Simplifying Cyber Foraging

Rajesh Krishna Balan

May 2006

CMU-CS-06-120

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

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

Thesis Committee:
Mahadev Satyanarayanan, Chair
David Garlan
Gregory Ganger
Srinivasan Seshan
Hari Balakrishnan, *Massachusetts Institute of Technology*

This research was partially supported by the National Science Foundation (NSF) under grant numbers ANI-0081396 and CCR-0205266, by a USENIX graduate fellowship, by an IBM Graduate Fellowship, and by an equipment grant from the Hewlett-Packard Corporation (HP). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF, USENIX, IBM, HP, or Carnegie Mellon University. All unidentified trademarks mentioned in this dissertation are properties of their respective owners.

Keywords: interactive applications, mobile computing, pervasive computing, ubiquitous computing, remote execution, software engineering, stub generation, user study, usability testing, operating system support

To Yamuna, my beloved wife.

Abstract

The rapid proliferation of mobile handheld computing devices, such as cellphones and PDAs, has led to an unfortunate conflict. On one hand, we have light mobile computing devices that can be carried anywhere. However, on the other hand, these devices are frequently unable to execute applications that are of highest value to a mobile user such as language translators and speech recognizers. One way to resolve this conflict is to use *cyber foraging* – utilize compute resources available in the environment to augment the capabilities of mobile devices.

A key challenge in enabling cyber foraging is that there exist many applications of high value to a mobile user that must be *quickly*, *easily*, and *effectively* retargeted to support cyber foraging. This retargeting is made more difficult as applications can be written in any programming language and style. In this thesis, *quickly* refers to the retargeting time, *easily* refers to the retargeting effort, and *effectively* refers to the retargeted application's runtime performance.

This dissertation shows that it is possible to quickly, easily, and effectively retarget computationally-intensive useful applications for cyber foraging. I developed a process called *RapidRe* that allows even novice developers to easily, quickly, and effectively retarget large unfamiliar applications for cyber foraging. To create RapidRe, I first developed a powerful remote execution system, called *Chroma*, that is able to achieve excellent application performance in mobile environments. Chroma uses the concept of *tactics* to greatly reduce its search space when deciding the optimal remote partitioning of applications. Tactics are enumerations of the useful application partitionings. At runtime, Chroma picks the tactics that would have the optimal performance for the given resource environment and user preferences.

I then developed a domain-specific language, called *Vivendi*, that allows developers to specify the adaptive characteristics of an application that are relevant for mobile computing. These characteristics include the parameters, fidelity variables, and tactics of the application. The parameters give hints about the expected application resource usage. These hints are used by Chroma to decide the optimal tactic. The fidelity variables are application settings that Chroma must set based on the available resources. Finally, tactics are described in two parts; the first part is the list of application procedures that can be remotely executed and the second is the possible ways to combine these procedures to do useful work.

RapidRe is language agnostic and consists of four steps. In step 1, application developers specify the adaptive characteristics of the application using Vivendi. In step 2, the stub

generator processes this description and creates most of the interface code needed for the application to work with Chroma. In step 3, the developer inserts automatically generated API calls, created in step 2, into the application to create the client and server components of the applications. Finally, in step 4, the client and server components are compiled, together with the automatically generated code and the Chroma libraries, to create the final retargeted adaptive application.

I validated my dissertation in the following way. I first conducted an extensive user study that showed that novice developers can use RapidRe to quickly, easily, and effectively retarget large applications for cyber foraging. In particular, novice developers can retarget large, complex, and unfamiliar applications in under 4 hours with no knowledge of Chroma and minimal knowledge of the application. I also show that these novice-retargeted applications achieve comparable performance to expert-retargeted applications. I then present extensive runtime system measurements that show that these expert-retargeted applications can achieve excellent performance under various cyber foraging scenarios. Finally, I validated that RapidRe is versatile by performing all of the previous validations using a large and diverse set of computationally-intensive useful mobile applications.

Acknowledgements

I found completing this dissertation to be very enjoyable experience. I believe this was, in large part, due to my constant interaction with some tremendous people.

My advisor, Satya, was instrumental in making this dissertation happen. He provided constant advice about possible research avenues and was a constant guide during this process. I am very grateful for the opportunity to work with him. It has definitely sharpened my research ability and helped me grow as an individual and a professional. Perhaps the most important lesson I learned from him was how to effectively convey my thoughts and ideas in both written and spoken mediums.

I am also very grateful to my thesis committee members, Hari Balakrishnan, David Garlan, Greg Ganger, and Srini Seshan, for guiding my research and providing invaluable advice that helped me fully flesh out the rough bits in my dissertation.

My thesis builds upon the Odyssey system. I am thus indebted to Brian Noble, Dushyanth Narayanan, and Jason Flinn, for the key components of Odyssey that they built. In addition, Dushyanth and Jason, when they were still students, provided invaluable advice, guidance, and friendship during my initial years at Carnegie Mellon. Tadashi Okoshi and SoYoung Park were instrumental in developing and testing the first version of Chroma. It would have been much harder to build and test Chroma without their help.

During my stay at Carnegie Mellon, I have made many friends and they have been vital in making the Ph.D. process a fun and enriching experience. I would particularly like to thank Mukesh Agrawal, Jan Harkes, and Tracy Farbacher. They were my closest friends and kept me sane and amused during this long and eventful six year ride. I will definitely miss the three of them when I leave. In addition, I would like to thank Hu Ning Ning, Julio Lopez, Mahim Mishra, Ted Wong, Joao Sousa, and the above mentioned Jason, Dushyanth, SoYoung, and Tadashi for their friendship.

Finally, I am especially grateful to Yamuna and my mom for being extremely patient with me while I was doing this “Ph.D. thing”. Their support, sacrifices, and love were instrumental in me finishing this thesis.

Contents

List of Figures

List of Tables

Glossary of Thesis Terms

Chroma	The dynamic adaptive remote execution system developed for this thesis
Data Decomposition	A computing paradigm where data is split into smaller pieces. Each of these smaller pieces is then processed by separate servers. Finally, the partial results are recombined to create the final output
Fidelity	Application-specific notion of quality
Fidelity Variable	Application-specific variables that affect quality. The runtime values of these variables are decided by Chroma
Join Function	The application-specified function that can recombine partial results to form a complete result. Needed for data decomposition
Operation	The part of the application that does the computationally intensive work. E.g., rendering loop for graphics applications, translation routine for language translators
Runtime Setting	The exact tactic plan and fidelity variable settings chosen by Chroma for the current operation. This is the complete set of information needed by the application to successfully perform the operation
Parameter	Application-specific variables that determine the resource usage of the application
RapidRe	Process created to allow rapid retargeting of applications The 4 main components of RapidRe are Vivendi, Chroma, a a smart stub generator, and a well-defined procedure for retargeting applications
RPC	Remote procedure call
Split Function	The application-specified function that can split input data into smaller pieces. Needed for data decomposition
Tactics	The possible partitionings of an application
Tactic Plan	A particular tactic with precise server selections for each RPC in the tactic. The exact tactic and servers to use are decided by Chroma
Vivendi	Language used to describe the adaptive characteristics of applications

Chapter 1

Introduction

The rapid proliferation of mobile handheld computing devices, such as cellphones and PDAs, has led to an unfortunate conflict. On one hand, we have light mobile computing devices that can be carried anywhere. On the other hand, these devices are frequently unable to execute applications that are of highest value to a mobile user. These applications include natural language translation and speech recognition which would, for example, be extremely helpful to a foreign traveller. Optical character recognition of signs in a foreign script could help a lost traveller find his way. An augmented reality application coupled with a light heads up display might allow mobile users to augment their vision with extra information such as the names of people they meet etc. Unfortunately, the CPU, memory and energy demands of these applications far outstrip the capabilities of devices that people are willing to carry or wear for extended periods of time as these devices are optimized for size and weight and not computing power.

