task drops below the blocked task in priority. (a) With traditional locks

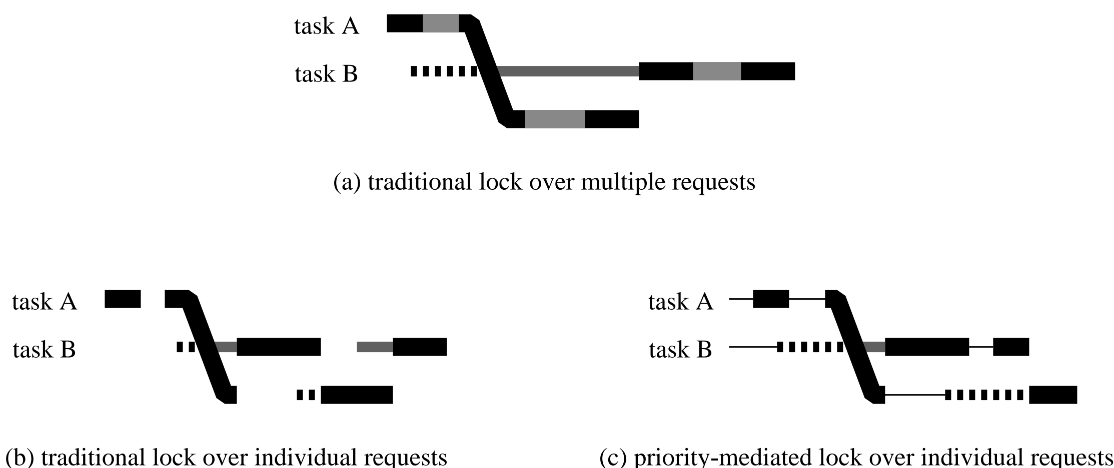

Figure 4.11: *Priority-Mediated Locks: Resilient Under Dynamic Priority Changes.*

over aggregated requests, the coarse grain locking causes the priority inversion to last a long time compared with the other schemes. (b) With traditional locks over individual requests, this priority inversion is resolved more quickly because of the finer granularity of locking, but there is inevitably priority inversion between requests using traditional locks, as discussed earlier. (c) Priority-mediated locks resolve the priority

inversion quickly as in (b) and avoid stalling the high priority task between requests as in (a)— the best of both worlds. The resource can be reserved for the high priority task without any priority inversion occurring if it happens to be released by task A before task B requests it.

4.6 Summary

In this chapter we have introduced a novel framework to help programmers implement application features that help improve responsiveness when resources are scarce. The original concept was spawned from the desire to apply optimizations to the backlog of work that piles up whenever users surpass the ability of resources to keep up. In these circumstances we want the user interface to keep in synch with the user, and have the response time of the ensuing results to be improved by means of (1) delivering results incrementally via multi-resolution techniques, (2) exposing concurrency, (3) controlling resource allocation to deliver important results sooner than peripheral results, and (4) cancellation of work that has become obsolete. These optimizations are enabled by deriving a global dependence graph among concurrent tasks at runtime.

Chapter 5

A PROTOTYPE PETRA-FLOW IMPLEMENTATION

In order to validate the Petra-Flow framework, we constructed a prototype implementation, which is presented in this chapter. We validated the practical adequacy of the model by building several applications with the prototype, as discussed in the subsequent chapter. The importance of building the prototype for validation is demonstrated by the fact that it was only through experience with actual applications written with Petra-Flow that we learned how susceptible such programs are to priority inversion with traditional locks. It was for this reason that priority-mediated locks were added.

When introducing new programming abstractions, there is a tension between realizing them as a library or with language extensions. The former often has the advantage of easier portability and acceptance, the latter cleaner expression. We built the prototype as a library principally, but for the implementation of asynchronous tasks, we built an analog of a remote procedure call (RPC) stub generator [13] to generate the necessarily tedious stub procedures that maintain the dependence graph. (Refer to Figure 5.1 for a diagram of the compilation process.) Unlike most RPC stub

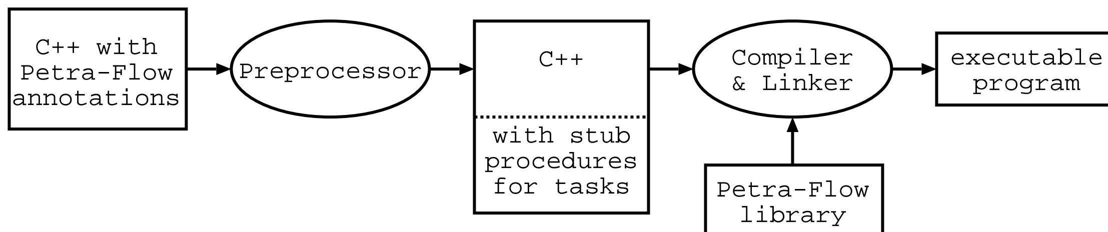

Figure 5.1: *Petra-Flow Pre-processing and Compilation Process.*

generators, which require a separate input file for the generated procedures, the Petra-Flow preprocessor was designed to consume normal source code with task declarations interspersed. It passes the source code through and inserts stubs when it encounters a task declaration, i.e., source-to-source translation.[1] By not imposing constraints on the structure of the source files, we lower the barrier to using a preprocessor somewhat.

The process of developing the prototype involved (1) selecting a language and platform on which to build, (2) devising syntactic annotations for asynchronous tasks and a preprocessor to accept annotated source code and generate appropriate procedure stubs, (3) fashioning a runtime library that supports the Petra-Flow abstractions and services, and (4) integrating support for debugging. These are discussed in turn below.

5.1 Language and Operating System

The Petra-Flow framework is independent of any particular language or operating system, yet these must be chosen in order to build a working system. To demonstrate its practicality, it is worthwhile to select a widely used and performant language; we selected C++. Its *template class* mechanism in particular facilitates the seamless integration of versioned variables; the template parameter accepts any user-defined or built-in type for which versioning is to be provided. (In languages that lack this facility, such as Java, a textual preprocessor can generate the type-parameterized classes, as is done for prototype classes in the GNU C++ library [52].)

Our requirements of an operating system are that it support kernel threads (as opposed to just user-level threads, which block the entire application when any individual thread blocks for operating system service), dynamically adjustable thread

[1] The preprocessor also generates source-level debugging information so that the compiler and debugger can refer to the correct line numbers in the original source file.

priority, and thread cancellation that allows for cleanup. We selected the POSIX-standard Pthreads interface on DEC OSF/1 version 3.2 (a flavor of UNIX that runs on DEC Alphas); no other supported platform available to us at the time supplied kernel threads. This threads package enables clean termination of a thread by sending it an abort signal (`pthread_cancel(thread)`), as opposed to the more widespread interface for killing a thread instantly (e.g., `lwp_destroy(thread)` in SunOS). Threads can dynamically select whether the signal is to be received asynchronously or held until the thread polls `pthread_testcancel()`. When the signal is received, it causes a runtime exception, which can be caught using the `TRY-CATCH-FINALLY` exception handling mechanism supplied by DEC or by the C++ language standard [89, r.15]. This provides an avenue for threads to deallocate any resources they are holding before terminating.

5.2 Annotations for Asynchronous Tasks

An asynchronous task is declared just like a normal procedure, but with two additions: (1) it is preceded by the word "`task`,"[2] and (2) each input and output managed by Petra-Flow receives an annotation specifying its synchronization semantics. Table 5.1 lists the synchronization choices (described in Section 4.4.2) that were implemented.

Table 5.1: *Choice of Synchronization Semantics.*

`read_final`	run only on the final quality version
`read_any`	run once on any available quality version
`read_skip`	run on each new version, skipping backlog
`write_once`	will submit a single quality version
`write_multi`	will submit a series of increasing quality versions

[2] To avoid name-space conflicts, all names associated with the prototype are prefixed with "`TG`" (Task Graph). The names have been changed in this thesis for a clearer presentation, e.g., "`task`" instead of "`TGtask`."

An example task declaration is given below:

```
task Mytask(int i, read_skip int j, write_multi int k)
```

This task will be executed on each new quality version that becomes available for the parameter `j`, skipping any backlog that might accumulate while this task is executing on a prior quality version. It will generate multiple quality versions for the output parameter `k`. Notice that traditional parameters can be included as well. The value passed in for `i` will be handed to the task each time it is run; this works for normal reference parameters, as well.

The preprocessor generates three procedures for each asynchronous task `Foo`:

_User_Foo: This procedure contains the user's source code for the task *verbatim*. It is renamed so that we can interpose on calls to the task. Because quality versions may be large objects, they are passed as reference parameters to task inputs; C++ performs the dereferencing implicitly. Task outputs are references to Petra-Flow *value* nodes, for which the essential public operations are to (1) submit a quality version, and (2) get a reference to the previously submitted version, or null if none has been submitted yet. The latter call allows tasks to generate improved quality versions by building on the previous one.

Foo: This is the routine actually invoked directly by the programmer's calls. It installs the task into the dependence graph, allocates new values for its output parameters (which are assigned to versioned variables), and marshals the arguments into the task node for use later. Finally, if the input synchronization semantics are already met, it starts a thread on the procedure `_Thread_Foo`. If not, the responsibility for starting a thread for this task lies with the runtime library, specifically when new quality versions are submitted on the inputs.

_Thread_Foo: New threads start at this procedure. It establishes the default cancellation mode (asynchronous reception of cancel signals is turned off) and

sets the thread priority according to the task. It then unmarshals the arguments from the task node, obtains handles to the appropriate quality versions, and calls `_User_Foo`. This call is made in a `TRY-FINALLY` scope so that it can perform cleanup regardless of uncaught exceptions.

When the user's function returns, the synchronization semantics are checked against the currently available input quality versions. If warranted, it loops back and calls `_User_Foo` again. If the task is finished forever, it is removed from the graph. The thread then terminates, freeing its stack for use by other threads. If the task remains in the graph, a thread will be started anew when its synchronization semantics are met again.

Alternately, one could have designed the prototype to keep the thread throughout the task's existence, blocking it while the task does not need to execute. This would consume more memory for the stack space of blocked threads (the default stack allocation is over 20 KB for OSF/1 Pthreads) and would not admit as cleanly the parallel execution of threads on different quality versions for a given task.

Simplifications for the Prototype

Ideally, tasks should be able to use Petra-Flow services for global variables as well as for parameters. Support for globals was omitted from the prototype implementation, however, because the C++ *default parameter value* feature can be easily applied by hand to mock up the support of globals. For example, a task that reads versions from the global versioned variable `I` and writes versions to the global versioned variable `O` would be declared thusly:

```
task Globtask(read_skip int I = I, write_versions int O = O)
```

From the perspective of the call site, it is just as though task globals were supported, i.e., "`Globtask()`." The body of the task can use the global names, although they

are in fact parameters. The shortcoming of this approach is that such parameters must be passed along to any subroutines the task calls that need them.

The notational luxury of functional return results was also dropped from the implementation effort, although it would be straightforward to add. Instead, tasks use output parameters. So, the call site looks like "`UserTask(var)`" instead of "`var = UserTask()`."

Some tasks have important side-effects, such as printing output to the user, and cannot be pruned. The notation "`output task Mytask(...`" is designated to mark such tasks, however, this was omitted from the prototype implementation. Instead, programmers manually attach such tasks to dummy output variables, e.g., a task that displays something for the user and also generates quality versions for an output parameter would be given a second output to prevent it from being pruned if its first output became obsolete in the course of execution. (As a special case, tasks are assumed to have important side-effects if they have no output parameters whatsoever, such as the print task in the slide browser example, otherwise there would be no point in creating them.)

5.3 Runtime Library

The remaining Petra-Flow abstractions are furnished by the library.

5.3.1 Versioned Variables

Versioned variables are the essential abstraction with which the programmer interacts. They are defined as a template class that takes a type as its parameter, e.g.,

```
versioned<any_user_type>  variable1, variable2, ...;
```

The public methods of versioned variables are to (1) set and get their priority, (2) assign them to one another, (3) get a reference to their current quality version, or null if none has been submitted yet, and (4) reset a variable so that it again has no

value (inciting branch pruning). An assignment "`X = Y`" is by reference to the current value of `Y`, which may have an associated task generating improved quality versions. Unlike traditional assignment by reference, `X` is unaffected if `Y` is later reassigned to a new value because of the versioning.

An Example

At this point we have introduced enough of the notation to demonstrate what Petra-Flow code looks like in practice. Figure 5.2 shows a simple producer-consumer

```
task Producer(write_multi float out)
{
    loop N times {        // generate incremental quality versions
        float* qv;
        qv = user computation generating new quality version,
             perhaps based on previous quality version *qv
        out->Submit(qv);  // out = *qv;
                          // quality version *qv is now read-only
    };
}

task Consumer(read_skip float in)
{
    printf("%g", in);
}

main()
{
    versioned<float>  V;
    Producer(V);
    Consumer(V);
    Producer(V);            // writes to a new version of V
    Consumer(V);
    V.SetPriority(1);       // whatever V depends on shall have priority
}
```

Figure 5.2: *Petra-Flow Producer-Consumer Example.*

example. The main program declares a versioned variable `V` and then launches producer and consumer tasks that read and write to this variable. Two versions of `V` exist simultaneously if the first pair of producer and consumer tasks have not completed when the second producer task is launched. From the point of view of the programmer, however, the semantics are that of a sequential execution. Splitting the variable into multiple versions is handled by Petra-Flow transparently. The final statement in `main` boosts the priority of the variable and, indirectly via Petra-Flow services, also raises the priority of the producer task that `V` depends on— that is, the second producer task. No other task in this example receives priority. (A sample dependence graph for this program will be shown shortly in Section 5.4.)

Each producer task generates and submits multiple quality versions (using the method `Submit`), causing the consumer to be re-run on each new quality version, skipping backlog.

5.3.2 Priorities

The prototype implements integer priorities with programmer-selectable options for determining how priorities are to be combined (sum or maximum) when propagated up the global dependence graph, and how priorities are mapped to the operating system's notion of priorities. Unfortunately, OSF/1 allows only five discrete priority values for unprivileged processes, thus Petra-Flow priorities in general must be clipped or compressed into this range. Worse, priority differences have only a small effect under the priority-weighted time slicing policy (`SCHED_OTHER`); in running two threads at different priorities, each discrete step in their priority difference grants only about 10% more processor time to the favored thread, for a maximum of 40%. We would like this factor to be adjustable up to an order of magnitude at least. At the extreme, the programmer can select strict priority scheduling instead.

The notion of Petra-Flow is independent of any specific priority mechanism. With only minor changes to the prototype, programmers could be allowed to replace the

standard priority class with their own implementation having its own representation and methods for combining and mapping down to operating system priorities. This flexibility could be used, say, to implement the priority token-sets developed in Section 4.4.1.

The current implementation avoids unnecessary traversal of the dependence graph when propagating priority changes. For example, if we are combining priorities by taking the maximum and the priority up-flow from an arc rises from 3 to 5 while another arc to the same node has priority 10, we need not propagate the priority change any further up the graph. This feature could be preserved even with user-defined priority classes by having the method that merges priorities indicate when no substantial change takes place in a merge. Without this, we would need to propagate priority changes all the way to the top leaves of the dependence graph.

Once the prototype was built, we discovered another source of unnecessary graph traversal that should be eliminated in any future implementations. In assignments of the form "`X = UserTask(X)`" where `X` has priority associated with it, Petra-Flow propagates priority changes up the graph twice: once to rescind the priority boost when `X` is detached from its old value, and again to restore the identical priorities when `X` is attached to the new value being produced by `UserTask`. This was not a significant source of overhead for the applications we built with the prototype.

Finally, the prototype provides an interface to install and remove callback routines to be invoked when a task changes priority. This has been used, for example, in an implementation of prioritized TCP sockets that connect a task's priority with the network socket without any programmer involvement.

5.3.3 Reference Counter

When a quality version is submitted, the system can (optionally) take the responsibility for deallocating it when it becomes obsolete. Because C++ objects can be allocated in different ways, the submit routine takes an optional parameter to indic-

ate whether it should be deallocated with "`delete p`," "`delete[] p`," or "`free(p)`." This is difficult to arrange in C++. One cannot simply pass the address of the appropriate delete method; this is forbidden in C++[89, r.12.4] and besides, each method requires a *different syntax*. The solution involves obtaining a pointer to one of three template functions that are parameterized by the user's type and deleting the object with the appropriate method.