One way to resolve this conflict is to use the computing capabilities of nearby servers, via remote execution, to augment the capabilities of mobile devices. This transient and opportunistic use of resources is known as *cyber foraging*. Even though compute servers for public use are currently not common, this dissertation addresses future environments where cheap commodity machines will become widely dispersed for public use as compute servers. Indeed, as the price of computing continues to fall, they may become as common as water fountains, lighting fixtures, chairs or other public conveniences that we take for granted today. When public infrastructure is unavailable, other options may exist. For example, the body-worn computer of an engineer who is inspecting the underside of a bridge may use a compute server in his truck parked nearby. Finally, if there are absolutely no compute resources available, the mobile device can still fall back onto its own limited resources.

Successfully implementing cyber foraging requires many pieces. For example, service discovery and security mechanisms are necessary to detect available servers and to securely use them. In this thesis, I build on others' work for these other pieces, and concentrate solely on the following key challenge: how to *quickly*, *easily*, and *effectively* retarget applications to support cyber foraging. This retargeting is difficult because applications are written in any programming language and style. In this thesis, *quickly* refers to the retargeting time, *easily* refers to the retargeting effort (i.e., how difficult was the retargeting. This may or may not be correlated with the retargeting time), and *effectively* refers to the retargeted application's runtime performance.

A quick, easy, and effective retargeting solution is vital due to the short useful life of mobile devices. Smart cell phones, wearable computers, PDAs and other mobile devices are emerging at a dizzying rate that shows no sign of slowing [69, 96, 126, 223]. However, these devices also have very short market life spans; typically just a year or less. Hence, developers cannot spend months or even weeks either building brand new applications or retargeting existing applications for these new devices.

The key solution insight is that a large number of these applications share commonalities that allow a combination of abstraction, language support, stub tools, and a dedicated runtime layer to be used to satisfy the retargeting goal. In the rest of this chapter, I first discuss previous approaches to retargeting applications for cyber foraging. I then briefly describe my solution. I next state the thesis statement and describe the validation plan. Finally, I end with a roadmap describing the rest of this dissertation.

1.1 Previous Approaches to Supporting Cyber Foraging

Previous approaches at providing application support for cyber foraging can be broadly categorized into two main groups. The first group uses thin client solutions, such as VNC [179], SSH [1], or web-based services such as GoToMyPC [47], to allow mobile devices to execute applications running on remote servers. These solutions do not require any application modification and are extremely easy to use and deploy. To be usable, they require low latency network connectivity to the remote server. However, this connectivity may not be present, or may be present in limited and expensive forms, in many environments, such as planes and city outskirts, that a mobile user might find himself in. In such situations, thin clients either cannot be used at all (no bandwidth) or have unacceptably poor interactive performance (high latency link). They are unable to adapt to the lack of appropriate bandwidth and it is entirely up to the user to manually decide on the appropriate corrective action.

To overcome this major shortcoming, thicker client solutions have been developed. In these solutions, part of the application also runs locally on the mobile device. Hence, when a remote machine cannot be used, the local application component can still provide a degraded quality output and/or a corrective action.

The extreme form of this solution is the completely thick client solution where the entire application runs on the mobile device. However, this solution is impossible for computationally-intensive applications as they are unable to run effectively on resource-limited mobile devices.

To support these computationally-intensive applications, two types of thick client solutions have been proposed. Both of these solutions use a smart runtime system, running on the mobile device, to dynamically change the application quality and to migrate parts of the application to remote servers. They differ in how the runtime interacts with applications.

In the first method, known as application-transparent adaptation, the runtime uses existing application APIs to control and communicate with the application. This method requires no application modification. However, the adaptation it can perform is limited by the existing APIs. Additionally, many applications do not have APIs that allow them to be externally adapted. Examples of systems that use this kind of adaptation include Coign [115] and Puppeteer [54].

The second method, called application-aware adaptation, requires applications to be explicitly modified to work with the runtime. This method is thus able to work with any application and exploit the full adaptive capabilities of the runtime. However, it requires far more effort than application-transparent adaptation as each application must now be manually modified. Additionally, this requires the source for each application to be available. Odyssey [160, 76, 152] and Rover [121] are examples of systems that use application-aware adaptation.

For this thesis, I use application-aware adaptation as I wanted to effectively support a broad and diverse range of useful mobile applications. The key goal of this thesis is to show that it is possible to achieve the full power of application-aware adaptation without significant retargeting overheads.

1.2 Solution: RapidRe

To achieve this goal, I propose a solution, called *RapidRe*, that is based on the well-known approach of *little languages* [25]. By developing abstractions that are well-matched to the problem of cyber foraging, I enable a compact static description of the user-specified meaningful partitions and fidelities of an application. A stub-generation tool uses this description to create application-specific interfaces (API) to the underlying adaptive runtime system that provides the dynamic components necessary for adaptation in changing mobile environments. These APIs are then inserted into the application to complete the retargeting process.

RapidRe is able to easily and quickly retarget applications for cyber foraging. In addition, the retargeted applications have excellent performance under various mobile scenarios.

1.3 The Thesis

The thesis statement can thus be stated as follows:

It is possible to easily, quickly, and effectively modify an important class of existing computationally-intensive applications, such as language translators, speech recognizers, face detectors, and graphics applications, for cyber foraging.

This dissertation establishes the thesis via the following steps:

- First, it identifies the salient characteristics of a cyber foraging environment that need to be addressed by the solution. It then clearly identifies the kinds of applications that this thesis is focusing on.
- It then identifies the commonalities of these applications that allow a general solution to be developed.
- Next, it presents a solution, called RapidRe, that exploits these commonalities. In particular, RapidRe uses a combination of abstraction, language support, stub tools, and a dedicated runtime layer to accomplish its goals.
- It then demonstrates the versatility of RapidRe by using it to retarget a large and diverse set of eight computationally-intensive useful applications.
- It then demonstrates, via a software usability study, that RapidRe allows even novice developers to quickly and easily retarget these applications, in under four hours, to support cyber foraging. In addition, the quality of these retargeted applications is comparable to applications retargeted by expert developers.
- Finally, it shows that retargeted applications are effective as they can achieve excellent performance in a cyber foraging environment. In particular, they achieve comparable performance to an oracle that always achieves the best possible performance

1.4 Dissertation Roadmap

The rest of this dissertation is organized into six chapters and four appendixes as follows:

Chapter 2 describes the characteristics of cyber foraging environments. In particular, it defines the types of applications, computationally-intensive interactive applications, that this dissertation is concerned with. It then details the requirements of any solution geared towards allowing applications to be rapidly retargeted for cyber foraging. These requirements are that the solution must be language independent, work with even novice developers, and result in high quality applications. Next, it describes the insight that facilitates a solution; namely that for the class of applications being considered, the application information needed to make them adaptive is small and can be concisely described. The chapter than describes the RapidRe process that uses this insight to provide a solution. It first describes the three components, a little language called *Vivendi*, the Chroma runtime system, and a smart stub generator, that make up RapidRe. It then describes, in detail, the four stage process that developers must use to retarget their applications.

Chapter 3 presents the ten applications used to validate this dissertation. It describes the characteristics of each applications and gives the tactics file for each application. This chapter also demonstrates that RapidRe is applicable to a wide range of applications in terms of functionally, language the application was written in, and application size.

Chapters 4 and 5 present the validation for this dissertation. Chapter 4 presents the results of a detailed software usability study that shows that novice developers are able to use RapidRe to quickly and easily retarget large computationally intensive applications for cyber foraging. It also shows that quality of the retargeted applications is comparable to that achievable by an expert developer. In Chapter 5, I show that these expert-modified applications achieve excellent absolute runtime performance in a variety of mobile environments. These two chapters validate the thesis statement and show that it is possible to quickly, easily, and effectively retarget computationally-intensive interactive applications for cyber foraging.

Finally, Chapters 6, and 7 present the related work and the dissertation conclusion (that summarizes the main contributions of the dissertation and presents future work) respectively.

This dissertation also has five appendices.

Appendix A, describes Chroma. Chroma is the dynamic adaptive runtime component of RapidRe, built for this dissertation, that dynamically partitions applications in a dynamic environment. It is the key system component that allows applications to achieve excellent performance in a cyber foraging environment. This appendix first describes the goals of Chroma and then proceeds to give an overview of every component of Chroma. In particular, it describes how Chroma decides on the optimal partitioning and fidelity settings for an application. It also describes how Chroma discovers, secures, and uses available servers in the environment. Finally, it describes the API used by applications to communicate with Chroma. Appendix B provides the complete details of the Vivendi syntax. Appendix C presents the details of how each individual user study was conducted while Appendix D and Appendix E provide the actual documentation and questionnaires, respectively, that were given to each participant during each user study experiment.

Chapter 2

Retargeting Applications for Cyber Foraging

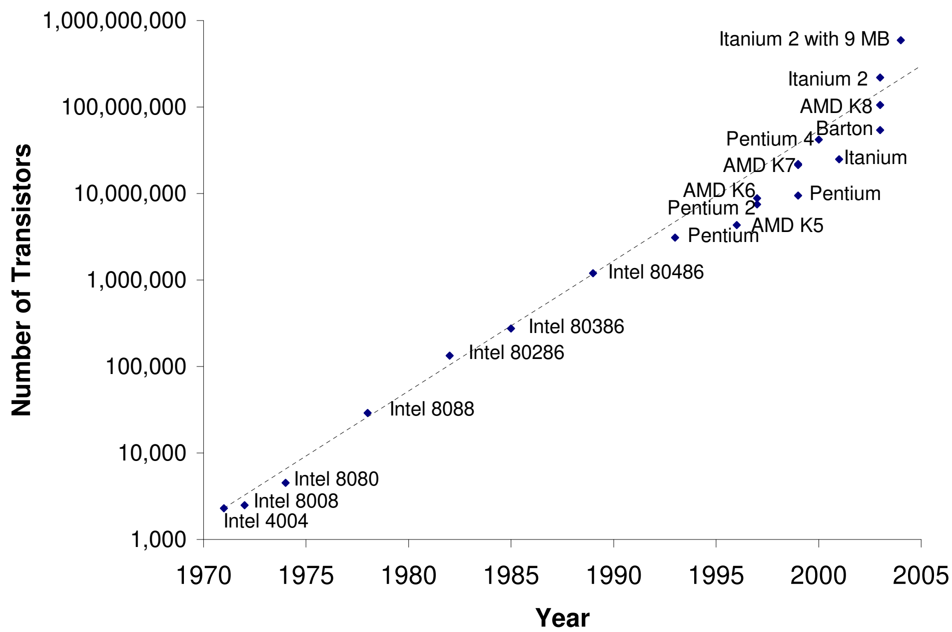
In this chapter, I describe what cyber foraging is. I then list the salient characteristics of a cyber foraging environment. I then describe the kinds of applications that this thesis is concerned with. Finally, I present the RapidRe solution that allows developers to quickly, easily, and effectively retarget applications to support cyber foraging.

2.1 What Is Cyber Foraging?

As mentioned earlier, cyber foraging is the opportunistic use of resources in the environment to augment a device's capabilities. The scenario below describes how cyber foraging could be used in practice.

"Jane is traveling in Portugal with her PDA. However, Jane speaks very little Portuguese and is depending on the language translation application on her PDA to assist her in communication. However, the PDA, by itself, can only run the language translation application at the lowest quality due to its limited resources. In particular, the PDA does not have enough memory to use all of the language translator's data files. Fortunately, Jane's PDA is cyber foraging enabled and can use servers in the environment to both run the language translator faster and with higher quality.

Currently, Jane is lost in the back alleys of Lisbon. She approaches a local and attempts to ask the local for directions. However, because there are no servers available, the language translator has to be run locally on the PDA. This results in degraded quality and Jane is only able to ask "Where is the nearest cafe?" to the local. She hopes that a cafe will provide servers that can be used to perform more powerful translations. The local, indicates via hand gestures, where the nearest cafe is. Jane thanks the local and heads towards the cafe. As Jane approaches the cafe, her PDA discovers available servers, provided by the cafe, in the environment and establishes a secure connection with those servers. Jane's PDA can now use those servers to perform more powerful language translation. Noting this, Jane approaches another local and starts obtaining directions to her hotel. After ob-



The figure shows the increase in the number of transistors in Intel and AMD processors. The Intel data was obtained from [3] and the AMD data from [2]. The dotted line plots a doubling of the number of transistors every 2 years.

Figure 2.1: The Moore's Law Effect

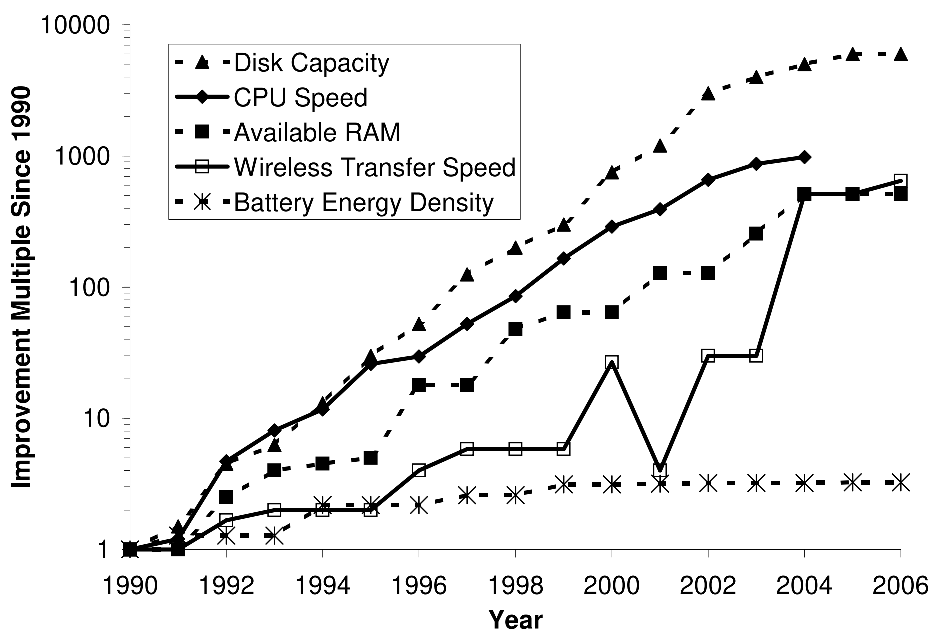
taining these directions, Jane continues the discussion and starts asking the local about activities and festivals that might be of interest to her. Finally, Jane thanks the local and, no longer lost, heads off towards her next destination."

2.2 The Cyber Foraging Environment

The above scenario presents some characteristics of the cyber foraging environment that I will now expand upon.