The prototype provides a reference counter module for this service. It maintains a hash table on memory addresses to keep the reference count and a pointer to the appropriate deallocation function. Its interface is made public so that programmers can also use it for objects that are not managed by Petra-Flow, or to increment and decrement reference counts for managed quality versions for special needs unknown to Petra-Flow.

5.3.4 Task Services

For task cancellation, Petra-Flow sends an abort signal to the thread as well as providing a polling interface for tasks that are not equipped to be canceled at any moment. Tasks can enable asynchronous cancellation with the incantation:

```
pthread_setcancel(CANCEL_ON);      // enable (synchronous) cancels
pthread_setasynccancel(CANCEL_ON); // enable asynchronous cancels
```

This can be put in a `TRY-FINALLY` scope if there is cleanup code that must be performed whether or not the task is canceled. If special actions must be performed only on cancellation, the programmer can catch the exception `pthread_cancel_e`.

Finally, the library also provides an interface for determining the number of outstanding tasks or waiting until there are none, e.g., before terminating the process.

5.4 Debugging Support

Petra-Flow presents a rare opportunity to provide the programmer with high-level debugging information, because it automatically maintains a global view of what is

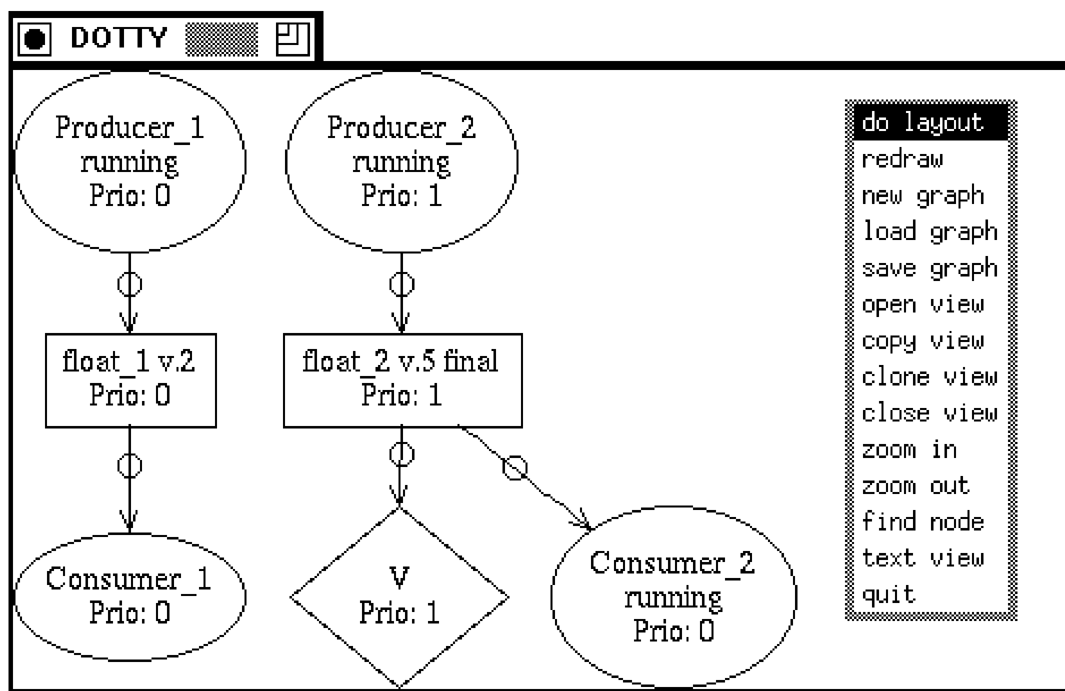

Figure 5.3: *Producer-Consumer Dependence Graph Snapshot Viewed with Dotty.*

going on during execution. We built in several forms of such debugging output to validate that this support is practical. In our experience, we found them helpful, and so we discuss them here as an observation about the prototype.

The prototype can generate a snapshot of the global view at any time on request. Alternately, by turning on snapshot tracing, a history of the global view is saved to disk. Snapshots of the dependence graph can be viewed graphically with `dotty`, a graph editor and automatic layout tool from AT&T in the `graphvis` package [67]. Figure 5.3 shows an example dependence graph snapshot visualized with `dotty`. Tasks are represented by ovals, values by squares, and versioned variables by diamonds. (The small circles on the dependence arcs are added by `dotty` as handles for manipulating the arcs.) The text within each node shows its unique name on the first line, and its (derived) priority on the last. The text for a task node also indicates whether a thread is currently running for that task. The text within a value node

shows the number of quality versions submitted so far, and whether the current best quality version is known to be final. This particular dependence graph is from the producer-consumer example program in Figure 5.2.

We provide a simple shell script to view a trace history as a slide show. Although `graphvis` is a powerful and flexible tool, it does not have the capability to animate smoothly the transition from one snapshot to the next. Also, because there is no way to enforce coherence between layouts, nodes sometimes jump around unintuitively.

In addition, the prototype can generate a time-line trace showing in different columns when tasks produce and consume values, quality versions, etc. The programmer can select independently classes of events to be included in the time-line, i.e., those dealing with tasks, threads, priorities, cancellation, quality versions, or versioned variables.

Meaningful labels on the debugging output are derived from several sources. The preprocessor generates code to record names for tasks and values. Versioned variables can be given a descriptive label at their declaration by an optional argument to the class constructor. This can be specified either explicitly by the programmer, or more conveniently with the help of a supplied macro that declares the variable and also passes its name as a string (`#var`) to the constructor:

```
#define declare_versioned(type,var)    versioned<type>  var(#var)
```

Arrays in C++, however, do not permit an argument to the constructor[89, r.5.3.3]. For this case and others, a descriptive label may be explicitly assigned to any dependence graph node. When a task submits a new quality version to Petra-Flow, it can be given a descriptive label via an optional parameter; this is most useful for noting the quality in domain-specific terms, such as image resolution, instead of just the version number, which is the default.

Finally, because of the high level view established by Petra-Flow, the system can also detect certain programming errors, for example, the use of a versioned variable

before any assignment has been made to it or an upstream producer task terminating without producing any quality versions.

5.5 Conclusion

For the implementation of asynchronous tasks, we view the choice to use source code annotations and a preprocessor to generate the needed stub routines a good one—the structure of the programmer's source code is unconstrained and the preprocessor generates a substantial amount of tedious code, most of which would otherwise need to be written by the programmer. For example, it generates about 90 lines of code for a task with one input and one output (involving only three words of annotation). More specifically, it generates approximately 70 lines per task plus an additional 9–13 lines per task parameter.

Note that the prototype does not involve a specially designated master thread to implement the Petra-Flow services. The worker threads themselves maintain the graph and perform services for other tasks, such as launching a thread when a task's inputs are ready or adjusting task priorities throughout the graph.

In the process of building applications with the prototype, opportunities arose for code reuse across applications, so we constructed reusable components for some of them, including a multi-versioned image class and routines for iterating across an abstract 2-D integer parameter space for quality versions. For tasks that use the network, we provide a prioritized TCP stream class that transparently employs priority update callbacks (described in Section 5.3.2) to conduct task priority changes to the network connection.[3] While these are not strictly part of Petra-Flow, such auxiliary support naturally complements any Petra-Flow implementation.

As some measure of the effort involved in building the prototype, we list here

---

[3] Since the operating system does not provide for prioritized network connections, we also had to build an implementation of this service, as discussed in Section 6.1.4.

the approximate number of lines of C++ code (including comments) that compose these software artifacts. (The use of the LEDA C++ class library helped raise the level of programming abstraction significantly [60].) The preprocessor is 500 lines and the runtime library is 2200 lines, 200 of which are for debugging support. Of that total, the reference counter module comprises 275 lines, and priority-mediated locks 500. The auxiliary support amounts to nearly 600 lines, including prioritized TCP streams, but excluding the implementation that was needed to supplement the operating system with network priorities.

Chapter 6

EVALUATION

Having presented the Petra-Flow framework and its conceptual benefits, we turn now to its practical evaluation. This involved building the prototype implementation atop DEC Alpha OSF/1 using POSIX Pthreads, as described in the preceding chapter, and applying it to build multiple applications. We are going to look at two kinds of measures principally: coding effort and runtime performance. For coding effort, we use two metrics: code size and elapsed coding time. For one of the programs we built, a Mandelbrot fractal explorer, there are over a dozen similar programs available by anonymous FTP, and so we are able to compare coding effort across similar applications built without Petra-Flow. Naturally, such comparisons are coarse and cannot provide indisputable evidence that Petra-Flow eases the programming task; nevertheless, the comparison does make a compelling case.

For runtime performance, we present three kinds of measures: the macroscopic percentage overhead for Petra-Flow, the absolute timings of individual micro-benchmarks compared against equivalent actions using raw Pthreads, and the response time for particular benchmarks under the presense and absence of individual features facilitated by Petra-Flow.

Additionally, we developed a lock simulation, calibrated with dynamic measurements from one of the applications, to evaluate the cost and benefit of priority-mediated locks.

This chapter is organized as follows. The first section describes three applications we built, and in addition, a testing harness that supports network priorities (which are lacking from the base operating system) and allows us to control (simulated)

network conditions dynamically. The static analyses are grouped in Section 6.2, and the dynamic analyses are grouped in Section 6.3. Finally, Section 6.4 presents the lock simulation.

6.1 Applications

The three non-trivial applications we constructed are listed below. Each was built to explore a different critical resource.

Name	Application	Domain	Critical Resource
pDatabase	database front-end	text	remote server
pFractals	Mandelbrot fractal explorer	images	processor
pAlbum	Web browser for photo albums	images	network

In the sections below, we describe each application, how it benefits from Petra-Flow, and key points about its implementation. Keep in mind that the challenge for Petra-Flow is to improve responsiveness during periods of poor service. Hence, we focus on what can be done when resources are scarce.

These applications were written for X Windows atop several libraries: the graphical user interface is provided through Tcl/Tk extended for color photos [69], database communication is parsed with Expect [54], and general data structures are provided by LEDA [60].

6.1.1 Database Front-End

The first application that was built is a database front-end that interfaces with existing back-ends supported by the University of Washington for campus-wide access, including the library catalog, Grolier's Encyclopedia, Books in Print, and the INSPEC database of periodical abstracts. The critical resource for this application is the back-end database. Service varies widely with query complexity and with user contention, especially for the library catalog.

Database: LCAT – UW Library Catalog

Query: Forman and George

Query matched 19 records.

Title List:

```
Tannenbaum, Samuel A|Shaksperian scraps, and other Elizabethan fragment|1933
Forman, Maurice Buxt|George Meredith; some early appreciations.        |1909
Eliot, John, 1933-. |Children's spatial development.                   |1974
Forman, Maurice Buxt|Bibliography of the writings in prose and verse of|1922
Forman, Maurice Buxt|Meredithiana; being a supplement to the Bibliograp|1924
Smith, Robert Metcal|Shelley legend.                                   |1967
                    |Action and thought : from sensorimotor schemes to |1982
                    |Joys of research.                                 |1981
Brown, Forman George|Small wonder : the story of the Yale Puppeteers an|1980
National Forest Advi|Report on the problem of mining claims on the nati|1953
```

Full Record:

```
YR   1981
DT   The Joys of research / Walter Shropshire, Jr., editor.
PI   Washington, D.C. : Smithsonian Institution Press, 1981.
SH   Einstein-Albert-1879-1955 -- Anniversaries-etc.
     Research -- Congresses.
     Science -- Congresses.
LL   Suzzallo/Allen.  Odegaard Undergraduate.
IT   Natural Sciences General Stacks Q179.9 .J69 1981.
     Undergraduate General Stacks Q179.9 .J69 1981.
OA   Shropshire, Walter.
     Einstein, Albert, 1879-1955.
     Smithsonian Institution.
CO   Contents: Einstein and research / Paul Forman --
     Biochemical pharmacology / Julius Axelrod -- Mathematics /
     I.M. Singer -- Oncology and virology / Howard M. Temin --
     Theoretical astrophysics / George B. Field -- "On first
     hearing" : the act of creation in music / William Schuman
     -- Biomedical investigation / Rosalyn S. Yalow --
     Geophysics / J. Tuzo Wilson -- Chemistry / Linus Pauling --
     Evolutionary biology / Ernst Mayr.
NT   Talks presented at a colloquium celebrating the centennial
     of the birth of Albert Einstein, held Mar. 16-17, 1979 at
     the Smithsonian Institution, Washington, D.C.
     Includes bibliographical references and index.
```

Figure 6.1: *Snapshot of the Database Front-End.*

A screen snapshot of this application is shown in Figure 6.1. From top to bottom, users can (1) dynamically select which database back-end to connect with, (2) enter queries, (3) view the list of "hits" to a query, and (4) view the full record for a selected hit. The standard front-ends provided by the University of Washington, Wilco and Willow [43], have additional features, such as the ability to print or e-mail results, select which fields to display, and generate sophisticated queries via menus and forms rather than with complicated query syntax. These amenities are not essential to the application, and so were not included in our implementation.

On the other hand, the Petra-Flow framework facilitates features either not present at all or not as pervasive in each of the the standard front-ends and which are especially beneficial when database service is slow:

Asynchrony: The operation of the user interface is permitted to outpace request completion. For example, one can enter a query while the database connection is still being established (which takes about six seconds even during periods of good service); when the connection is finally made, the query is put through.

Wilco and Willow do not permit further user input while in the process of connecting to a database, executing a query, or retrieving a full record. Even were they to allow buffered user input during these times, asynchronous operation of the user interface is nevertheless superior. To illustrate this, consider what happens if, while waiting for the database connection, a user enters a query and then another one, obsoleting the first. Under Petra-Flow, the first query task would be constructed and deleted before it ever had a chance to execute, but under a traditional interface with buffered input, the user must either wait for both queries to complete, or manually interrupt the program (if it is provided for, e.g., by the escape key) to flush the no longer desired query. This flushes the desired one, as well, and possibly also terminates the connection to the database, depending on how the program handles the interruption.

Incremental Quality: Text from the database is retrieved and displayed incrementally from top to bottom, both for hits and for full records. The former is needed even when service is fast, because a query may generate a huge number of hits. This is the only case for which the standard front-ends implement incremental retrieval.

Prefetching & Caching: Records are prefetched and cached for the user, whereas the standard front-ends re-fetch a record each time the user selects it. The implemented policy is to prefetch the first hit immediately, and whenever a user demands a record, to prefetch the subsequent one as well.

Priority: The connection to the back-end database allows only synchronous operation per user session. It is therefore represented to Petra-Flow as a priority-mediated resource so that it is arbitrated in an intelligent way among the multiple incremental fetching tasks.[1] The simultaneously outstanding tasks may include multiple prefetches and old demand fetches, a current demand fetch that is to be displayed, and the ongoing retrieval of hits. Logically, priority flows from the visible window panes to the appropriate tasks. In practice, high priority is assigned to the variables representing the window panes for hits and for the full record. The latter is given an additional priority boost whenever the mouse points to the full record. The effect of these policies is that fetches whose results are immediately visible take priority over other fetches, and the demand fetch takes strict priority over all other activities when the mouse hovers over the full record window. Otherwise, activities of equal priority interleave among incremental quality versions.

Recall that Petra-Flow is independent of the graphical user interface. The

[1] Priority-mediated resources are also used to represent software libraries that were not written to guarantee thread-safety, such as Tcl/Tk and the license acquisition routines re-used from Wilco.

responsibility for translating mouse movements to changes in the priority of the record pane variable falls to the application code (in our case, two Tcl/Tk event bindings and four lines of C++). Petra-Flow then conveys this priority information up the dependence graph to the right tasks.

Cancellation: Because records are cached, selecting a different record to view does not cancel the old fetch, but simply lowers its priority to that of a prefetch. When the user enters a new query or selects a new database, the outstanding tasks are canceled implicitly when the variables are reassigned. In contrast, the standard front-ends require the user to explicitly cancel the current activity before he or she can begin entering new commands (with the exception that commands are accepted while hits are being downloaded).