2.2.1 Small Light Devices

In 1965, Gordon Moore published his famous article in which he stated that the number of transistors in integrated circuits would double roughly every two years [148]. This has proved to be true (Figure 2.1) and similar exponential increases have been observed for other computing components such as RAM and hard disk densities. However, not all computing components exhibit this kind of rapid growth. In particular, as shown in Figure 2.2, the density of battery cells (the bottom-most line in the figure) has improved very



This figure was produced using data generously provided by Thad Starner (the graph originally appear in Paradiso and Starner [169]). The original data stopped at 2003 and I have added values for 2004 to 2006 (except for CPU speed which stops at 2004 as published speed values for the latest processors (Intel Core Duo and AMD Turion 64 X2 Mobile) are not publically available yet). The figure shows the relative improvements in laptop computing technology from 1990-2003. The wireless-connectivity curve considers only cellular standards in the US, and does not consider short-range 802.11 “hot-spots”. The dip in 2001 marks the removal of the Metricom network while the large increase in 2004 marks the introduction of the 1x Evolution-Data Optimized (EVDO) service. Overall, the figure shows that the rate of increase of most laptop computing components is starting to flatten out.

Figure 2.2: Improvements in Technology Over the Years

slowly. Every now and then, the battery capacity increases a bit due to improvements in technology (such as the switch from Nickel-Cadmium to Nickel-metal hydride, and finally to Lithium-Ion). However, within each new technology tree, the improvements have been mostly linear.

This presents a huge conundrum to mobile device manufacturers as consumers of these devices frequently demand smaller and lighter devices. For example, a mobile watch like IBM's Linux watch [153] is far more likely to be carried everywhere compared to a laptop computer. Manufacturers understand this, and strive to make each new mobile device (such as cellphones and PDAs) generation as small and light as possible.

However, to achieve the desired size, manufacturers are forced to use smaller batteries in these devices. As a result, they also have to use less powerful CPU, memory, disk, and network components. Otherwise, the battery lifetime of these devices would be too low to be useful. Each new mobile generation has significantly more CPU, memory, and disk capacity than the previous generation (due to improvements in these components). However, due to battery concerns, mobile devices are unable to use the most powerful (or largest) current-generation computing components. Hence, there is a large gap between the resource availability of desktop computers and mobile devices. This gap is unlikely to change in the next ten years without a significant improvement in battery technology (such as a switch to fuel cells).

This gap leads to the fundamental resource conflict that is the focus of this dissertation. This conflict is that there are many large computationally intensive applications, that were originally developed for desktop machines, that would also be highly useful for mobile users. Unfortunately, mobile devices do not have the resources to effectively run these applications. These applications are described in more detail in Section 2.2.3.

2.2.2 Variable Resources and User Preferences

Another salient feature of mobile computing environments is that they are characterized by highly variable resource availability. This has been noted by many researchers such as Fox et al. [80], Katz [128], and Satyanarayanan [189]. In particular, wireless bandwidth availability can vary by orders of magnitude in just a few seconds. This has been demonstrated for many different environments such as universities [65], commercial factory floors [232], metropolitan wireless networks [8, 137], local area wireless networks [120, 178], home wireless networks [236], and even ideal undistorted wireless environments [97].

Additionally, these mobile devices have to adjust to the needs of users. For example, the user may require highly accurate application results. This in turn needs to be translated into appropriate settings for the individual applications running on the device. The user may then require that low application latency be the highest priority. This new policy will require the mobile device to change the settings for every application from accuracy to low latency.

This combination of resource variation and changing user preferences requires mobile devices to be able to dynamically change application behaviour. For example, an application running remotely on a server may need to dynamically change which server to use depending on the available wireless bandwidth and the current load on the remote server. Alternatively, mobile applications may need to decrease their quality to meet a user specified latency goal.

2.2.3 Application Taxonomy

In this section, I present a taxonomy of mobile applications and clearly identify the application areas that this dissertation is addressing. I characterize potential mobile applications along three axes. The first axis is "Is the application a useful mobile application?". In this dissertation, I only consider applications that are useful for mobile users. Hence, applications such as web server load-balancing tools and integrated development environments like Eclipse [66] and Visual Studio [145] are not considered.

The set of useful mobile applications is then further classified according to two axes:

- **Interactive**: Does the application have an interactive or continuous nature? For this dissertation, an interactive application has the property that it waits for input from the user before performing work. The results of this work are then presented to the user and the application then returns to waiting for input from the user. For example, a language translator that performs translation only when provided input by the user. A continuous application is one that performs work without waiting for any user input. For example, streaming a movie over the network.

- **Resource Intensive**: Does the application require significant resources to achieve adequate performance? For this dissertation, I limit the resources to network bandwidth, CPU, and memory usage. The main resource omitted was energy as the client hardware used for this thesis did not allow accurate measurements of battery consumption. In particular, the hardware did not support ACPI or have smart batteries. This omission is acceptable as conservation of energy, in a similar context, has already been demonstrated by Flinn [74].

Table 2.1 shows the four possible combinations of these two classifications along with examples of applications that satisfy each combination. I focus solely, in this dissertation, on resource intensive interactive applications. This restriction arose due to three reasons. First, non-resource intensive applications can be easily run on mobile devices. Hence, there is nothing interesting to investigate with these applications. Second, there is very little that can be done to support resource-intensive continuous applications as these applications tend to be bandwidth limited. It is possible to make these applications use less bandwidth by reducing the bitrate of the multimedia streams. However, these methods have already been heavily investigated by previous researchers[54, 144, 160, 220] and we can just reuse those techniques.

Finally, the remaining class of computationally-intensive interactive applications is interesting for a number of reasons. The first reason is that this class of applications has

	Resource Intensive	Non-Resource Intensive
Interactive	Language Translation Pattern Recognition Applications[a] 3D Augmented Reality[b] Speech Synthesis[c]	Calendar TODO List Email
Continuous	Streaming Media Voice Over IP	Clock on Screen Resource Monitors[d]

[a]E.g., face recognizers, speech recognizers, and optical character recognizers
[b]these are applications that can display a 3D display on a light heads up display worn by a mobile user
[c]an application that converts text into speech
[d]E.g., little applets that continuously show the bandwidth and CPU usage

This table shows examples of various applications that fit each of the 4 possible combinations of resource intensive and interactive. This dissertation concentrates solely on applications that are both resource intensive and interactive (the shaded quadrant).

Table 2.1: Application Taxonomy

not been well supported on mobile devices as they require more resources than is available on most mobile devices. Manufacturers usually create "lite" mobile-device optimized versions of these applications for use on mobile devices. However, these lite versions usually don't provide the full functionality and quality of the regular applications. The goal of this dissertation is to allow the regular applications to run effectively on mobile devices.

The second reason is that there are a large number of computationally-intensive applications that would be of great benefit to mobile users. These applications include language translators, speech recognizers, speech synthesizers, optical character recognizers, and augmented reality applications. Hence, this class of applications is an excellent choice for this thesis as they are currently not usable on most mobile devices even though they would be highly useful to mobile users.

2.3 Retargeting Applications for Cyber Foraging

In this section, I describe the RapidRe process that allows existing applications to be retargeted for cyber foraging.

2.3.1 Motivating Scenario

Joe is a new hire at the company. His manager wants him to retarget an existing adaptive language translator for use on handheld devices running Linux and X. Not being a language translation expert, Joe collaborates with domain experts in order to understand how to make language translators adaptive. As a result, Joe realizes that language translators *adapt by changing the data files that they use to perform the translation. For example, if resources are available, the translator will use a large more accurate data file that requires more memory and CPU cycles than the smaller less accurate data files.*

Joe decides to use the Pangloss-Lite [81] application as the base. He modifies Pangloss-Lite to dynamically change the data files that it uses based on the available resources. This is a tedious and iterative process as Joe potentially has to make extensive changes to the Pangloss-Lite source to perform new tasks such as detecting the current resource availability and deciding on the optimal data file to use. Rather than adding such functionality directly to Pangloss-Lite, Joe decides to use a common resource management layer. However, interfacing Pangloss-Lite with this layer proves to be a non-trivial task that still requires extensive changes to the Pangloss-Lite source.

Finally, Joe achieves the desired modification: Pangloss-Lite now uses an underlying resource management layer to adapt its behavior to the available resources. His manager is delighted and asks Joe to make a number of other applications, such as a speech recognizer, adaptive. These applications are completely different from Pangloss-Lite – many are also written in a different programming language. Joe must redo all the painstaking work that he put into Pangloss-Lite for each of these new applications.

However, help is on hand for Joe. Using the RapidRe process, Joe is able to reuse the knowledge he learnt when modifying Pangloss-Lite to quickly and easily modify the other applications. The methodology and tools provided by RapidRe makes it easy for Joe to take an existing application, written in an arbitrary language, that Joe knows very little about and quickly modify it to be an adaptive mobile application.

2.3.2 Solution Requirements

To make the above scenario a reality, RapidRe has to support the following requirements.

2.3.2.1 Be Language and Application Agnostic

An obvious design strategy for cyber foraging would require all applications to be written in a language that supports transparent fidelity adaptation and remote execution of procedures. Java would be an obvious choice for this language, though other possibilities exist. The modified language runtime system could monitor operating conditions, determine which procedures to execute remotely and which locally, and re-visit this decision as conditions change. No application modification would be needed. This language-based, fine-grained approach to remote execution has been well explored, dating back to the Emerald system [123] of the mid-1980s.

I rejected this strategy because of its restriction that all applications be written in a single language. An informal survey of existing useful applications (as described in Section 2.2.3) reveals no dominant language in which they are written. Instead, the preferred language depends on the existence of widely-used domain-specific libraries and tools; these in turn depend on the evolution history and prior art of the domain. For example, my validation suite, presented in Chapter 3, includes applications written in C, C++, Java, Tcl/Tk and Ada.

This decision to be language agnostic had three consequences.

1. **Application-aware adaptation**: First, it naturally leads to an application-aware adaptation solution where the application source code is required.

2. **Unable to use code-analysis techniques**: Next, it eliminated the use of fully automated code-analysis techniques because these tend to be language-specific. These analysis techniques also don't support many languages such as C++ and are also unable to identify the fidelity metrics and partitioning points of an application without higher-level guidance or language support. Hence, without automatic tool support, I have to manually modify applications to add runtime support for cyber foraging.

3. **Coarse-grained remote execution**: Finally, it led to a coarse-grained approach in which entire modules rather than individual procedures are the unit of remote execution. Without language support, every procedure would need to be manually examined to verify if remote execution is feasible, and then modified to support it. By coarsening granularity, I lower complexity but give up on discovering the theoretically optimal partitioning. This is consistent with my emphasis on reducing programmer burden and software development time, as long as an acceptable cyber foraging solution is still produced.

   I show, in Chapter 5 that the absolute performance of these applications, when using coarse-grained remote executed, is good in mobile environments. It may be possible to achieve better results using a finer-grained strategy. However, that strategy is not explored in this dissertation.

   Finally, because coarse-grained remote execution is performed at the modular level, I use a *Remote Procedure Call* (RPC) [30] model for the remote execution.

2.3.2.2 Support Rapid and Effective Retargeting

The mobile device market is highly dynamic. In particular, a large number of new devices appear on the market every year. For example, Federal Communications Commission data [69] reveals that there were 52 new cellphone models registered in 2000, 61 in 2001, 146 in 2002, 223 in 2003, 385 in 2004 and 55 in the first two months of 2005. This increase in new models is primarily driven by increasing consumer demand. For example, the worldwide cellphone market is estimated to reach 2 billion users by 2007 [226] and ∼700 million cellphones were shipped in 2004 compared to ∼500 million in 2003 [223]. Manufacturers thus release many different mobile device models, each with different feature sets and pricing, to ensure that they have an appropriate device for a prospective buyer.

These devices also have very short life cycles. It is not uncommon for a device to be considered "outdated" after 6 months, for it to not be sold in store after 1 year, and for the manufacturer to remove all support for the device after 2-3 years. This phenomena results in mobile devices having to sell well within 6 months. Otherwise, they end up being commercial failures.

This short device life and its implications for software development were dominant considerations in my design. My target context is a vendor who must rapidly bring to market a new mobile device with a rich suite of applications. Some applications may have been retargeted to older devices, but others may not. To attract new corporate customers, the vendor must also help them rapidly retarget their critical applications. The lower the quality of programming talent needed for these efforts, the more economically viable the proposition. Hence, in many companies, the task of porting code often falls to junior developers. However, the quality of the retargeted applications, in mobile environments, should not be sacrificed.

This leads to the central challenge of this dissertation: *How can junior software developers rapidly, easily, and effectively retarget large, unfamiliar applications for cyber foraging?* I assume that the application source code is available; otherwise, we cannot support arbitrary applications (the next requirement listed below). Even with source code, just finding one's way around a large body of code is time consuming. My design must help a developer rapidly identify the relevant parts of an unfamiliar code base and then help him easily create the necessary modifications for coarse-grained remote execution and fidelity adaptation. Obviously, the quality of the resulting retargeting must be good enough for serious use. In rare cases, a new application may be written from scratch for the new device. My design does not preclude this possibility, but I do not discuss this case further in this dissertation.

2.3.3 Summary of Requirements

The entire set of RapidRe requirements can be summarized as follows: RapidRe must support:

- Any application that fits the requirements listed in Section 2.2.3
- Any language (C, C++, Ada, Java, Tcl, etc.)
- Any developer (novice or expert)
- Quick retargeting of applications
- Excellent retargeted application performance (in terms of absolute runtime performance)

Satisfying all these requirements seems to be an impossible task. Fortunately, there is a solution.

2.3.4 Exploiting Application Commonalities

The key insight that allows a general solution to be created is that many computationally-intensive interactive applications share a common structure. This structure allows the development of RapidRe.

The first commonality is that all these applications have an *operation* model of execution. In particular, they receive input from a user and then execute some computation. The results of the computation are then presented to the user. This sequence of input, computation, and output is an operation.

The second commonality is that all these applications need to perform the same set of steps to adapt effectively in a mobile environment. They first need to measure the available resources in the environment. They then need to decide on the optimal application runtime settings, for the current operation, that best match the available resources and the current user preferences.

This commonality allows us to move common tasks into a common runtime layer. For this thesis, I use Chroma [20] as the runtime layer. In particular, this layer can perform the tasks of measuring the available resources and determining the optimal application runtime settings. This greatly simplifies the modifications that need to be made to applications to support cyber foraging – applications just have to provide the appropriate information to the runtime and receive the chosen runtime settings whenever they perform an operation. More precisely, applications have to specify just three pieces of information. These are the *parameters*, *fidelity variables*, and *tactics* of the application.

Parameters are precise application settings that affect the resource usage of the current operation. Given the current parameter settings, the runtime can determine the expected resource usage of the application. For example, for a language translation application, the size of the sentence being translated affects the resource usage. Applications could have one or more parameters. For example, a graphics application usually has multiple parameters such as the camera position, the lighting position, and the size of the model being rendered. The expected resource usage is then used by the runtime to pick the optimal application runtime settings. Narayanan [151] has demonstrated that accurate runtime parameter values are vital in correctly predicting application resource usage.

These runtime settings consist of fidelity variables and tactics. Fidelity variables are particular application settings that need to be set according to the available resources. For example, a graphics application may want the runtime to determine the optimal resolution to render the model at. Not all applications may have fidelity variables. Tactics, as explained in more detail in the next section, are an enumeration of the possible coarse-grained partitioning choices for the application. At runtime, Chroma will pick the optimal tactic and fidelity variable settings.

In Appendix A, I explain how Chroma uses parameters, fidelity variables and tactics to pick the optimal application runtime settings. In Section 2.5, I explain how an application can specify its parameters, fidelity variables, and tactics.