The implementation uses four kinds of asynchronous tasks: (1) establish a connection to a back-end database, (2) submit a query and retrieve the hits incrementally, (3) fetch a full record incrementally, and (4) display each version of a record to the lower window pane. Figure 6.2 shows how these tasks interrelate in an example dependence graph. (In actual operation, the task that establishes the connection would be terminated before any fetching tasks are launched.) An array of versioned variables, allocated as needed, is used to cache records. In the figure, the degree of priority is illustrated by line width. The window pane displaying the full record currently has the highest priority— as when it contains the mouse. This priority flows up through the display task to one of the record versioned variables and then to the appropriate demand fetch task. The "hits" window pane carries an intermediate priority level between demand fetching and prefetching.

Figure 6.3 shows the structure of the program in pseudo-code. (Although the prototype does not support function return results, we have used this notation in this pseudo-code to make the input/output relationships clearer.) The initialization in `main` establishes the priorities of the two window pane variables and the event loop

adjusts the priority of the record window pane whenever the mouse enters or leaves it. When the user enters a query, a query task is launched and the first record is prefetched. When the user selects a particular hit to view, it is demand fetched (if it does not already have a prefetching task) and a display task is started that will display each successive quality version that is brought in. In addition, a prefetch task is started for the following record, unless it already has one.

6.1.2 Mandelbrot Fractal Explorer

The second application generates Mandelbrot fractals, which are computationally intensive [56]. This implementation uses Petra-Flow to help maintain responsiveness even when the requested computations dominate the available processing resources by far. Figure 6.4 shows a screen snapshot. There are essentially three user commands: (1) zoom in on a region of a fractal, (2) zoom back out to the starting position, and (3) re-color a fractal by a somewhat computationally intensive procedure (histogram

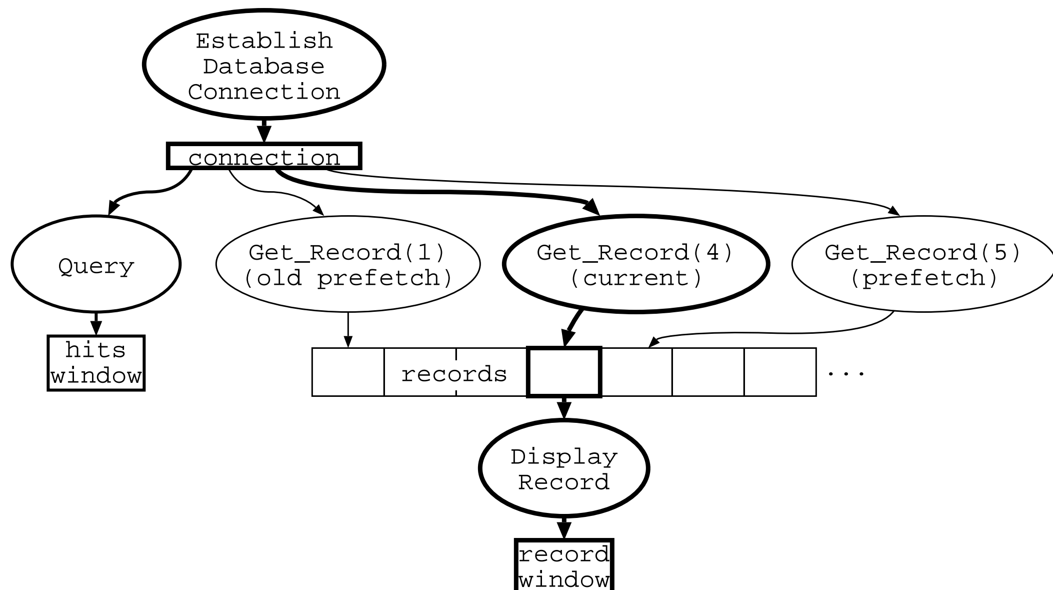

Figure 6.2: *Example Dependence Graph for Database Front-End.*

```
versioned<socket>        connection;    // current database connection
array<versioned<string>> records;       // full records to current query
versioned<hwindow>       hits_window;   // upper Tcl/Tk window pane
versioned<rwindow>       record_window; // lower Tcl/Tk window pane

// establish a connection to a database back end
task write_once socket Establish_Connection(string database_selection);

// send a query to back end and retrieve/display hits incrementally
task write_multi hwindow Query(string query, read_any socket connection);

// fetch a full record incrementally a few fields at a time
task write_multi string Get_Record(int record_num, read_any socket connection);

// display full record in lower window
task write_multi rwindow Display_Record(read_skip string record);

main()
{
   hits_window.SetPriority(1);
   record_window.SetPriority(1);

   user event loop {
     case mouse enters record window:  record_window.SetPriority(2);
     case mouse leaves record window:  record_window.SetPriority(1);

     case select database:
               connection = Establish_Connection(user's selection);

     case query:
               hits_window = Query(user's selection, connection);
               records[1] = Get_Record(1, connection);         // prefetch

     case view record i:
               if (records[i] is unassigned)
                   records[i] = Get_Record(i, connection);     // demand fetch
               record_window = Display_Record(records[i]);
               if (records[i+1] is unassigned)
                   records[i+1] = Get_Record(i+1, connection); // prefetch
   }
}
```

Figure 6.3: *Program Structure of* pDatabase.

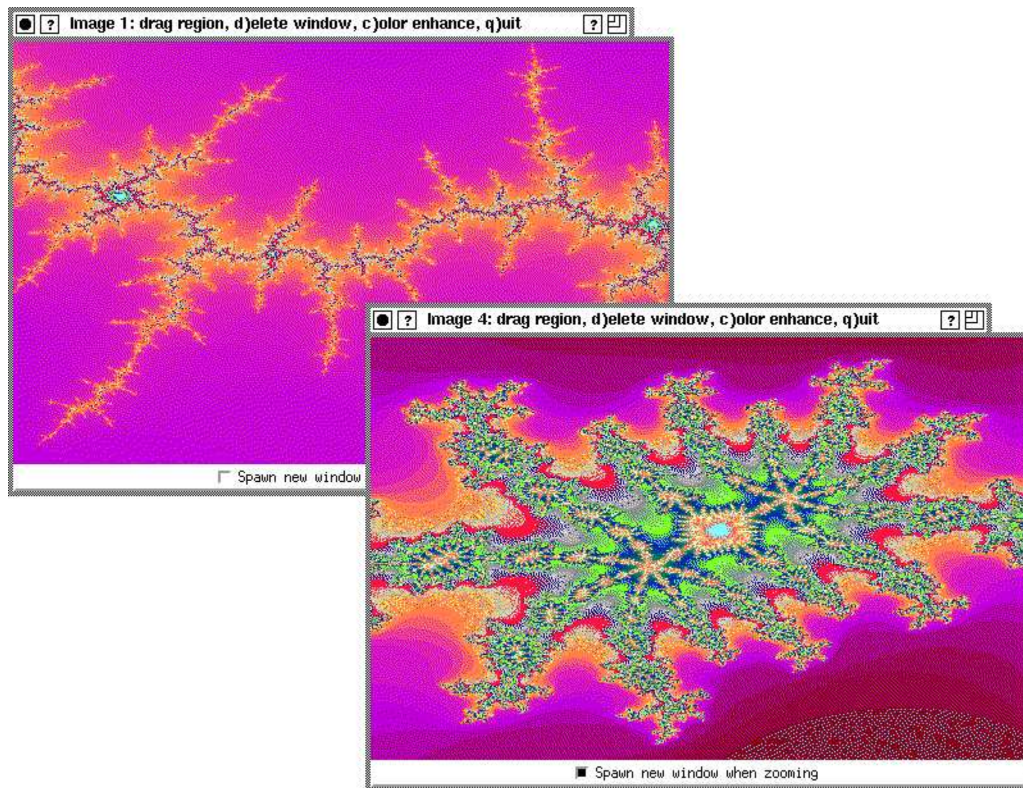

Figure 6.4: *Snapshot of the Mandelbrot Fractal Explorer.*

equalization). A toggle switch at the bottom of each window selects whether the contents of the window may be replaced or a new window should be spawned. There is a corresponding command to delete a window.

Like the first application, Petra-Flow supports a number of features for enhancing responsiveness in this application:

Incremental Quality: A low resolution version ($\leq 64^2$ pixels) of a fractal is put up at once, and its resolution is refined as computational resources allow. Image resolution quadruples with each new version. This application domain admits another dimension for incremental quality versions: the maximum number of Mandelbrot iterations attempted at each point. Since this is of no benefit on large portions of the fractal where iteration terminates quickly,

our implementation does not take advantage of this (nor does any other fractal program that we have found).

Asynchrony: Multiple windows can be refining simultaneously and new commands can be accepted at any time, unlike most fractal programs available on the Internet that must finish the current fractal before beginning the next user request. The main program thread, which processes the event queue, has the highest non-privileged priority available from the operating system so that new events are noticed even when there is work outstanding.

Cancellation: Branches of the dependence graph that become detached from a window are pruned automatically by Petra-Flow. This may occur, but does not always occur, when a window is deleted or its contents are replaced. For example, if one starts a new fractal zoom and immediately requests a recolored copy in a newly spawned window, the task that is generating the fractal would be canceled only when both of these windows are replaced or deleted.

Priority: The application prioritizes among simultaneously generating fractals to give better responsiveness in the window(s) that hold the user's attention, as determined approximately by the position of the mouse and window state: All windows are given a small degree of priority. While a window is iconified its priority is dropped to zero. A priority boost is given to the window containing the mouse, if any. There are other cues that could also have been incorporated but were not, such as the window stacking order and overlap. In practice, we find our attention is frequently focussed on the most recent zoom. This rule could be included easily.

This application is built from three kinds of asynchronous tasks for: (1) generating a fractal incrementally, (2) re-coloring each version of a fractal, and (3) displaying

each version of a fractal to a particular window. The first two task types accept cancellation signals asynchronously, imposing no overhead for polling. Figure 6.5

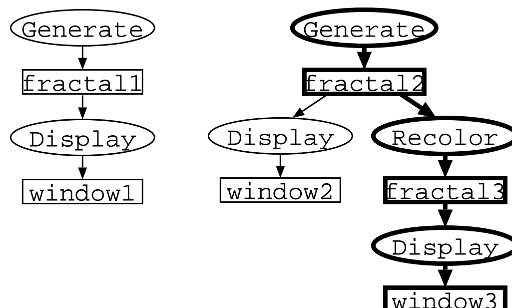

Figure 6.5: *Example Dependence Graph for Fractal Explorer.*

shows a hypothetical snapshot of the dependence graph, clarifying the relationships among tasks. As before, priority is indicated with bold lines. In this situation, one window has high priority, and this propagates up through a display task, a re-color task, and all the way up to the fractal generator task. In the actual program, the fractal and window variables are organized as dynamically extendible arrays.

The structure of the `pFractals` program is shown in Figure 6.6 in pseudo-code. The variable `coordinates` is used to map from screen coordinates to Mandelbrot space. There is no need to use versioning for this variable, because its maintenance requires only a few floating point operations and should be reasonably quick in any environment.

The program begins by launching two tasks— one to compute a canonical Mandelbrot fractal (incrementally) and another to display it. The first two cases of the user event loop translate user interface actions to priority adjustments for the individual windows. When zooming or re-coloring a fractal, we may spawn a new Tcl/Tk window first, depending on the state of the toggle button at the bottom of the particular window.

We call attention to the case where a window is deleted. The versioned variables

```
array<pair<complex>>      coordinates;   // regions in Mandelbrot space
array<versioned<image>>   images;        //      & their fractal images
array<versioned<window>>  windows;       // Tcl/Tk window panes

// compute fractal image
task write_multi image Compute_Fractal(pair<complex> corner_coords,
                                       int width, int height);

// re-color an image by histogram equalization
task write_multi image Recolor(read_skip image in);

// display image in Tcl/Tk window pane number i
task write_multi window Display(read_skip image in, int i);

main()
{
   coordinates[1] = pair(complex(-1.5,-1), complex(0.5,1));
   images[1] = Compute_Fractal(coordinates[1], 640, 480);
   create Tcl/Tk window 1
   windows[1] = Display(images[1], 1);

   user event loop {
     case window i gets mouse or de-iconified:  windows[i].AdjustPriority(+1);
     case window i loses mouse or iconified:    windows[i].AdjustPriority(-1);

     case zoom a sub-region of window i:
                // first we spawn a new Tcl/Tk window, if called for
                d = replace_window ? i : new destination window;
                map mouse drag through coordinates[i] to get coords
                coordinates[d] = coords;
                images[d] = Compute_Fractal(coords, 640, 480);
                windows[d] = Display(images[d], d);

     case recolor the fractal in window i:
                d = replace_window ? i : new destination window;
                coordinates[d] = coordinates[i];
                images[d] = Recolor(images[i]);
                windows[d] = Display(images[d], d);

     case delete window i:
                windows[i].Free();
                images[i].Free(); // detaches image value, but it may not be
                                  // obsolete if used by another window
   }
}
```

Figure 6.6: *Program Structure of* `pFractals`.

Figure 6.7: *Snapshot of the Photo Album Web Browser.*

representing the window and the fractal are both cleared, that is, detached from any value. This is equivalent to a versioned variable that has been declared, but as yet has no value assigned to it. When the values are detached, they may become obsolete and be pruned. A display task will certainly be pruned, but a fractal generator task might not be if, say, a non-obsolete re-color task still depends on its result.

6.1.3 Photo Album Web Browser

The third application was chosen to be network intensive in order to demonstrate the benefits of Petra-Flow in, say, a wireless environment. It is a Web browser for collections of pictures, akin to an album of developed photographs labeled with descriptive titles.

Figure 6.7 shows a screen dump of this application. In the top text entry field, users enter a URL address for a photo album stored at a Web server. An album file

contains titles and Web addresses for a sequence of images, each of which is stored in a multi-resolution format. These are displayed in a horizontally scrollable strip just below the text entry field. The photos are fetched in parallel using separate asynchronous tasks, each with an independent, prioritized[2] TCP connection to the HTTP server, and are displayed incrementally. One such task is launched for each photo that is visible in the strip, plus an additional one off to the right of the strip. This prefetching task (plus any others that have been launched but are no longer visible in the scrolling strip) automatically operate at lower priority by the Petra-Flow priority mechanism, which is used to grant priority only to those tasks associated with a visible window pane. This application uses the summation policy when combining priorities, rather than the maximum, as the other two applications. Users can open (or close) a large window pane for any picture by clicking on it, in which case its loading task receives an even greater priority boost by the association with multiple visible window panes. This boost is rescinded whenever the these windows are iconified, and therefore not visible. The presence of the mouse grants an additional priority boost to the window. Furthermore, users can manually adjust the priorities of individual photos, e.g., to have one downloaded at high priority even when it is not visible. Because summation is used and the interface allows manually adjusted priorities to go negative, the user can effectively reduce the priority of a photo below the base priority. Again, these are application policies, which are implemented with just over a dozen Tcl/Tk event bindings, as opposed to being specific to Petra-Flow. Finally, like the other applications, Petra-Flow provides for asynchronous operation and automatic

---

[2] Tasks use the prioritized TCP stream class provided with the Petra-Flow library to link the task's dynamic priority to the network connection transparently. Network priorities are not currently supported by the operating system, but by our test harness software, described in the following section. While network connection priorities are mainly intended to address network scheduling in bandwidth-scarce situations, they could conceptually be communicated also to the HTTP server to influence its scheduling.

cancellation of obsolete tasks.