Finally, these applications are also designed, for code maintenance and modularity reasons, in very well-defined ways. For example, modules are cleanly separated from other modules and the GUI portions of the application are usually cleanly separated from the computational portions. This makes it possible for developers to quickly skim the application code and narrow their focus to just the required computational portions. I revisit this issue again in Section 4.7.1 after presenting the actual performance of RapidRe.

2.3.5 Solution: RapidRe

To address the requirements stated in Section 2.3.2, I developed a process called RapidRe. RapidRe consists of three main parts.

First, there is a "little language" called *Vivendi* for expressing the application-specific information relevant to cyber foraging. A developer examines the source code of an application and creates a Vivendi file called the "tactics file." The tactic file contains the parameters and fidelity variables of the application. It also contains the function prototype of each module deemed worthy of remote execution, and specifies how these modules can be combined to produce a result. Each such combination is referred to as a *remote execution tactic* or just *tactic*. For many applications, there are only a few tactics. In other words, the number of practically useful ways to partition the application is a very small fraction of the number of theoretical possibilities. A tactics file has to be created once per application. No changes are needed for a new mobile device.

The second part of RapidRe is a runtime system tuned for the target environment. For this dissertation, I use Chroma as the runtime system. By using a runtime, I am able to cleanly separate the application-specific adaptation information from the general mechanisms, such as resource monitoring and prediction, needed for adaptation. This greatly reduces the amount of code that needs to be added to each application. Applications can just query Chroma to discover the appropriate operation setting to use for the current operation. A common runtime system also makes it much easier to coordinate the adaptive behaviour of multiple applications running on the same device.

The third part is the Vivendi stub generator, which uses the tactics file as input and creates a number of stubs. Some of these stubs perform the well-known packing and unpacking function used in remote procedure calls [31]. Other stubs are wrappers for Chroma calls. Calls to stubs are manually placed in application source code by the developer.

Although not a tangible artifact, there is an implicit fourth component to my solution. This is a set of application agnostic instructions to developers to guide them in using the three solution components mentioned above. This includes documentation, as well as a checklist of steps to follow when modifying any application.

2.4 Using RapidRe

Figure 2.3 shows how the three parts of RapidRe are used as part of a 4-step process that allows developers to rapidly retarget applications. The four steps are as follows; first, the developer describes, using Vivendi and the help of a domain expert (explained below), the

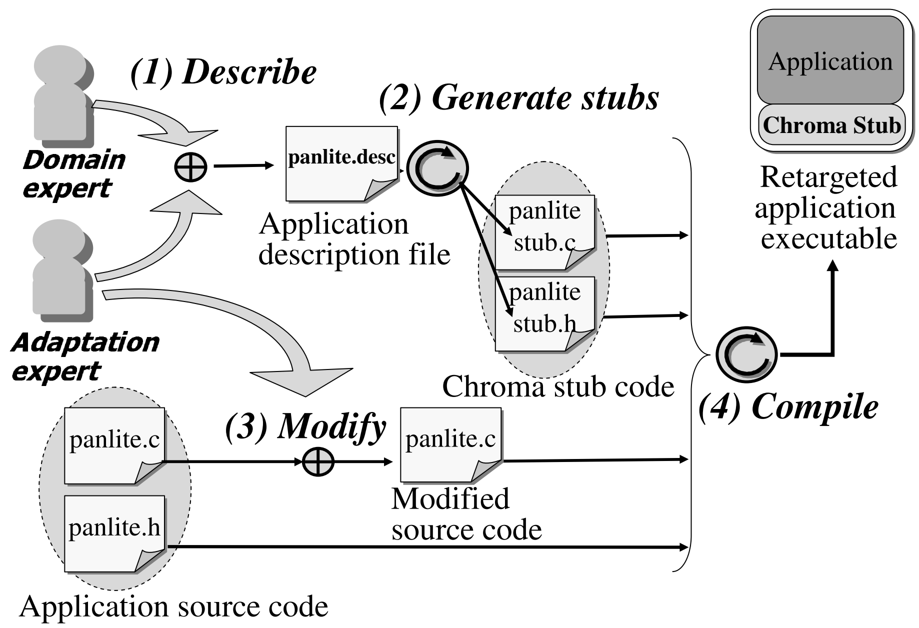


Figure 2.3: The 4 Stages of RapidRe

adaptive capabilities of the application to be retargeted. Second, the description is processed by the stub generator that generates most of the code needed to interface the application with the runtime system (Chroma). The stub generator also generates application-specific APIs that need to be manually inserted into the application. Third, the developer manually inserts these APIs, using a well-defined methodology, to create the client and server components of the application. Finally, the application is compiled together with the generated stub code and runtime libraries to create the final retargeted application. After this process is complete, the application will be able to connect to Chroma and participate in cyber foraging. I describe each of the four steps of RapidRe in more detail below.

2.5 Step 1: Creating the Tactics File

The first step in using RapidRe is describing the adaptive behaviour of an application using Vivendi. Vivendi was designed using a top-down approach – a large number of applications were examined before the Vivendi syntax was finalized. The goal was to create the simplest syntax that was sufficient for effectively describing a large number of computationally-intensive applications. I do not claim that the syntax is necessary. This goal of simplicity was to make it as easy as possible for even novice developers to use Vivendi. However, where necessary, more advanced syntax was also provided for developers who wished to exploit advanced application adaptation possibilities. In the rest of this section, I present the key components of Vivendi using examples that highlight the most common language

```
APPLICATION graphix;
OPERATION render;

IN int size DEFAULT 1000; // parameters
OUT float resolution FROM 0.0 TO 1.0; // fidelities

// RPC definitions
RPC step_1 (IN string input, OUT string buf1);
RPC step_2 (IN string input, OUT string buf2);
RPC step_3 (IN string buf1, IN string buf2,
            OUT string final);

// TACTIC definitions
// do step 1 followed sequentially by step 3
TACTIC do_simple = step_1 & step_3;

// do steps 1 & 2 in parallel followed by step 3
TACTIC do_all = (step_1, step_2) & step_3;
```

This is a complete example of a hypothetical graphics application, that renders 3D scenes, description using Vivendi. It is used as a working example throughout this dissertation and it is referenced multiple times in this Chapter and also in Chapter 5 and Appendix A. This description is intentionally kept simple (using names such as `step_1` etc. for the RPCs) as the point of this figure is to illustrate tactics file syntax.

Figure 2.4: Example Tactics File in Vivendi

usage patterns. The formal language specification is presented in Appendix B.

2.5.1 Basic Syntax

In this section, I present the basic Vivendi syntax. This is the syntax that has to be used for every application. Figure 2.4 shows an example of a tactics file description for a hypothetical graphics application. Each part of the description is explained in more detail below.

2.5.1.1 Application and Operation

First, the developer has to specify the name of the application using the `APPLICATION` tag. Next, the developer has to specify the operations that this application provides by using the `OPERATION` tag. Any specifications following an `OPERATION` tag is assumed to belong to that operation. This allows developers to easily describe applications with multiple operations, for example an application that provides both speech recognition and language translation, by just using multiple `OPERATION` tags to separate the information for each operation.

2.5.1.2 Parameters and Fidelities

Next, the developer has to specify the parameters and fidelity variables of the application. Recall, from Section 2.3.4, that parameters are application-specific values that are used to determine the application's resource usage. Fidelity variables, on the other hand, affect the quality and resource usage of the application. The precise setting for these variables is determined by Chroma at runtime.

Parameters and fidelity variables are specified in Vivendi using a syntax similar to describing variables in C. The only difference is that the `IN` keyword is used to denote parameters and the `OUT` keyword is used to denote fidelity variables. Vivendi supports all basic data types as shown in the formal syntax presented in Appendix B.