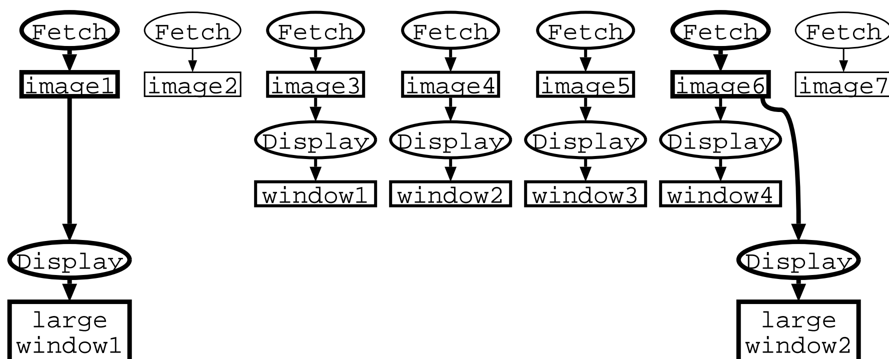

Figure 6.8: *Example Dependence Graph for Photo Album Browser.*

A hypothetical dependence graph for this program is depicted in Figure 6.8. It shows four display tasks corresponding to a strip of four small images panes, plus two more display tasks for large window panes. The user has scrolled to the right by two images; note that the following image, `image7`, is being prefetched. Priority flows up from the window variables, granting highest priority to demand fetches for large windows, intermediate priority to demand fetches only visible on the strip, and low priority to prefetching tasks. When the user last scrolled to the right, the second image stopped being visible, but it continues to be fetched at a lower priority.

The pseudo-code in Figure 6.9 approximates the structure of the `pAlbum` program. (To simplify the exposition, the image strip window panes and the large "blow-up" windows have been combined into a single array and the task that fetches the album directory has been inlined.) When the user opens a new album, pairs of tasks are launched to fetch and display incrementally the first six images in the image strip. Then an additional fetch is launched for the following image, which is automatically performed at lower priority, since it is not initially attached to a window. When the user scrolls the strip, display tasks are established corresponding to the six images that are currently visible. Tasks to fetch the images are assigned for any of these six

```
array<versioned<image>>   images;   // album photos
array<versioned<window>>  windows;  // Tcl/Tk window panes. The first 6 are
// in a small image strip; others are large and instantiated as needed.

// fetch an image (incrementally)
task write_multi image Fetch_Image(string URL);

// display image in Tcl/Tk window pane number i
task write_multi window Display(read_skip image in, int i);

main()
{
   for (i = 1..6) windows[i].SetPriority(1);

   user event loop {
     case window i gets mouse or de-iconified:  windows[i].AdjustPriority(+1);
     case window i loses mouse or iconified:    windows[i].AdjustPriority(-1);

     case new album URL:
                clear images and windows
                get album (list of URLs) from Web server
                for (i = 1..6) {
                     images[i] = Fetch_Image(URL[i]);
                     windows[i] = Display(images[i]);
                }
                images[7] = Fetch_Image(URL[7]);            // prefetch

     case scroll to position p:
                for (i = p..p+6) {
                     if (images[i] hasn't been assigned yet)
                         images[i] = Fetch_Image(URL[i]);
                     windows[i-p+1] = Display(images[i]);
                }
                if (images[p+7] hasn't been assigned yet)     // prefetch
                    images[p+7] = Fetch_Image(URL[p+7]);

     case open a large window for an image i visible in the strip:
                make new Tcl/Tk window w
                windows[w].SetPriority(2);
                windows[w] = Display(images[i]);

     case close large window i:
                windows[i].Free();
   }
}
```

Figure 6.9: *Program Structure of* `pAlbum`*.*

images (or the following one) that has not yet had a fetching task assigned to it, such as can happen when scrolling to the right five positions in one step.

Image Format

Unfortunately, we found it in our best interest to develop a new data format for progressively encoded, color-mapped images. Existing formats and re-usable software libraries either (1) only permit a few levels of incremental quality, (2) are not thread-safe, or (3) require colors that are not in the original color-map when merging pixels for lower quality versions, which explodes the number of colors needed in the display color-map [19]. Our format consists of the color-map followed by progressively finer sub-samplings of the image until all pixels have been encoded. No pixel is encoded redundantly— higher resolution passes exclude pixels previously included.

For a fair comparison against standard formats, which nearly all involve compression, our format is compressed as well. Again, we were unable to find a reusable compression library that is thread-safe, so we implemented our own from scratch. Our method of compression employs dynamic frequency sorting of encoding-adjacent pixels using the "move-to-top" rule and a static Huffman encoding of the resulting indices to the frequency table [53]. We do not encode redundant sequences of pixel values specially, as the common LZW compression scheme [103] does for GIF encoding [16]. Even so, our compression typically beats GIF compression for photographic images. Comparing the two on a sample of eighteen 640×480 photographs, our format is 7% smaller in the median (1% smaller on average). This is remarkable, since our reordering of pixels for progressive encoding suffers a loss in spatial locality; GIF format grows by 6% in the median (8% on average) in experiments where we forced it to use our reordering of pixels.

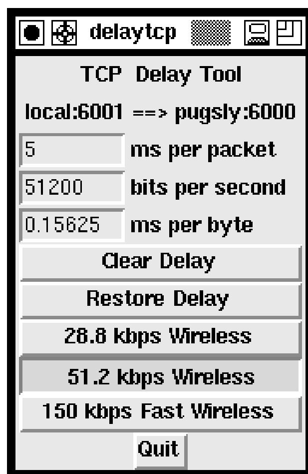

Figure 6.10: *Snapshot of the Testing Tool for Prioritized Network Traffic.*

6.1.4 Testing Tool for Prioritized Network Traffic

The photo album Web browser relies on Petra-Flow for the intelligent allocation of (scarce) network bandwidth among multiple TCP connections, each incrementally downloading a different image. Petra-Flow performs resource allocation by controlling priorities, leaving the job of prioritized scheduling to the operating system. Unfortunately, today's operating systems have little support for prioritized network communication (although there is plenty of work in this direction [2, 9, 48, 72]).

For the short term, we built a tool that runs as a separate process to give the effect of prioritized network traffic and to let us adjust the simulated network conditions dynamically. TCP connections that are made through this tool have their traffic delayed artificially as if they shared a simulated network, such as a wireless cell. The graphical user interface of the tool is shown in Figure 6.10. One can set latency and bandwidth explicitly with the numerical fields, or select known configurations with the push-buttons— the depicted setting is for the effective performance we measured for Proxim RangeLAN2. Furthermore, one can dynamically select the priority scheduling policy. Among the choices are strict priority scheduling, priority-

weighted timesharing, and lottery scheduling [98].

We have used this tool also with the database application to make the network connection the critical resource. This has been useful to control the rate of database service artificially, and to simulate the conditions of using the databases over cellular telephony, such as CDPD [37].

6.2 Static Analysis

Having discussed the applications we built using Petra-Flow's support for responsiveness, we present several static analyses. Specifically, a rough measure of coding effort for each, a comparative survey against other programs available by anonymous FTP, and an analysis of storage overhead.

6.2.1 Coding Effort

First, we evaluate the effort to produce these applications with Petra-Flow. It is difficult to measure coding effort precisely, but as some rough estimate, Table 6.1 lists for each application the number of lines of code[3], the number of semicolons, and the approximate calendar time that elapsed during the implementation according to the revision control logs (Unfortunately, the number of days of actual programming time was not measured.). The numbers in parentheses indicate how many extra lines of code are generated by the preprocessor and the number of asynchronous tasks, respectively.

The application sources grow by 10% to 30% with the preprocessor, but in every case they are significantly less than the Petra-Flow library itself, which amounts to 1376 lines of code (1242 semicolons). Presuming that the Petra-Flow prototype was

---

[3] In this chapter, we exclude comments and blank lines from code size measurements to normalize for different styles and degrees of documentation. This helps facilitate comparison across programs written by different authors in the following subsection.

Table 6.1: *Coding Effort of Petra-Flow Applications.*

Application	Code Lines	Semicolons	Elapsed Time
pDatabase	676 (+247/4)	474	10 days. . . 2 months
pFractals	615 (+222/3)	341	15 days
pAlbum	700 (+173/3)	420	1 month

not written in a particularly inefficient way, we conclude that Petra-Flow contributes a substantial amount of code to each of these applications, some significant portion of which would otherwise need to be re-designed and re-written for each application. Since Petra-Flow itself is reusable, the effort to construct it is amortized across applications built with it.

The time that it took to build the `pDatabase` application is less precise than the others. There is an entry in the revision log at ten days that indicates the application was limping along well, and one other at two months with the fully functional version. The elapsed implementation time was prolonged by the completion and debugging of the Petra-Flow prototype (this being the first significant application built with it) and by an arcane license and communication interface to the back-end databases. Likewise, the implementation of `pAlbum` was prolonged by having to build support for network priorities.

6.2.2 Comparative Survey of Other Programs

How can we judge whether the coding effort for these programs is great or little given the application? For the evaluation of `pFractal`, at least, there are a number of other interactive Mandelbrot fractal explorer programs written without Petra-Flow that are available by anonymous FTP. Table 6.2 compares `pFractal` with all such programs we were able to obtain.[4] The table is sorted by the number of lines of

[4] Many of these are available at `ftp://spanky.triumf.ca/fractals/programs/unix/`.

Table 6.2: *Comparison of Interactive Mandelbrot Fractal Generator Programs.*

#	Program/Author	Code Lines	Semicolons	Interruptible?	Incremental?
1	fractal/Young	133	78	no	no
2	xmandel/Freeman	576	463	no	no
3	**pFractals/Forman**[1]	615	341	tacit	2-D
4	xmandel/Zsoldos	635	346	tacit	no
5	xmandel/UC Santa Cruz	1052	514	manual	no
6	mandelbrot/Anderson	1064	645	tacit[2]	2-D
7	xfractal/Delabre	1142	674	tacit[2]	2-D
8	fun_factory/Burgess	1479	454	no	no[3]
9	gnumandel/Swiston[4]	1700	1068	tacit	no
10	xfracky/Jensen[1,5]	2000		tacit[2]	no
11	xmb/Helminen[4]	2304	1411	no	no
12	net-fract/Johansson[4]	2523	1313	no	no
13	mxp/Brady	2835	2106	tacit[2]	no
14	xfexplorer/Guerin	6216	4295	manual[2]	no or 2-D
15	xmfract/House	56477	34952	manual	1-D or 2-D
16	xfractint/Shirriff	61341	34824	tacit	1-D or 2-D

[1] These programs are written in a combination of C++ and Tcl/Tk, whereas the others are in C.

[2] Although these programs are interruptible, they poll at a granularity coarser than the other programs by over two orders of magnitude.

[3] The “Fractal Fun Factory” program lets the user select what resolution granularity to use while dragging a fractal with the mouse or after “dropping” it. In this way, the user explicitly controls the trade-off between quality and responsiveness.

[4] These three programs employ distributed multiprocessing to compute fractals faster.

[5] The size of program #10 was estimated by its author, as its source code is not publicly available.

code, as evaluated by feeding the source code to the software metric program `csize`.[5] The last two columns indicate whether certain features that improve responsiveness at the user interface are present— specifically, whether one can interrupt an active fractal computation, and whether multi-resolution techniques are used to present the fractal incrementally. (The label "tacit" means that fractal computations are canceled tacitly when a new user input makes the computation obsolete, as opposed to having to manually cancel the computation before entering new inputs.) Nearly all the programs that did not support incremental resolution at least presented the fractal in row-major order while it was being computed.

Note that `pFractals` is one of the smallest programs listed, almost half the size of the next smallest program that provides for multi-resolution display. This evidence should not be given too much weight, however. There are a few caveats to consider, the principal one being that no other program implements the same set of features as `pFractals`. While several have significant features that we left off (e.g., multiple types of fractals, 3-D views, image saving, and boundary finding methods to speed calculation), our program has features not found in the others (e.g., generation of multiple fractals simultaneously, priorities, and no polling overhead). One other program (#12) does provides for multiple fractal windows, but it does not generate them simultaneously and is completely unresponsive during a computation.

The other significant caveat is that these common measures do not necessarily reflect coding effort. Different authors may vary in their ability to express programs efficiently. Moreover, the use of C++ and Tcl/Tk, as opposed to raw C, may allow `pFractals` and program #10 to be more expressive with fewer lines of code. (This is exactly the purpose behind Petra-Flow.)

Nonetheless, this survey does lend credibility to the statement that `pFractals` was comparatively easy to produce, even with its advanced features for improving

---

[5] `ftp://ftp.sterling.com/usenet/comp.sources.reviewed/volume04/csize`

user responsiveness and for producing fractals simultaneously in separate windows with multi-threading. If one were to rewrite our applications without using Petra-Flow but preserving the responsiveness features, one would need to reproduce the global view for resource allocation and cancellation of outstanding tasks. This means re-implementing a significant portion of Petra-Flow in an application-specific setting. This effort must be repeated for each program written without Petra-Flow, whereas the effort to build Petra-Flow is amortized by re-use.

As a byproduct, this study produces some support for Petra-Flow's approach to attaining asynchronous operation of the user interface. Among the programs that allow interruption, sometimes their authors forgot to poll for new user input in certain long-running auxiliary operations, such as flipping the image end for end in programs #15 and #16. Some programs poll at a much coarser granularity than others. While this reduces polling overhead, it also leads to greater latency in responding to new user input, especially when the maximum number of Mandelbrot iterations is high. It is difficult for an author to manage this trade-off by static sizing, as discussed in the introduction to Chapter 3.

Three of the programs (#14, 15, & 16) have a command to continue an interrupted fractal computation. This provides a less than elegant way for one to use other operations of a program in the midst of computing a fractal. Full asynchronous use of the interface is superior.

We can evaluate the runtime cost of polling and multi-resolution incremental display by using a feature of program #14 that allows these options to be selectively disabled. We repeatedly measured the elapsed time to generate the canonical Mandelbrot fractal ($-1.5 - i$ to $0.5 + i$) at 640×512 resolution under different combinations of options. We found that the multi-resolution feature added 3–13% overhead and polling added 25–40% overhead, depending on whether the other feature was enabled. Although multi-resolution display delays the final result, "reasonable quality" was visible in just a tenth of the total time. By increasing the maximum

Table 6.3: *Storage Sizes.*

Object	Size (bytes)
versioned variable	25 (+50)
versioned value	145 (+175)
asynchronous task invocation	230 (+50)
integer	4
pointer	8
Tcl variable (overhead only)	40+
thread stack	21,000+
database record	1000–2000 (typical)
640×480 color-mapped image	308,000
(compressed)	192,000 (on average)

number of Mandelbrot iterations from 250 to 2500, we brought the overhead for polling down to 6% and completely eliminated the measurable overhead for multi-resolution. It is important to understand that under these conditions incremental display is even more valuable, since the overall elapsed time is much longer, but the time until an image of "reasonable quality" appears is only somewhat longer. Though the delay may be proportional, the time scales are different.

These analyses and our qualitative experiences with the applications mentioned in this chapter have confirmed for us that incremental display is a valuable and worthwhile technique, and that polling can have significant overhead, programming complexity and pitfalls.

6.2.3 Storage Overhead

The use of Petra-Flow abstractions incurs storage overhead for the runtime dependence graph. Table 6.3 shows these overheads in comparison to the size of several application-independent and application-dependent objects. The numbers in parentheses indicate how many extra bytes are added for debugging information, which could potentially be omitted for production compilations.

The overheads are significant[6] compared to raw C++ data types and therefore should not be applied to every program variable. On the other hand, for an interpreted language such as Tcl, where the overhead for each variable exceeds 40 bytes, it might be reasonable to provide Petra-Flow capabilities for any variable. Observe that these overheads are much smaller than application objects or thread stacks (which Petra-Flow conserves by keeping threads only for executing tasks).

This leads us to evaluate the extra storage requirements for incremental quality versioning, which is not specific to Petra-Flow itself, but only to the general technique. Consider our image domain where each quality version contains twice the number of pixels in each dimension as the previous version. In the worst case, where all quality versions coexist simultaneously, the total space consumed is characterized by the series:

$$1 + \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \ldots \quad = \quad \sum_{i=0}^{n} \frac{1}{4^i} \quad \leq \quad \frac{1}{1 - \frac{1}{4}} \quad = \quad 1.\overline{33}$$

where one unit is equal to the space for the full quality version. That is, 33% overhead in the worst case. If old quality versions can be deallocated when higher versions become available, at most two versions need to exist simultaneously, i.e., 25% overhead.

No extra space is needed in some applications, such as those that only append to a data structure to improve its quality, as in the database application. In the other two applications, as well, is it possible to reuse the full quality image space for all sub-versions, if only we promise not to change the subset of pixels that have been submitted for earlier versions. This optimization is not admissible for all image-domain applications and was not implemented in our applications.

Finally, there is the issue of how much memory is used for multiple versioned values of a single program variable, i.e., variable splitting as opposed to quality versioning.

---

[6] The size of a pointer on the DEC Alpha architecture is twice that of most architectures. This enlarges dependence graph nodes significantly, since they are pointer-rich.

This cannot be determined statically, as it depends on the specific application and on the degree to which users actually outpace resources and cause variables to be split into multiple values.

6.3 Runtime Analysis

In this section we evaluate Petra-Flow from a dynamic perspective. While we are able to measure the runtime *costs* of the Petra-Flow framework (that is, of its prototype implementation), it does not make sense to try to quantify its runtime *benefits*. The benefits vary tremendously depending on the situation and are not due to Petra-Flow itself, but to the underlying techniques, such as prioritization and incremental quality. The benefit of Petra-Flow is in facilitating the programming and execution of those techniques.

The first two subsections evaluate overhead costs via macro- and micro- measurements, respectively. The third subsection evaluates the individual techniques that Petra-Flow incorporates, including incremental quality results, prioritization, and cancellation. We focus on situations where the resource demands exceed the supply, since good responsiveness is not a challenge when resources are plentiful.

Before turning to the quantitative discussions, let us first say that qualitatively we and a half dozen others who have run our applications have indeed appreciated the responsiveness enhancements facilitated by Petra-Flow. Further, control over the resource allocation mechanism, a new component to the user interface, comes naturally to people. With little explanation, people are quickly able to use it to their benefit.

6.3.1 Macroscopic Runtime Overhead

We can estimate macroscopically an upper bound on Petra-Flow's overhead by measuring the proportion of time spent in Petra-Flow code, as opposed to application-specific code, while a user is actively driving the interface. We found this proportion relatively insensitive to the specific user actions taken. More important was the overall level of activity.

Unfortunately, we could not get a code profiler to function in our multi-threaded

environment, so instead we measured the proportion of time the dependence graph lock is held. Nearly every public interface to Petra-Flow begins by acquiring this lock and ends by releasing it, so our measurement is a near approximation to the actual time spent in Petra-Flow code. We instrumented the macros that manipulate this lock to measure and accumulate the elapsed locking time using the DEC Alpha instruction cycle counter (read with the assembly instruction "`rpcc $0`"). We sampled at ten second intervals via a signal handler while a user drove the interface at a rapid but reasonable pace for skimming. We calculate the overhead as a percentage of real time (i.e., overhead to the processor), and also as a percentage of the application's process time; these are nearly the same for a compute-bound application. We repeated this experiment to try to drive up the overhead as high as possible for an upper bound by having the user trigger Petra-Flow features at an extreme pace, e.g., wiggling the mouse rapidly among windows to twiddle dependence graph priorities. In the case of `pFractals`, we reduced the image size by a factor of 75 to amplify further the proportion of overhead. The results of this study are presented in Table 6.4.

Table 6.4: *Macroscopic Measurement of Overhead (as percentage).*

Application:	pDatabase		pFractals		pAlbum	
Time relative to:	real	process	real	process	real	process
Reasonable Use:						
median	0.1%	3.2	0.1	0.1	1.6	1.6
average	0.1	3.6	0.2	0.2	3.0	3.5
Extreme Use:						
median	0.5	7.7	1.9	3.3	2.2	2.2
average	0.5	7.4	2.4	3.6	8.7	9.0

The overheads are quite low, incurring less than one or a few percent typically, and even under unnaturally heavy usage, they continue to be reasonably low, under 10% in the worst case. (If these figures were substantially larger, it would be prudent to reduce the potential for contention by replacing the single lock for the dependence

graph with a more sophisticated mechanism based on graph partitioning [102, 106].) Note that for an input/output-bound application, such as `pDatabase`, the overheads with respect to real time vs. process time have a significant disparity. Under extreme use of Petra-Flow features, we see this discrepancy grow for `pFractals`, because we reduced the computation by a factor of 75, making it an input/output-bound process.

6.3.2 Micro-Benchmark Comparison with Raw Pthreads

Here we present a micro-benchmark evaluation of individual Petra-Flow operations compared against the cost of performing similar actions with the raw Pthreads interface. These measurements were made on an unloaded DEC3000/400 (Alpha) computer using 100 repetitions of each test. The results are shown in Table 6.5 below.

The first grouping measures the time to launch a thread. In the case of Petra-Flow, we measure the elapsed time while installing a new task invocation in the dependence

Table 6.5: *Micro-Benchmark Comparison to Raw Pthreads (microseconds).*

Test	Median	Average	CV(%)
1. Pthread thread startup	1290	1300	5
2. same, but with pthread_detach()	790	800	10
3. P-F task startup	1020	1040	6
4. Pthread thread startup & join	1480	1500	6
5. same, but with pthread_detach()	908	914	7
6. P-F task startup, submit, & start consumer	2700	2700	25
7. P-F submit & start consumer	810	840	19
8. Pthread set priority, sleeping thread	56	57	6
9. P-F set priority, sleeping thread	83	85	6
10. P-F set priority, no thread	18	18	3
11. P-F set priority of chain w/o threads (per task)	7.6	7.7	3
12. Pthread cancel, sleeping thread	62	64	11
13. P-F cancel, sleeping thread	125	125	11
14. P-F cancel, no thread	284	290	17

graph and starting a thread for it. There are two measurements listed for Pthreads. The later achieves nearly 40% savings by indicating via `pthread_detach()` that a retired thread may be recycled. To prevent dangling references, programmers can only recycle threads when they are certain that no future thread will want to collect its result via `pthread_join()`. In Petra-Flow, because data synchronization is done with the dependence graph and not by joining, we detach all threads, allowing for maximal re-use.

The second group of tests measures the time to launch a thread and get its result. Again, we see nearly 40% savings if retired threads are detached. In this simple test, we have a single join per thread, so it is easy to determine when threads may be detached; in practice, it is not always so easy. The Petra-Flow version of this test (#6) includes two full task startups (#3). The last test in this group measures the time to submit a result and start up the downstream consumer thread; both the producer and consumer tasks were created before the timer started.

The fourth grouping is for priority adjustments. Changing the priority of a Petra-Flow task only incurs the 56 microsecond Pthread system call overhead if there currently exists a thread for the task. Tasks that are waiting for new inputs to execute have no thread and therefore less overhead when changing priority. This difference is illustrated between tests #9 and #10, the latter executing nearly five times faster for having no thread. In test #11 we repeatedly adjust the priority of a chain of 100 tasks, none of them having threads. In this case, the call overhead per task is halved.

The final tests measure the overhead of cancellation. If there is no thread behind a Petra-Flow task at the time of cancellation, we remove the task from the dependence graph and check for newly uncovered opportunities for cancellation. But if there is a thread, we simply signal for its cancellation and defer removing the task node until the thread terminates.

6.3.3 Evaluation of Techniques Supported by Petra-Flow

Again, although it does not make sense to try to measure the runtime benefits of Petra-Flow itself, we can reconfirm the known usefulness of the techniques it employs, such as incremental quality, prioritization, and cancellation. Even so, the benefits depend greatly on the specific situation, e.g., the benefit from granting high priority to a thread depends on the scheduling policy, the number of other threads competing for service, and their relative priorities. In this section we briefly demonstrate the trade-offs of the responsiveness-enhancing features in the fractal explorer application.

Incremental Quality

First, we address the technique of producing and displaying results in an incremental fashion. Recall that for fractal program #14 of the survey in Section 6.2.2 we measured a 3–13% overhead for displaying a fractal incrementally by refining its resolution vs. displaying it in one pass in row-major order. We did not quantify the benefits, however. We do that here by instrumenting `pFractals`.

Qualitatively, one experiences a clear view of the whole fractal much earlier with multi-resolution display. It is awkward to determine this quantitatively, however, since there is no well defined moment in time when the quality becomes "adequate" for the user. In lieu of defining such a point artificially, we measure and present each point of improved quality.

Moreover, matters are complicated by the fact that quality is a subjective issue that depends on the content of the image and its intended use. Rather than measuring subjective quality with human subjects, we use the logarithm of the image resolution, since user studies in the human factors literature have determined that subjective image quality is roughly proportional to the logarithm of the image resolution (improving less steeply at high resolutions) [11, 10].

The quality vs. response time graph in Figure 6.11 plots a typical set of `pFractal`

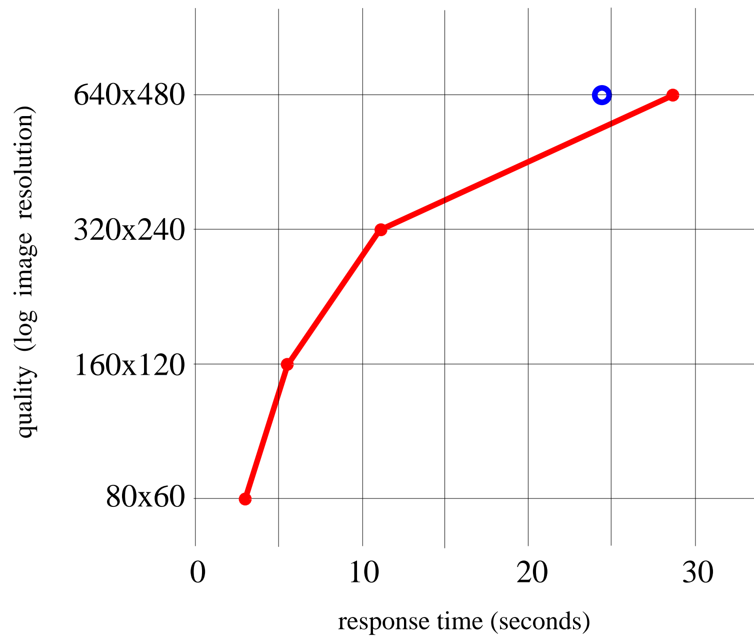

Figure 6.11: *Incremental Quality Results: Quality vs. Response Time.*

incremental responses. The additional hollow point at 24.5 seconds marks the response time with the incremental quality feature is disabled— the baseline for comparison. The incremental results were produced in 12%, 22%, 45%, and 117% of the non-incremental time, respectively, and more importantly, on a time scale that matters to real users, i.e., seconds as opposed to microseconds. In cost-benefit terms, the initial quality version is produced in an order of magnitude less time than the non-incremental result, while the final quality version takes 17% longer to produce. If one deems 17% overhead unacceptable, the cost can be reduced somewhat by using fewer multi-resolution passes.

Prioritization

To assess the effectiveness of Petra-Flow's resource allocation mechanism in trading off the response time of some activities for others, we generated two identical fractals simultaneously under various priority policies and measured the response time at each

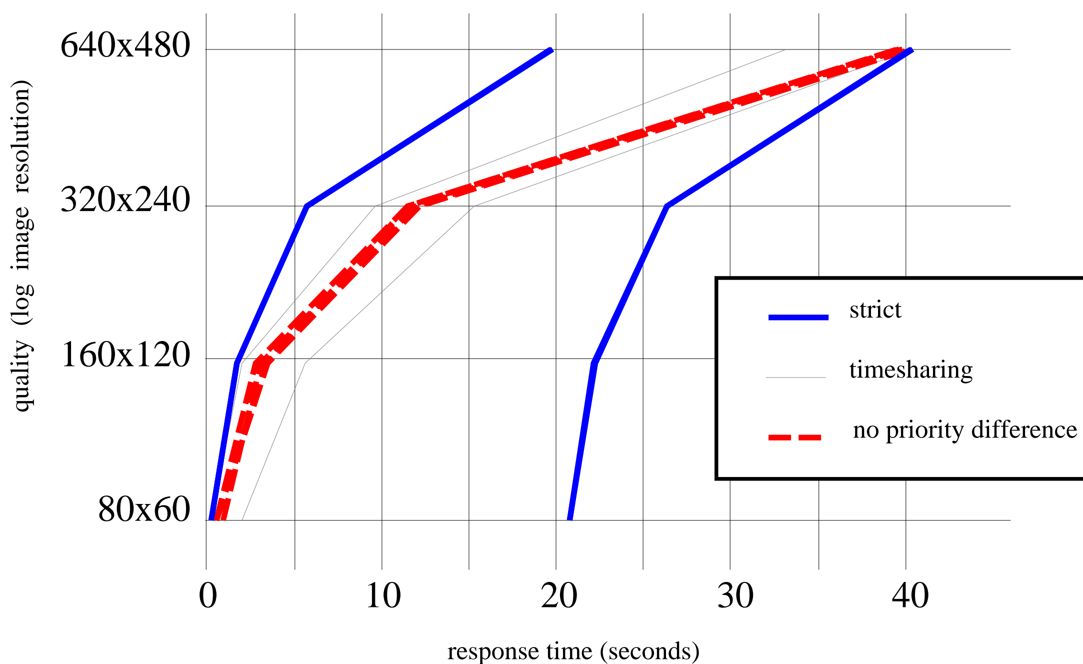

Figure 6.12: *Prioritization: Quality vs. Response Time.*

incremental resolution. The results are shown in the quality vs. response time graph in Figure 6.12.

Under a policy of strict priority scheduling, the favored fractal completes before the other even begins. Under priority-weighted timesharing, the favored fractal achieves better response time at each quality level than the disfavored one (20–90% faster), but is generally slower than under strict priority scheduling (by about 65% at the highest quality level). Finally, as a baseline, we performed the experiment with no priority difference. In this case, the response times of the two fractals are nearly indistinguishable. Relative to this baseline, the favored fractal is generated twice as fast under strict priority scheduling, and about 16% faster under the timesharing policy.

The level of processor service depends on the specific priority values under the timesharing policy. More specifically, it is proportional to the difference between the

priorities of the two fractal tasks (their absolute values are irrelevant). For the results presented so far, the priority difference was four (the maximum allowed without privilege by the operating system). In experiments where we tried different priority values, the two curves grow closer to the baseline as the priority difference decreases. We would like future operating systems to permit much greater flexibility so that we can cover the territory between the strict priority curves and the timesharing curves.

Observe that the final completion time for the latter fractal is the same under all policies, i.e., prioritization has no penalty on the completion of the unfavored computation, because the total amount of work is unchanged. Only by measuring the response time of the intermediate quality results can we detect any cost for prioritization. This illustrates nicely that prioritization does not affect total throughput, but merely trades off the response time of the intermediate quality results of the unfavored computations for improved response time for the favored computations at all quality levels.

Cancellation

Next, we demonstrate quantitatively the benefit of Petra-Flow's automatic cancellation mechanism. For this test we start two fractal computations in rapid succession, the second obsoleting the first, and measure the response time at each quality level of the second task. (The Petra-Flow priority mechanism was not used in these experiments to avoid compounding benefits from different features. Normally, the obsolete generator task would also be reduced in priority.)

As a baseline for comparison, we repeated the experiment without the benefit of cancellation, i.e., the measured task competes for service with an identical task that is obsolete but not canceled. And to compare against a hypothetically optimal case, we timed a single fractal computation with no competitor.