To specify the parameters and fidelity variables, I assume that developers are able to leverage the assistance of domain experts. In particular, I assume that the domain expert will be able to advise the developer, in very general terms, as to what the parameters and fidelities of the type of application being retargeted might be. For example, for graphics applications, the domain expert might say that the screen size, viewing position, camera position, and lighting position affect the resource usage of the application while the resolution is a dynamic quantity that should be set based on latency requirements. Given this broad description, the developer has to then find the exact variables in the application that provide this information and specify them as parameters and fidelity variables. I believe that this assumption accurately reflects reality where a developer, tasked with the retargeting task, is able to quickly obtain general information about an application obtained from either existing documentation or a colleague. The developer then has to use this general information to retarget the application.

2.5.1.3 Tactics

As stated earlier, using servers to augment the capabilities of limited mobile devices can result in significant performance improvements when executing computationally intensive applications. However, before applications can be remotely executed, it is necessary to determine an appropriate partitioning of those applications. This is non-trivial because there are potentially a very large number of ways to partition any large application. Fortunately, the use of coarse-grained remote execution greatly reduces this search space. For most applications, in practice the number of useful ways to partition them, at the modular level, is small. These small number of useful partitions are called tactics. In addition, I claim that tactics can be quickly and effectively specified by developers. I validate this claim in Chapter 4.

The power of tactics lies in their ability to concisely express the small number of useful partitions of an application. This dramatically reduces the search space for any dynamic adaptive runtime system, such as Chroma, that is attempting to pick an optimal application partitioning given the current resource availability and user preferences. However, before tactics can be used, it is necessary to clearly identify the type of remote execution that tactics are describing.

I again assume that the developer is able to use the assistance of a domain expert to determine if the application requires any special forms of remote execution. For example, the domain expert might say that for graphics applications, it is common to perform the rendering on the server. The server then ships back the rendered pixels to the client which displays it locally. The developer then has to use this general information to figure out the exact RPCs and tactics for the application.

In this section, I explain how tactics can be used to describe the coarse-grained remote partitionings of an application. Describing these partitions requires three components:

1. Description of each remote procedure that can be partitioned. For this dissertation, these are the procedures that perform the computational operations of an application. For example, the procedures that perform the rendering for a graphics application or the procedures that perform the language translation for a language translator.

2. Description of the possible ways to execute each RPC to successfully perform the operation. Each of these possible combinations is a different tactic.

3. Description of optional restrictions on assigning servers to individual RPCs in a tactic.

In the next few subsections, I describe the Vivendi syntax that is used to describe the three tactic components (as listed above).

RPCs RPCs are described using a syntax similar to declaring procedures in the C programming language. For example,

```
RPC do_it (IN int size, IN float type, IN string name,
           OUT string text);
```

specifies an RPC called `do_it` that has 3 input arguments (denoted by the keyword `IN`) called `size`, `type`, and `name` (which are an integer, float and string respectively). The RPC has an output argument (denoted by the keyword `OUT`) called `text` that is a string variable. Vivendi supports all basic data types and requires that RPC arguments be explicitly marked as either inputs or outputs by using the `IN` and `OUT` keywords respectively.

Tactic Plans Figure 2.5 shows how tactics can be described using Vivendi. Tactics can consist of sequential and / or parallel sequences of RPCs. Vivendi itself has no limitations on specifying arbitrary combinations of sequential and parallel RPCs. However, Chroma has limitations on the types of tactics that it can support.

```
// legal TACTIC declarations

// do RPCs a, b and c in sequence
TACTIC do_in_sequence = a & b & c;

// do RPCs a and b in parallel and then do c
TACTIC do_in_parallel = (a, b) & c;

// do RPCs a and b in parallel and then do c; repeat
TACTIC do_more_parallel = (a, b) & c & (a, b) & c;

// illegal TACTIC declarations

// do RPCs a, b, and c in parallel with no sequential stage after
TACTIC do_only_parallel = (a, b, c);

// parallel RPCs within a parallel stage. not a single parallel stage
TACTIC do_fanout = (a, (b, c, d), c);
```

This figure gives examples of legal and illegal tactic declarations.

Figure 2.5: Examples of Various Tactic Specifications

Even though there are no Vivendi limitations on describing sequential and parallel RPCs, Chroma only supports a subset of all possible tactic specifications. In particular, Chroma only supports one level of parallel RPCs. I.e., there cannot be any sequential or parallel stages within a parallel stage. In addition, all parallel stages must be followed by a sequential stage. This restrictions arose because no real application that was tested required more complicated tactics. If necessary, support for more complicated tactics can be added to Chroma. Figure 2.5 shows the types of tactics that are supported by Chroma. Chroma will try to ensure, to achieve best possible performance, that parallel stages (such as RPCs `a` and `b` in tactic `do_in_parallel`) use different servers.

Vivendi uses an implicit mechanism to pass arguments between RPCs in tactics. In particular, arguments are passed between RPCs if they share the same name and are correctly specified as either inputs or outputs of the RPC. For example, in Figure 2.4, the argument `buf1` is passed between RPCs `step_1` and `step_3` in tactic `do_all` as it is an output of RPC `step_1` and an input of RPC `step_3`. This mechanism has the key advantage of being simpler for developers to use – no additional syntax to specify the exact argument passing between RPCs in a tactic needs to be learnt and used.

```
// set up server groups 1 and 2
SERVER one = foo.cs.edu, bar.cs.edu;
SERVER two = mop.cs.edu;

// do RPC a on a group one server, then sequentially do RPC b on a
// group 2 server
TACTIC affinity_1 = a:one & b:two;

// do RPC a on server group 1. Tag its choice with the symbol h. do
// RPC b on the server with tag h in parallel with RPC c. Finish
// the tactic with RPC d.  RPCs c and d can use any available
// server (discovered by the service discovery module).
RPC d TACTIC affinity_2 = a:one@h & (b@h & c) & d;

// RPCs a, b and c will be done in parallel on the same server.
// There are no restrictions in picking the initial server. Note
// that this is just a demonstration of the syntax. In practice,
// doing three parallel stages on the same server would eliminate
// the benefits of doing those stages in parallel (as the three
// parallel stages would be sequentialized on the common server).
// The runtime will ensure that the common server picked can run
// RPCs a, b, and c.
TACTIC affinity_3 = (a@h, b@h, c@h) & d;

// RPC a has to be done locally and b has to be done remotely. The
// reserved keywords local and remote refer to the local and remote
// servers respectively.
TACTIC affinity_3 = a@local & b@remote;
```

This figure gives examples of server specifications that can be specified using Vivendi.

Figure 2.6: Server Specification Example

```
// tactic has only 1 RPC, a, which is a decomposable RPC. The split
// function is called split and the join function is called join
TACTIC single_decomp = %split:(a):join;

// similar to the first tactic except that after the data is split,
// it is processed by more than just a single RPC. In this case,
// the split data is first sent to RPC a and then it is sent in
// parallel to RPCs b and c. After b and c return, the split data is
// sent to RPC d which creates the final partial output.
TACTIC longer_decomp = %split:(a & (b, c) & d)):join;

// in this tactic, there are 2 decomposition stages (using RPCs a
// and c) and 2 normal sequential stages (using RPCs b and d).
// Note that the split and join functions are different for different
// decomposition stages.
TACTIC mixed_decomp  = %split1:(a):join1 & b & %split2:(c):join2 & d;
```

This figure gives examples demonstrating how data decomposable tactics can be specified using Vivendi.

Figure 2.7: Data Decomposition Examples

2.5.2 Advanced Syntax

In the following subsections, I describe more advanced Vivendi syntax that developers can use to specify advanced application characteristics.