Additionally, we measured several variants of the program that poll to check for cancellation, instead of accepting cancel signals asynchronously. We repeated the

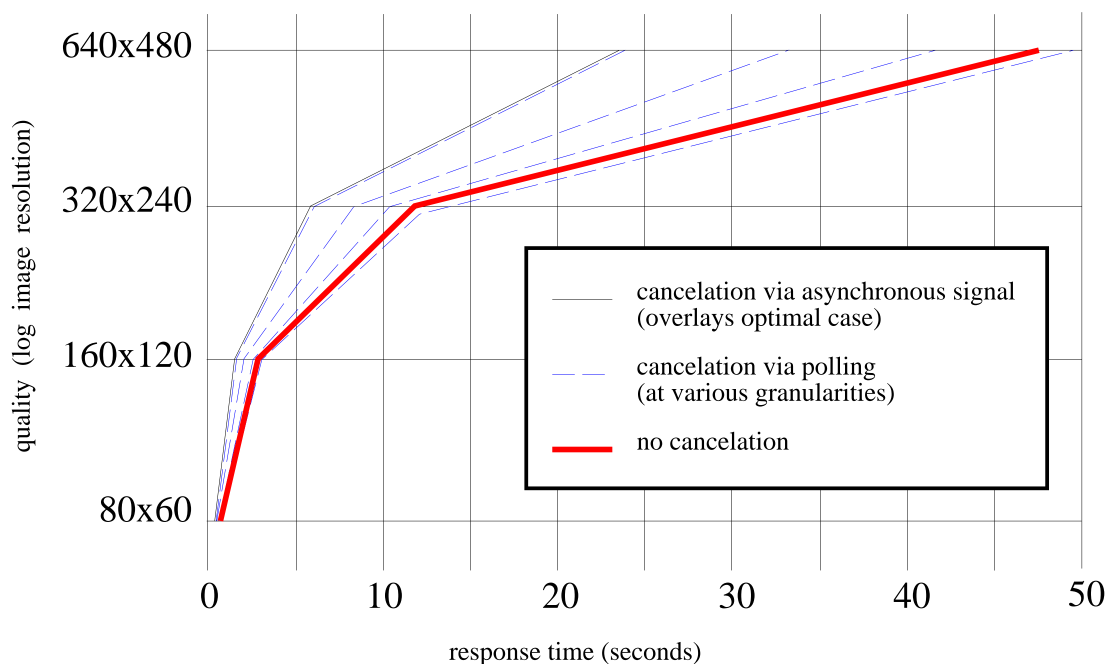

Figure 6.13: *Cancellation: Quality vs. Response Time.*

experiment with the polling call placed at different points in the code, effectively varying the polling granularity and hence its overhead.

The results of these experiments are given in Figure 6.13, which shows the response time for each quality version. The four curves that represent the polling experiments are at the following granularities, from left (fastest) to right (slowest): every row of the image, every pixel, every 100 Mandelbrot iterations beginning with the hundredth (of 256 maximum in this experiment)[7], and every 100 Mandelbrot iterations beginning with the first.

Asynchronous cancellation achieves the ideal speedup factor of two over the baseline at each quality version, and comes within a tenth of a percent of the optimal

---

[7] Mandelbrot points that iterate fewer than 100 times result in no polling, hence, this technique is dependent on the Mandelbrot region generated and will not poll at all in some regions of Mandelbrot space. For this test, we generated the canonical Mandelbrot set.

case, hence only one curve is visible for the two experiments.

The experiments with polling, however, do not benefit as much from canceling the obsolete task, because of the additional overhead for polling. The benefit depends strongly on the polling granularity: at the coarsest, the final quality version exceeds the optimal case by only 2%; at the finest, polling costs exceeds the benefit of canceling the obsolete task.

Polling at a coarse granularity has its downside, as well: increased latency until the obsolete task detects that it should cancel. To explore this trade-off, we ran the experiment again, varying the polling granularity from every pixel to every 153600 pixels, i.e., 50% of the entire image. (Incremental quality was disabled for this experiment.) For comparison against an optimal baseline, we ran the experiment again with no competing task and once more with asynchronous cancellation. The latter achieved optimal performance in this test, as well, and so we will discuss it no further.

Figure 6.14 plots the results of these experiments as the percentage slowdown over the optimal case vs. polling granularity on a log scale. For each data point we use the median of five repetitions of the experiment to smooth out noise. To give more meaningful labels, a portion of the horizontal axis is labeled as the percentage of the image, rather than the specific number of pixels.

At fine granularity, the polling overhead slows the computation by as much as 30%. (Recall that polling overhead causes a 25–40% slowdown for fractal program #14 of the survey in Section 6.2.2.) Keep in mind that this overhead is paid even in situations where there are no obsolete tasks to cancel.

At coarse granularity, the task pays an opportunity cost in competing for a longer duration with the obsolete task, which is polling at the same granularity. In the extreme, if the obsolete task only polls once when it is half finished with the image, the first half of the useful task takes twice as long, resulting in 50% slowdown.

Such coarse polling is a realistic possibility for applications that do not have one

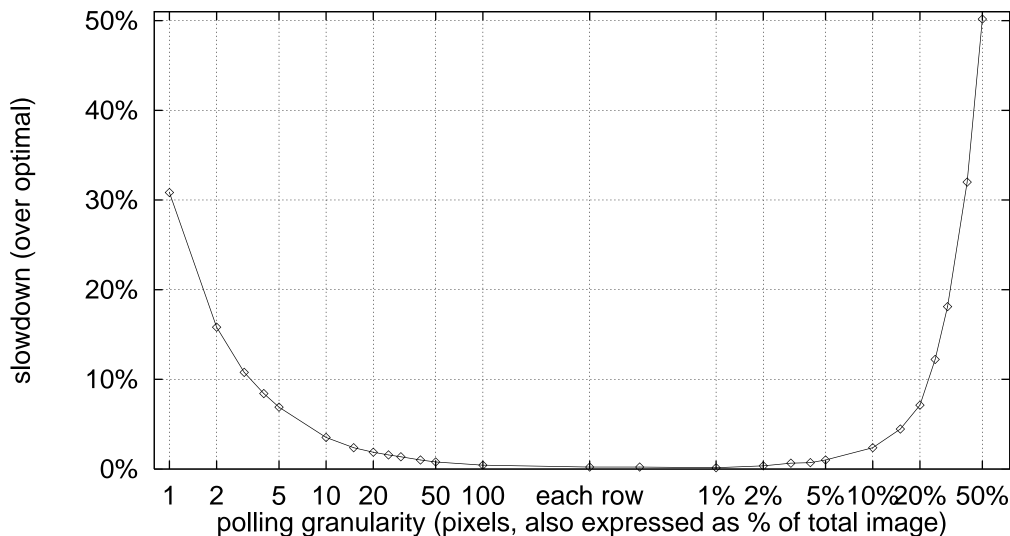

Figure 6.14: *Slowdown vs. Polling Granularity.*

or a few reasonably fine grain loops in which to poll, as we have for the fractal explorer application. For a large body of code with many subroutines, it can be difficult to estimate where polling tests are needed. A long-running routine may be overlooked, or one may not be able to inject polling tests for library routines. (In fairness, neither are most library routines today equipped to handle exceptions from asynchronous cancellation, so neither cancellation mechanism would be active while in the library.) For any of these reasons, there may be a significant delay before the task recognizes that it should terminate early.

We conclude that Petra-Flow's asynchronous cancellation facility is the mechanism of choice. It avoids the overhead risk of polling too often and the latency risk of polling too seldom. The latency risk concerns both timely cancellation of obsolete tasks and, in applications that poll for new user input instead of having a separate thread for it, good user responsiveness. Finally, polling complicates programming. The source code must be sprinkled with polling tests, which are unrelated to the computation at hand. To achieve good polling granularity, the programmer must estimate

the running time of portions of the computation. Recall that variable environments thwart the "static sizing" approach.

6.4 Simulation Analysis of Priority-Mediated Locks

The priority-mediated lock was designed to reduce priority inversion in circumstances where the lock holding time is significant relative to the time between lock requests, such as during periods of scarce resources. In order to verify that it meets its goal, we developed a simulation to measure its trade-offs quantitatively in a controlled environment. For comparison, we also simulated the traditional lock, and the priority-queued lock. The latter improves on the traditional lock by using a priority queue when selecting which blocked thread to resume.

We divide this presentation into three parts: workload and parameterization, statistics and stopping criteria, and results.

6.4.1 Workload and Parameterization

The workload for the simulation is derived and parameterized from the `pDatabase` application, though somewhat simplified. It takes the form of one (or more) low priority tasks (simulation customers) that prefetch records in a loop, and a high priority customer that demand fetches records in a loop that includes a "thinking" delay to allow time for the hypothetical user to skim the record before demanding the next.

To simplify the workload characterization, we suppose that the record demanded is never cached, i.e., the prefetching task is doing a terrible job of predicting the user's access pattern. Further, we assume that there is no contention besides the critical section and that the time spent in the critical section is not dependent on other tasks in the system. These assumptions factor out any compounding effects of priority scheduling of the processor. They are also consistent with input/output-bound tasks,

or running the program on a computer that has enough processors to run all active tasks simultaneously.

Just as in the real application, fetching a record is done incrementally in three installments, each of which requires mutually exclusive access to the back-end database followed by a relatively short period outside the critical section to store the results. In addition, there is a preparatory computation that happens before the first of three accesses to the database.

The high priority task states its interest in the priority-mediated lock sometime during this preparatory period.[8] In the experiments that follow, we vary this point and parameterize it as the percentage of preparation time during which the interest is registered in advance of need, e.g., "100% in advance" means that we state our interest in the priority-mediated lock as the first step of the preparatory computation. The registration of interest is removed immediately after relinquishing the lock for the final quality installment. This parameter has no effect when simulating the other locking policies for the critical section.

We exercised the `pDatabase` application with prefetching disabled (i.e., a single thread of execution) in order to measure the service time distribution for the back-end database and the processing time outside the critical section where there is no lock contention. In the simulation, we generated random service times according to these measured distributions. For the think time of the hypothetical user, we used a negative exponential distribution with a mean of a half second. We experimented with other values from one quarter of a second to two seconds, verifying that the relative shape of the results is unaffected by this parameter. (We note that the database literature often uses the negative exponential distribution for human think times between transactions [83, 94].)

---

[8] So does the *low* priority task, but it has no effect, since it has the minimum priority.

6.4.2 Statistics and Stopping Criteria

The statistics we collected for the simulation runs include the percentage of time there was priority inversion (i.e., the customer holding the lock had strictly lower priority than a customer blocking for the lock) and the percentage of time that the lock was free but unclaimable by a blocking thread due to a reservation of higher priority. We will call the latter statistic the "percent waste."

Note that we have a steady-state simulation, as opposed to one that runs to completion for a fixed number of cycles. Accordingly, we chose statistics that are independent of the length of simulated time we measure. In collecting statistics, we used the "batch means" method [51, page 296], terminating the simulation when the confidence interval for each statistic was less than 1% of its mean.

6.4.3 Results

We present the results of two experiments here. In the first, we vary the percentage of advance reservation to explore the trade-off between waste and priority inversion, and to compare the amount of priority inversion across locking schemes.

Figure 6.15 plots for each locking policy the percentage priority inversion vs. the percentage waste[9], and for priority-mediated locks, we vary the advance percentage from 0% to 100%.

In these simulations there were two prefetching tasks. (For greater numbers of prefetching tasks, the data looks substantially the same. With only a single prefetching task, there is no difference between the two traditional locking policies, since only one thread can be blocked on the lock at a time.)

First, notice priority inversion is greatly reduced for priority-mediated locks: by a factor of four over traditional locks and a factor of three over priority-queued

---

[9] The two traditional types of lock naturally exhibit no waste, since their policies never keep a (low priority) thread out of the critical section.

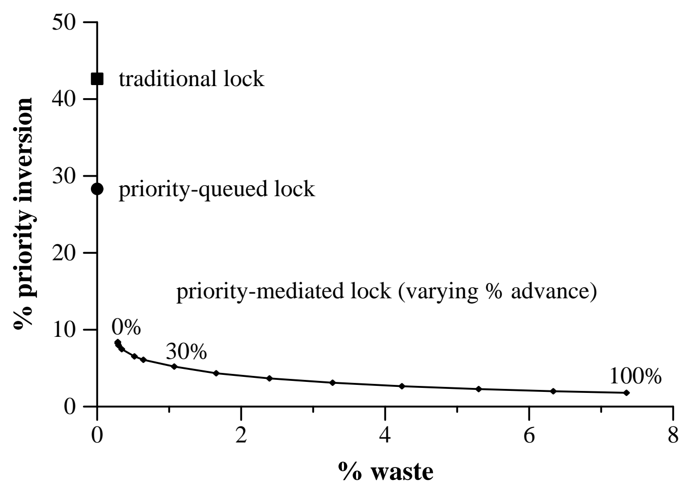

Figure 6.15: *Simulation Results: Priority Inversion vs. Waste.*

locks. Second, observe the trade-off between priority inversion and waste for priority-mediated locks. A good reduction in priority inversion is had even at 0% advance warning, because the reservation is held over three consecutive fetches from the back-end database.

In the second experiment, we held the advance percentage at 30% and varied the number of prefetch tasks that compete with the high priority demand fetch task. The graph in Figure 6.16 shows the percentage priority inversion as a function of the number of prefetch tasks (i.e., the total number of tasks is one greater). Under priority-mediated locking, the level of waste was consistently 1%. The curves converge at the origin: where there is no contention, there can be no priority inversion.

Observe that the degree of priority inversion is highly sensitive to the level of contention under traditional locks, but much less so for the other two kinds of locks. Even so, a single competitor task raises the amount of priority inversion significantly for priority-queued locks compared to priority-mediated locks.

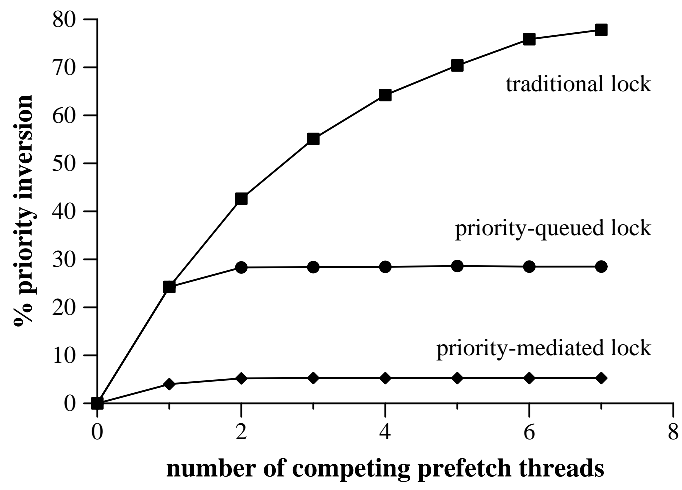

Figure 6.16: *Simulation Results: Priority Inversion vs. Competing Threads.*

Chapter 7

CONCLUSION

In this thesis we have identified a trend in modern environments toward greater service variability and performed a study of the variability and short term volatility in one such modern environment, the World Wide Web. We identified a resulting need to support programmers in building and executing applications with good user responsiveness even in such resource-variable environments. To meet this need, we proposed a new framework that provides responsiveness-enhancing services based on a global view of the interdependencies among concurrent tasks that are producing incremental results. The framework takes advantage of the fact that users can outpace resources during periods of poor service, turning the backlog of work into an opportunity for optimization. In similarity to providing implicit memory management services through language facilities (e.g., in LISP and Java), by providing services such as obsolete task cancellation and resource allocation in a reusable, application-independent manner, we can lighten the burdens of many programmers in writing and maintaining user-responsive codes and help instill greater confidence that they will exhibit responsive behavior even under scarce resources. Finally, in evaluating the framework quantitatively and qualitatively through experience with the applications we built with it, we found the trade-offs it makes for improved responsiveness reasonable and worthwhile.

In the first section below we discuss straightforward generalizations of the presentation given in this thesis. Following that, we review related work, and conclude with future work.

7.1 Generalizations

In the foregoing chapters we simplified the discussion by restricting the context in which Petra-Flow is used, namely, a single user running a single application in a resource-variable environment. Petra-Flow generalizes in several ways that broaden its applicability beyond what has been described.

First of all, the human user in non-essential, i.e., Petra-Flow is not restricted to applications that are interactive. It is advantageous also where the application driver is, say, an intelligent agent [24, 29] that may act on preliminary incremental quality results and "change its mind" about which results to focus resources on and which to discontinue interest in (cancel).

Second, Petra-Flow generalizes to applications that consist of multiple (distributed) processes that are cooperating either in a nested application model, such as OLE [61] and OpenDoc [3], or in a client-server relationship via sockets or remote procedure calls [13]. In such a scenario, the Petra-Flow graph spans multiple processes with dependence arcs along the communication channels. The implementation would require a parallel channel to communicate at the "meta" level, i.e., Petra-Flow information. This might be facilitated by an extended RPC interface that accepts supplemental operations to indicate, e.g., priority changes and obsoleteness.

Third, with the addition of such a facility, Petra-Flow could be extended to multi-user environments. The idea is that servers (central resources) are constructed with Petra-Flow and requests of the server can enjoy Petra-Flow services, including incremental quality results, cancellation, and resource allocation, as hinted at in Section 6.1.3 regarding the prioritized delivery of photo album images.[1]

---

[1] A server may only compare the priorities of concurrent requests from the same user, unless users agree to a common priority scale and are willing to dispense with equal share scheduling. Suppose a widespread application, say Netscape, employs a known priority scale for all users. This allows enhanced Web servers to deliver visible images before off-screen images for those users. People using other Web browsers that either do not provide priorities or are not calibrated to the same

Fourth, Petra-Flow's priority mechanism may be applied to applications transparently at a very coarse grain by embedding it in the operating system and display management system (e.g., the X Server or the window manager). Tasks become whole processes and priority flows along inter-process communication links starting from the window manager. Such an implementation also benefits cooperating processes as described above, but because it it has no knowledge of the internals of the applications, it cannot provide other Petra-Flow services, such as cancellation— just because a process currently has no write dependence path to the window system or file system does not mean that it never will, and so it cannot be canceled.

The uniform constraint on any Petra-Flow embodiment is that there be low latency communication with the driver of the program (human or otherwise) so that the interface can keep up. It would not be effective to use Petra-Flow in an environment where the principal source of service variability is between the application and the user's terminal— one potential model for mobile computing [50]. Figure 7.1 illustrates the gamut of mobile computing models in terms of where the system is cut by the wireless network, the chief source of variability. A well known example of the remote file system model is Coda [44]. The application partitioning model is characterized by keeping the user interface running near the user for good responsiveness and executing remotely the (core) application operations that require resources unavailable on the mobile computer, as embodied by, e.g., Wit [100, 99]. Petra-Flow is applicable to all of these models except for the mobile terminal model (unless low latency communication with the mobile terminal can be assured, as with the Berkeley InfoPad project [64, 85]).

---

scale should not have their priorities compared against those assigned by Netscape. Instead, they may receive service in proportion to the population of requests, as in the current system; within that proportion, they may be prioritized.

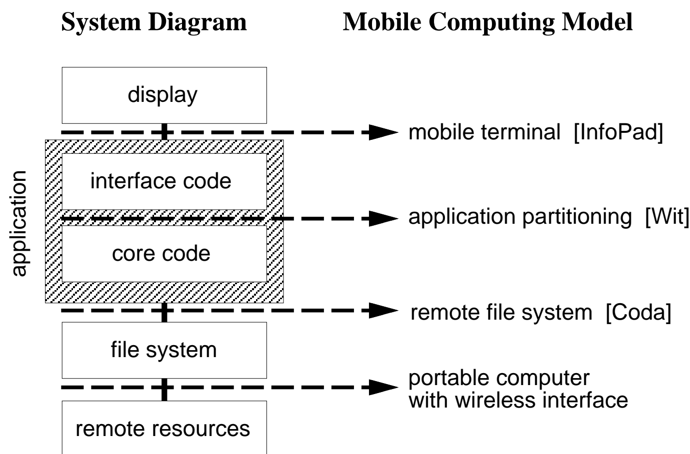

Figure 7.1: *Mobile Computing Models in Terms of Where the Wireless Network Cuts the System.*

7.2 Related Work

There are several classes of related work to consider.

7.2.1 Asynchronous Programming

Petra-Flow facilitates concurrency through its asynchronous tasks, automated synchronization for incremental quality versions, and versioned variables. The notion of asynchronous tasks was motivated by LISP futures [38], which provide a higher level interface for asynchronous execution than threads. A future is a handle to a result that is being produced by a separate thread of execution. Futures do not permit multiple quality results to be returned and require explicit synchronization to retrieve a result, whereas synchronization for consumers in Petra-Flow is implicit via their data dependencies. The LISP runtime environment does not develop a dependence graph for tasks, and hence cannot provide the level of service that Petra-Flow does.

Execution paradigms that do construct explicit dependence graphs, such as AVS [95], IBM Visualization Data Explorer [1], IRIX Explorer [86], Khoros [74],

and other data-flow programming languages [22] provide for multiple results and implicit synchronization via "restart semantics" of task code. These differ from the Petra-Flow paradigm in several important ways. They are not geared toward incremental quality results and so, for example, do not tear down parts of the graph that have achieved full quality. To program in these paradigms, programmers construct static dependence graphs explicitly, rather than having them built implicitly and dynamically via the data dependencies that transpire at runtime from traditional imperative programming.

The idea for versioned variables came from the compiler optimization technique of variable splitting [45] and is also found in register scoreboarding [70] and multiversion database systems [36, 46, 105]. Its purpose is to eliminate false dependencies: write-after-write and write-after-read conflicts. This helps expose greater concurrency. No previous work provides versioned variables at the language level, as Petra-Flow does.

7.2.2 Resource Allocation

Resource allocation in Petra-Flow is served by priority up-flow, priority-mediated locks, and cancellation of obsolete tasks. The idea of controlling resource allocation by automatically adjusting priorities along a graph of interdependent tasks was derived from the Synthesis operating system [57, 58], where real-time tasks that are interlinked by pipes have their relative time slices adjusted whenever an I/O buffer approaches full or empty. An appropriate balance is achieved over many fine-grain iterations. Petra-Flow, however, does not require iteration to establish new task priorities when the priority of a result is changed. Petra-Flow is furthermore distinguished in that it establishes task priorities in terms of their results (variables) as opposed to the tasks themselves; this data-centric stance is also taken by the time-sensitive object model [15], where real-time constraints are placed on objects, causing update procedures to be invoked at appropriate times.

Regarding priority-mediated locks, the literature is rife with concurrency control

protocols that favor tasks with priority. Priority queuing of blocked tasks is the simplest of them [39]. Priority inheritance [82] is commonly used to help reduce priority inversion: the task holding a lock is boosted to the maximum priority of all the tasks that it are blocking. This is insufficient when critical sections may take a long time to execute, such as during periods of scarce resources. Locking protocols in the real-time database literature typically assume that tasks (transactions) can be aborted if a higher priority task arrives that conflicts with it [91, 94]. In that domain, all priority inversion can be avoided at the cost of re-starting aborted transactions. In our domain, we cannot safely assume that tasks in critical sections can be aborted and restarted.

Of the non-preemptable locking protocols, the closest to that of priority-mediated locks is the dynamic priority ceiling protocol (DPCP) [62]: for a fixed set of tasks that run periodically and require exclusive access to a resource, it determines the maximum priority (earliest deadline) task that intends to reserve the lock (its dynamic *priority ceiling*) and blocks any lower priority tasks that try to seize it. DPCP also includes priority inheritance based on this priority ceiling, not just those that are currently blocking for the resource. It has been shown to be deadlock-free under earliest deadline first scheduling. The principal difference with priority-mediated locks, besides the absence of the priority inheritance sub-protocol, is that tasks can determine how far in advance of locking they wish to stake their interest in each resource. As we saw in Section 6.4, this lets us control the trade-off between priority inversion and wasted utilization. DPCP is more restrictive in that priority ceilings are computed from the entire task set, i.e., 100% in advance.

Obsolete task cancellation is related to terminating orphan tasks in remote procedure call (RPC) systems [41]: an RPC invocation without side-effects may be killed if its calling process terminates. This is achieved by notifications of process deaths and, for fault tolerance, “keep alive” messages. A distributed implementation of Petra-Flow would need these features as well. What distinguishes Petra-Flow is

that a task may have multiple results written to distinct variables that each need to be checked for obsolescence.

The work in computational steering [27] is related to Petra-Flow in that it also allows users to direct a parallel computation in real-time. Its method, however, is primarily to adjust the parameters of a scientific computation, e.g., error tolerance, and although the computation is composed of concurrent tasks, user responsiveness cannot be improved by adjusting their relative priorities.

7.2.3 Responsiveness

The applications we built with Petra-Flow benefit a great deal from incremental quality, or multi-resolution, techniques by temporarily[2] trading off quality for responsiveness. There is a great deal of work in multi-resolution techniques for encoding and computing information, also known as *approximate results* [49, 18, 35, 19]. Petra-Flow itself does not implement any one technique, but supplies an infrastructure to support the programming of such techniques.

In contrast with the approach taken in this thesis, another way of leveraging multi-resolution techniques is to measure the current resource availability in the environment and generate a single quality result that meets the response time requirements [34, 35, 66]. As discussed in Chapter 3, this *dynamic sizing* approach is ill suited to highly volatile environments where resources may change dramatically between measurement and execution. In some environments this volatility can be eliminated by securing a guarantee for a known level of resources, known in the literature as *reservations* [7, 5] or *quality of service* [8, 30, 92, 97]. In other environments, the guarantee cannot be kept reliably because of conditions beyond the control of the resource scheduler, such as preserving bandwidth promises when

---

[2] The final quality result, though delayed somewhat by the intermediate results, is as good as without multi-resolution.

shifting from a wired interface to the network to a wireless one. Even when the scheduler does have the ability to keep its promises, they may need to be broken to accommodate new demands of greater importance [42, principle 3].

7.3 Future Work

This work could be extended in several interesting ways. First, the priority mechanism could be enhanced. In our experience with the applications we found that, although it typically does what we would like, it could make better trade-offs if the priority decayed as incremental results reached their final quality levels. While initially we want the visible objects to take priority over non-visible prefetching, at some quality level, prefetches should take favor. Such decisions are bound to be application-dependent. However, Petra-Flow could incorporate a common paradigm to ease the introduction of this feature.

Second, it may be possible to improve overall responsiveness by being less aggressive about starting threads on new input quality versions. In the current scheme, we start a thread on new quality inputs as soon as the input parameter semantics are satisfied. For the applications in the image domain we frequently found that the first two quality versions would arrive in rapid succession. Consumer tasks using the `Skip` semantics (run on every version, skipping backlog) are all started on the first version, yet it would be preferable to wait until the second quality version before starting the threads. We propose two ways to accomplish this. The first is a new option for input parameter synchronization semantics that lets one specify a time delay to wait before running on new quality versions— if better quality versions arrive in the meantime, we use the highest available when the time delay expires. This would be useful at low quality versions and when resources are plentiful. At high quality versions, the delay would typically be insignificant. The delay could be suppressed for the final quality version.

The second method is to supplement the global dependence graph with information about task execution time as a function of quality level, either specified by the programmer or measured by profiling, and use this information to perform some degree of global optimization about which quality versions to send down through the graph. Assuming resources are relatively stable, we can estimate whether it is best to execute on the next available version or to wait for the next. Making optimal decisions is likely to be NP-hard.

Petra-Flow uses variable splitting to expose concurrency. This may, however, be a problem for mobile computers that have stringent storage constraints due to their portability requirements [33, 76]. On such computers, Petra-Flow would need to be extended to control this memory vs. concurrency trade-off intelligently, rather than greedily splitting variables and executing tasks concurrently.

7.4 Conclusion

Future and modern computing environments can exhibit high variability in service delivery. This causes the responsiveness of traditional applications to suffer. While there are known techniques to cope with variability, they necessarily add to the complexity of building an application. To reduce this marginal cost, we have proposed a novel framework called Petra-Flow. In our experience building and executing applications with it, we find it a promising approach to this general problem that otherwise would need to be re-solved by many independent application programmers.

BIBLIOGRAPHY

[1] G. Abram and L. Treinish. An extended data-flow architecture for data analysis and visualization. In *Visualization '95*, pages 263–270. ACM SIGGRAPH, 1995.

[2] F. Adelstein and M. Singhal. Priority Ethernets: multimedia support on local area networks. In B. Fuhrt, editor, *Proceedings of the IASTED/ISMM International Conference. Distributed Multimedia Systems and Applications.*, pages 45–48, 1994.

[3] R. M. Adler. Emerging standards for component software. *Computer*, 28(3):68–77, March 1995.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[5] D. P. Anderson. Metascheduling for continuous media. *ACM Transactions on Computer Systems*, 11(3):226–252, August 1993.

[6] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, Feb. 1992.

[7] M. Arango, L. Bahler, M. Cochinwala, D. Cohrs, R. Fish, G. Gopal, N. Griffeth, G. E. Herman, T. Hickey, K. C. Lee, W. E. Leland, C. Lowery, V. Mak, J. Patterson, L. Ruston, M. Segal, R. C. Sekar, M. P. Vecchi, A. Weinrib, and S. Y. Wuu. The Touring Machine system. *Communications of the ACM*, 36(1):68–77, Jan. 1993.

[8] C. M. Aras, J. F. Kurose, D. S. Reeves, and H. Schulzrinne. Real-time communication in packet-switched networks. *Proceedings of the IEEE*, 82(1):122–39, Jan. 1994.

[9] J. J. Bae and T. Suda. Survey of traffic control schemes and protocols in ATM networks. *Proceedings of the IEEE*, 79(2):170–189, February 1991.

[10] P. G. J. Barten. The effects of pictures size and definition on perceived image quality. *IEEE Transactions on Electron Devices*, 36(9):1865–9, September 1989. (fig.4, p.1868).

[11] P. G. J. Barten. Evaluation of a subjective image quality with the square-root integral method. *Journal of the Optical Society of America A*, 7(10):2024–31, October 1990.

[12] T. Berners-Lee, R. Cailliau, A. Luotonen, Frystyk, H. Nielsen, and A. Secret. The world-wide web. *Communications of the ACM*, 37(8):76–82, Aug. 1994.

[13] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[14] G. Calhoun. *Wireless access and the local telephone network*. Artech House, 1992.

[15] H. R. Callison. A time-sensitive object model for real-time systems. *ACM Transactions on Software Engineering and Methodology*, 4(3):287–317, July 1995.

[16] W. E. Carlson. A survey of computer graphics image encoding and storage formats. *Computer Graphics*, 25(2):67–75, April 1991.

[17] K. S. Ce, C. M. Su, C. C. Tzu, and M. L. Chung. An object-oriented approach to develop software fault-tolerant mechanisms for parallel programming systems. *Journal of Systems and Software*, 32(3):215–25, March 1996.

[18] C. C. Ching. Photo CD and other digital imaging technologies: what's out there and what's it for? *Microcomputers for Information Management*, 10(1):29–42, March 1993.

[19] C. K. Chui. *Wavelet Analysis and its Applications, volumes 1 and 2*. Academic Press Inc., San Diego, California, 1992.

[20] H. Custer. *Inside Windows NT*. Microsoft Press, 1992.

[21] A. Dan, D. M. Dias, R. Mukherjee, D. Sitaram, and R. Tewari. Buffering and caching in large-scale video servers. In *Digest of Papers. COMPCON '95. Technologies for the Information Superhighway (Cat. No. 95CH35737). San Francisco, CA, USA*, pages 217–24, 5-9 March 1995.

[22] A. L. Davis and R. M. Keller. Data flow program graphs. *IEEE Computer*, 15(2):26–41, Feb. 1982.

[23] P. Domel. Mobile Telescript agents and the Web. In *COMPCON '96. Technologies for the Information Superhighway.*, pages 523–57, February 1996.

[24] B. Doorenbos, O. Etzioni, and D. Weld. A scalable comparison-shopping agent for the World-Wide Web. Technical Report 96-01-03, Dept. of Computer Science and Engineering, University of Washington., 1996.

[25] D. Duis and J. Johnson. Improving user-interface responsiveness despite performance limitations. In *IEEE COMPCON, Spring '90*, pages 380–386, 1990.

[26] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Computer Graphics Proceedings. SIGGRAPH 95. Los Angeles, CA, USA. ACM*, pages 173–82, 6-11 Aug. 1995.

[27] G. Eisenhauer, W. Gu, T. Kindler, K. Schwan, D. Silva, and J. Vetter. Opportunities and tools for highly interactive distributed and parallel computing. Technical Report GIT-CC-94-58, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, December 1994. Also in *Proceedings of The Workshop On Debugging and Tuning for Parallel Computing Systems*, Chatham, MA, October, 1994.

[28] O. Etzioni. Moving up the information food chain: Deploying Softbots on the World Wide Web. In *AAAI-96*, 1996.

[29] O. Etzioni and D. S. Weld. Intelligent agents on the Internet: Fact, fiction, and forecast. *IEEE Expert*, 10(4):44–49, Aug. 1995.

[30] D. Ferrari, A. Banerjea, and Z. Hui. Network support for multimedia a discussion of the tenet approach. *Computer Networks and ISDN Systems*, 26(10):1267–80, July 1994.

[31] A. Finkelstein, C. E. Jacobs, and D. H. Salesin. Multiresolution video. In *Proceedings of the ACM SIGGRAPH'96, New Orleans*, pages 281–290, August 1996.

[32] A. Finkelstein and D. H. Salesin. Multiresolution curves. In *Computer Graphics Proceedings. Annual Conference Series 1994. SIGGRAPH 94 Conference Proceedings. Orlando, FL, USA. ACM*, pages 261–8, 24-29 July 1994.

[33] G. H. Forman and J. Zahorjan. The challenges of mobile computing. *Computer*, 27(4):38–47, April 1994.

[34] A. Fox, E. A. Brewer, S. D. Gribble, and E. Amir. Adapting to client variability via on-demand dynamic transcoding. In *To appear in ASPLOS VII*, 1996.

[35] T. A. Funkhouser and C. H. Sequin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics Proceedings, Annual Conference Series*, pages 247–254, 1993.

[36] S. Gukal, E. Omiecinski, and U. Ramachandran. An efficient transient versioning method. In *Advances in Databases. 13th British National Conference on Databases, BNDOC Proceedings. Manchester, UK*, pages 155–71, 12-14 July 1995.

[37] S. Hall. Cellular digital packet data (CDPD). In *Second International Workshop on Mobile Multi-Media Communications. Bristol, UK. Hewlett-Packard Lab. Bristol Univ. IEEE Commun. Soc. Tokyo Chapter. Waseda Univ*, pages A4/3/1–6, 11-13 April 1995.

[38] R. H. J. Halstead. New ideas in parallel Lisp: language design, implementation, and programming tools. In *Parallel Lisp: Languages and Systems. US/Japan Workshop on Parallel Lisp Proceedings. Sendai, Japan*, pages 2–57, 5-8 June 1990.

[39] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[40] E. Hyden, J. Trotter, P. Krzyzanowski, M. Srivastava, and P. Agrawal. Swan: an indoor wireless atm network. In *1995 Fourth IEEE International Conference*

on Universal Personal Communications. Record. Gateway to the 21st Century (Cat. No. 95TH8128). Tokyo, Japan, pages 853–7, 6-10 Nov. 1995.

[41] V. Issarny, G. Muller, and I. Puaut. Efficient treatment of failures in RPC systems. In *Proceedings. 13th Symposium on Reliable Distributed Systems (Cat. No. 94CH35714). Dana Point, CA, USA. IEEE Comput. Soc. Tech. Committee on Distributed Process. IEEE Comput. Soc. Tech. Committee on Fault-Tolerant Comput. IFIP WG 10. 4 on Dependable Comput*, pages 170–80, 25-27 Oct. 1994.

[42] A. Jones and A. Hopper. Handling audio and video streams in a distributed environment. In *14th ACM Symposium on Operating Systems Principles. Ashville, NC, USA. ACM*, 5-8 Dec. 1993.

[43] D. S. Ketchell, S. S. Fuller, M. M. Freedman, and E. M. Lightfoot. Collaborative development of a uniform graphical interface. In *Sixteenth Annual Symposium on Computer Applications in Medical Care.*, pages 8–11, November 1992. http://www.washington.edu/willow.

[44] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, Feb. 1992.

[45] V. Konda and A. Kumar. Compiler algorithms for minimal reordering of the statements in DO loops. In *Sixth International Conference on Parallel and Distributed Computing Systems. Louisville, KY, USA. Int. Soc. Comput. & Their Appl. -ISCA. IEEE Comput. Soc*, pages 332–8, 14-16 Oct. 1993.

[46] H. F. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, second edition, 1991.

[47] D. J. Kruglinski. *Inside Visual C++.* Microsoft Press, 1996. Chapter 11.

[48] A. Kutlu, H. Ekiz, and E. T. Powner. Wireless control area network. In *IEE Colloquium on Networking Aspects of Radio Communication Systems (Ref. No.1996/056)*, pages 3/1–4, 1996.

[49] J. L. Kwei and S. Natarajan. FLEX: towards flexible real-time programs. *Computer Languages*, 16(1):65–79, 1991.

[50] J. A. Landay and T. R. Kaufmann. User interface issues in mobile computing. In *Proceedings. Fourth Workshop on Workstation Operating Systems (Cat. No. 93TH0553-8). Napa, CA, USA. IEEE Comput. Soc. Tech. Committee Oper. Syst. & Appl. Environ*, pages 40–7, 14-15 Oct. 1993.

[51] A. M. Law and W. D. Kelton. *Simulation modeling and analysis.* McGraw-Hill, 1982.

[52] D. Lea. libg++, the GNU C++ library. In *USENIX Proceedings. C++ Conference. Denver, CO, USA*, pages 243–56, 17-21 Oct. 1988.

[53] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, 1987.

[54] D. Libes. expect: Curing those uncontrollable fits of interaction. In *Proceedings of the Summer 1990 USENIX Conference. Anaheim, CA, June 11–15*, 1990.

[55] K. Maine, C. Devieux, and P. Swan. Overview of iridium satellite network. In *WESCON/95 Conference Record (Cat. No. 95CH35791). San Francisco, CA, USA. IEEE. ERA*, pages 483–90, 7-9 Nov. 1995.

[56] B. B. Mandelbrot. *The fractal geometry of nature.* W. H. Freeman, updated and augmented edition, 1983.

[57] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. *Operating Systems Review*, 23(5):191–201, 1989.

[58] H. Massalin and C. Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems. Winter 1990*, 3(1):139–73, 1990.

[59] J. McAffer and D. Thomas. Eva: an event driven framework for building user interfaces in Smalltalk. In *Proceedings of Graphics Interface '88. Edmonton, Alta. , Canada. Canadian Man-Comput. Commun. Soc. Canadian Image Process. & Pattern Recognition Soc. et al*, pages 168–75, 6-10 June 1988.

[60] K. Mehlhorn and S. Naher. LEDA: a platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, January 1995. ftp://ftp.mpi-sb.mpg.de/pub/LEDA.

[61] Microsoft Corporation. *Microsoft Backgrounder: Object Linking & Embedding, version 2.0*, November 1992.

[62] I. C. Min and J. L. Kwei. Dynamic priority ceilings: a concurrency control protocol for real-time systems. *Real-Time Systems*, 2(4):325–46, Nov. 1990.

[63] B. A. Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *UIST Fourth Annual Symposium on User Interface Software and Technology. Proceedings of the ACM Symposium on User Interface Software and Technology. Hilton Head, SC, USA. ACM. SIGGRAPH. SIGCHI*, pages 211–20, 11-13 Nov. 1991.

[64] S. Narayanaswamy, S. Seshan, E. Amir, E. Brewer, R. W. Brodersen, F. Burghardt, A. Burstein, C. C. Yuan, A. Fox, J. M. Gilbert, R. Han, R. H.

Katz, A. C. Long, D. G. Messerschmitt, and J. M. Rabaey. A low-power, light-weight unit to provide ubiquitous information access application and network support for InfoPad. *IEEE Personal Communications*, 3(2):4–17, April 1996.

[65] R. S. Nickerson. Man-computer interaction: a challenge for human factors research. *IEEE Transactions on Man-Machine Systems*, 10:164–180, Dec. 1969.

[66] B. D. Noble, M. Price, and M. Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. *Computing Systems. Fall 1995*, 8(4):345–63, 1995.

[67] S. C. North and E. Koutsofios. Applications of graph visualization. In *Proceedings Graphics Interface '94. Banff, Alta., Canada*, pages 235–45, 18-20 May 1994.

[68] J. Ousterhout. Why threads are a bad idea (for most purposes). In *USENIX 1996 Annual Technical Conference. Invited talks. Submitted notes.*, pages 29–36, 22–26 January 1996.

[69] J. K. Ousterhout. An X11 toolkit based on the Tcl language. In *Proceedings of the Winter 1991 USENIX Conference. Dallas, TX, January 21–25*, pages 105–15, 1991. http://www.sunlabs.com/research/tcl.

[70] D. A. Patterson and J. L. Hennessy. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, second edition, 1990.

[71] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, 1988.

[72] C. Plenge. HIPERLAN: A decentrally organised wireless LAN. *Telektronikk*, 91(4):88–98, 1995.

[73] A. Rast. Wireless communications engineer, Inficom Inc., Seattle. Personal communication, 1996.

[74] J. R. Rasure and C. S. Williams. An integrated data-flow visual language and software development environment. *Journal of Visual Languages and Computing*, 2(3):217–246, 1991.

[75] A. Rushinek and S. F. Rushinek. What makes users happy? *Communications of the ACM*, 29:594–598, 1986.

[76] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Fifteenth ACM Symposium on Principles of Distributed Computing. Philadelphia, PA.*, May 1996.

[77] M. Satyanarayanan, B. Noble, P. Kumar, and M. Price. Application-aware adaptation for mobile computing. *Operating Systems Review*, 29(1):52–5, Jan. 1995.

[78] B. N. Schilit and D. Duchamp. Adaptive remote paging for mobile computers. Technical Report CUCS-004-91, Dept. of Computer Science, Columbia University, NY, Feb. 1991.

[79] C. Schmidtmann, M. Tao, and S. Watt. Design and implementation of a multi-thread Xlib. In *USENIX Association. Proceedings of the Winter 1993 USENIX Conference. San Diego, CA, USA. USENIX*, pages 193–203, 25-29 Jan. 1993.

[80] J. Sedayao. World wide web network traffic patterns. In *Digest of Papers. COMPCON '95. Technologies for the Information Superhighway (Cat. No. 95CH35737). San Francisco, CA, USA*, pages 8–12, 5-9 March 1995.

[81] E. Selberg and O. Etzioni. Multi-service search and comparison using the MetaCrawler. In *Proceedings of the 4th World Wide Web Conference*, pages 195–208, 1995.

[82] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–85, September 1990.

[83] L. Sha, R. Rajkumar, and J. P. Lehoczky. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, 1991.

[84] S. Shaio, A. Van Hoff, and H. Jellinek. Java and HotJava: a comprehensive overview. In *COMPCON '96. Technologies for the Information Superhighway*, pages 424–429, February 1996. http://java.sun.com/.

[85] S. Sheng, A. Chandrakasan, and R. W. Brodersen. A portable multimedia terminal. *IEEE Communications Magazine*, 30(12):64–75, Dec. 1992.

[86] Silicon Graphics, Inc., Mountain View, CA. *IRIX Explorer User's Guide*, 1992.

[87] J. A. Smith. The multi-threaded X (MTX) window system and SMP. In *Distributed Computing, Practice and Experience. Proceedings of the Autumn 1992 OpenForum Technical Conference. Utrecht, Netherlands. 88Open. Cognos. Digital Equipment. BULL. IBM. et al*, pages 131–43, 25-27 Nov. 1992.

[88] C. Solomon. Researcher, Hewlett Packard Research Lab, Bristol. Personal communication regarding the Piglet mobile computer, 1992.

[89] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1991.

[90] C. D. Tait and D. Duchamp. Detection and exploitation of file working sets. In *11th International Conference on Distributed Computing Systems (Cat. No. 91CH2996-7). Arlington, TX, USA. IEEE*, pages 2–9, 20-24 May 1991.

[91] A. Thomasian. Performance analysis of locking policies with limited wait depth. In *1992 ACM Sigmetrics and Performance '92. Newport, RI, USA. ACM. IFIP*, 1-5 June 1992.

[92] D. Towsley. Providing quality of service in packet switched networks. In *Performance Evaluation of Computer and Communication Systems. Joint Tutorial Papers Performance '93 and Sigmetrics '93. Rome, Italy and Santa Clara, CA, USA*, pages 560–86, 1993.

[93] K.-H. Tzou. Progressive image transmission: a review & comparison. *Optical Engineering*, 26(7):581–589, July 1987.

[94] O. Ulusoy and G. G. Belford. Real-time transaction scheduling in database systems. *Information Systems*, 18(8):559–80, December 1993.

[95] C. Upson, T. F. Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, 1989.

[96] C. L. Viles and J. C. French. Availability and latency of world wide web information servers. *Computing Systems. Winter 1995*, 8(1):61–91, 1995.

[97] A. Vogel, B. Kerherve, B. G. von, and J. Gecsei. Distributed multimedia and QOS: a survey. *IEEE Multimedia. Summer 1995*, 2(2):10–19, 1995.

[98] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI). Monterey, CA, USA. ACM. IEEE*, pages 1–11, 14-17 Nov. 1994.

[99] T. Watson. Application design for wireless computing. In *Proceedings. Workshop on Mobile Computing Systems and Applications (Cat. No. 94TH06734). Santa Cruz, CA, USA. IEEE Comput. Soc. Tech. Committee on Operating Syst. & Appl. Environments (TCOS0. ACM SIGOPS*, pages 91–4, 8-9 Dec. 1995.

[100] T. Watson. Effective wireless communication through application partitioning. In *Proceedings Fifth Workshop on Hot Topics in Operating Systems (HotOS-V) (Cat. No. 95TH8059). Orcas Island, WA, USA. IEEE Comput. Soc. Tech. Committee on Oper. Syst. & Application Environ. (TCOS)*, pages 24–7, 4-5 May 1995.

[101] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):74–84, July 1993.

[102] L. R. Welch, B. Ravindran, J. Henriques, and D. K. Hammer. Metrics and techniques for automatic partitioning and assignment of object-based concurrent programs. In *Proceedings. Seventh IEEE Symposium on Parallel and Distributed Processing (Cat. No. 95TB8131). San Antonio, TX, USA. IEEE Comput Soc. Tech. Committee on Comput. Architecture. IEEE Comput. Soc. Tech. Committee on Distributed Process. IEEE Comput. Soc. Dallas Chapter*, pages 440–7, 25-28 Oct. 1995.

[103] T. A. Welch. A technique for high performance data compression. *IEEE Computer*, 17(6), 1984.

[104] J. E. White. *White paper: Telescript Technology: The Foundation for the Electronic Marketplace.* General Magic Inc., 1994. http://www.genmagic.com/Telescript.

[105] W. Wieczerzycki. Long-duration transaction support in design databases. In *Proceedings of the 1995 ACM CIKM International Conference on Information and Knowledge Management. Baltimore, MD, USA. ACM*, pages 362–9, 28 Nov. - 2 Dec. 1995.

[106] C. Xu. A graph transformation algorithm for concurrency control in a partitioned database. *Information Processing Letters*, 38(1):43–8, 12 April 1991.

[107] P. S. Yen. An event-driven model-view-controller framework for Smalltalk. In *OOPSLA '89, Object-Oriented Programming: Systems, Languages and Applications. New Orleans, LA, USA. ACM*, 1-6 Oct. 1989.

[108] D. Young. Developing applications with C++ and Motif. In *Object Expo. Conference Proceedings. New York, NY, USA*, pages 315–20, 6-10 June 1994.

[109] D. A. Young. *X window system, Programming and applications with Xt OSF/Motif edition.* Prentice Hall, 1990.

VITA

George Henry Forman was born in Pasadena, California on June 7, 1966, and was valedictorian of Lake Dallas High School, Texas in 1984. He graduated *magna cum laude* from Pomona College, California with a B.A. degree in Mathematics in 1988. With a Fulbright fellowship the following year, he was hosted by Professor Niklaus Wirth at the Swiss Federal Institute of Technology, Zürich where he worked under Professor Anaratone on the design of the K2 parallel computer. Since then, he has been pursuing graduate studies in the Department of Computer Science and Engineering at the University of Washington, where he received his M.S. degree in 1992, and his Ph.D. degree in 1996. He has been elected to both Phi Beta Kappa and Sigma Xi.