2.5.2.1 RPC Server Specification

Vivendi allows server groups to be specified. These groups can then be used to constrain particular RPCs in a tactic to a particular precise set of servers (the server group specifies the set of servers) or to specify that particular RPCs must be run on the same servers as previous RPCs (the server group specifies the server dependencies between RPCs). I don't expect this syntax to be used normally but it is useful in cases where license servers are required or in cases where future RPCs can benefit from state or warm caches left behind by earlier RPCs. Figure 2.6 shows examples of specifying server constraints using Vivendi.

2.5.2.2 Data Decomposition

Some applications may also be able to benefit from extra servers by splitting the work necessary for an operation among multiple servers. The applications can do this by decomposing the input data, using application-provided code, into smaller chunks and shipping each chunk to a different remote server. The partial results are then recombined, again using application-provide code, to form the final output. This modality is called *data decomposition*. As will be shown in Section 5.5.3, data decomposition can result in substantial performance improvements.

Data decomposition is a useful optimization as a large number of applications, including optical character recognition, language translation, and speech synthesis, can benefit from data decomposition. They benefit because many of these application have input that can be subdivided into smaller independent chunks. These smaller chunks can then be executed in parallel and the partial results recombined to form the final output. This has the potential to result in big speedups for the application (the speedup is dependent on the number of chunks and the amount of parallelization in the entire chosen tactic). The only requirement is that the application data must be easily decomposable into independent chunks. In this section, the Vivendi syntax for describing data decomposition is presented. Applications use data decomposition by specifying a data decomposable tactic using the following syntax:

```
%split_function:(rpc_sequence):join_function
```

where `split_function` is the name of the application-supplied function that splits input data into smaller pieces, `rpc_sequence` is the sequence of RPCs that will process each piece of data, and `join_function` is the name of the application-supplied function that can recombine the partial results created by the `rpc_sequence` to generate the final result. The `split_function` and `join_function` must be application-supplied (written by either the application developer or the retargeting expert) as these functions are very data and application specific. Figure 2.7 shows examples of how data decomposition stages can be specified.

Chroma supports data decomposition as follows; When a data decomposable application is run, Chroma determines how many servers are available in the environment and provides the number (y) to the application. The application-supplied split function then breaks the input into n pieces (where $n <= y$). Note that n could be less than y as the application may decide that splitting the input into y pieces may result in pieces that are too small (the overhead of shipping the piece to a remote server is higher than the computational gains). Chroma then sends the n pieces, in parallel, to different servers to be executed. Chroma waits for all the n partial results to returns and then sends them to the application. The application-supplied join function combines the partial results to create the final result. I present results to show the benefits of decomposition in Section 5.5.3.

2.6 Step 2: Using the Stub Generator

After the tactics file has been created, the developer gives it to a stub generator that creates most of the interface code needed to interface the application with the targeted runtime system (Chroma in this case). The stub generator also creates application-specific APIs that need to be inserted into the application, as described in Section 2.7, to create the client and server components of the applications.

RapidRe is able to use a stub generator because the tactics file description presents all the information to make the application adaptive is concise and complete. The use of a stub generator is crucial in achieving the thesis of this dissertation as it allows us to hide the runtime details from developers (as shown in Section 4.7.1.1 and isolates applications from runtime and device changes. Each of these is described in more detail below.

2.6.1 Hiding the Runtime Details from the Developer

Manually integrating an application with a runtime system can be tedious and time consuming. The stub generator greatly reduces this burden by generating most of the code needed to interface an application with the targeted runtime – Chroma in this case. The generated code, in particular, includes all the tedious and easy to get wrong marshaling and buffer management code needed to send data between Chroma and the application. The developer only has to use the simple application-specific generated APIs. The details of how these simple APIs perform the actual communication with Chroma can remain a mystery to the developer. I describe the application-specific APIs, how they should be used, and also provide details of what these simple APIs are actually doing under the covers in Section 2.7.

2.6.2 Isolates the Application from Runtime Changes

RapidRe was developed to facilitate rapid application retargeting. However, as mentioned in Section 2.3.2.2, the mobile device market is highly dynamic. It is thus important that retargeted application require as little modification as possible when moved to a new device or runtime.

The stub generator is crucial in enabling this as all the runtime and device specific information is hidden in the generated stub code. The developer only sees application-specific interfaces that are completely runtime and device agnostic. These APIs also provide very generic adaptive functionality (as shown in the next section) and can thus be easily be mapped to almost any underlying adaptive runtime system.

Hence, even if the runtime or underlying operating system changes (e.g, from Linux to Symbian or even WinCE), it is possible (only possible and not certain as I have not verified this claim) that the retargeted application will not need to be modified. Only the stub generator will need to be modified to support the interfaces of the new runtime or OS. However, once this is done, previously retargeted applications just have to use the new generated stub code to support the new runtime or OS. This is a process that can be done in minutes, even for hundreds of applications, using automated build scripts.

Note however, that normal bugfixes, feature additions, and code changes to compile under specific OSes will still require manual application modifications. It may also be necessary to modify the application if the application's tactics description changes.

```
/* APIs to interface with adaptive runtime */
int graphix_render_register ( );
int graphix_render_cleanup ( );
int graphix_render_find_fidelity ( );
int graphix_render_do_tactics (char *input,
     int input_len, char *final, int *final_len);

/* APIs to set parameters and retrieve fidelity variables */
void graphix_render_set_size (int value);
float graphix_render_get_resolution ( );
```

Figure 2.8: Stub Generated Application-Specific Client Header File

2.7 Step 3: Inserting the APIs

In this section, I explain how developers can use the generated API calls to successfully create the client and server application components.

2.7.1 Client APIs

Figure 2.8 shows part of the generated client header file for the tactics file shown in Figure 2.4. It shows the generated calls to set parameter values and retrieve fidelity values and the four main generated API calls that have to be inserted into the application to create the client component. The four API calls are used as follows:

- `graphix_render_register` and `graphix_render_cleanup` are inserted at the start and end (wherever it exits) of the application respectively.
- `graphix_render_find_fidelity` call is inserted before the code that performs the operation. It tells the runtime to decide the operation setting for this operation. The developer must use the `graphix_render_set_size` call to set the value of the parameter before calling `graphix_render_find_fidelity`. After `graphix_render_find_fidelity` returns, the developer must read the runtime-determined fidelity value using the `graphix_render_get_resolution` call. These fidelity values should then be used to set the appropriate state for the operation. For example, the returned resolution value might need to be used to setup a rendering data structure.
- `graphix_render_do_tactics` is then inserted to perform the actual operation using the tactic selected by the runtime. The `graphix_render_do_tactics` argument list will contain all possible inputs and outputs for any tactic. This simplification allows one single API call to support any number of tactics. Finally, the

```
/* Server RPCs that need to be created */
void do_step_1 (char *input, int input_len,
                char *buf, int *buf_len);
void do_step_2 (char *input, int input_len,
                char *buf, int *buf_len);
void combined (char *input, int input_len,
               char *final, int *final_len);

/* Server APIs */
int service_init (int *argc, char ***argv );
void run_server ( );
```

Figure 2.9: Stub Generated Application-Specific Server Header File

developer must remove the code that used to perform the operation. Once all these steps are performed, the developer has successfully created the adaptive client component of the application.

If the application is data decomposable, the developer has to also create the split and join functions specified in the tactic description (as shown in Section 2.5.2.2). The stub generator will create all the data structures needed to store the partial data and results.

2.7.2 Server APIs

Figure 2.9 shows part of the generated server header file. It shows the RPC functions and generated APIs that the developer needs to be concerned about. To create the server component of the application, the developer must do the following: First, `service_init` must be inserted at the start of the application to initialize the server. The developer must then preserve any initialization code that is needed for the server to function. `run_server` is then inserted after the initialization code. This starts a non-terminating event loop that listens for RPC requests from clients. When it receives a request, it retrieves the argument list, calls the appropriate RPC, and returns the RPC output to the client.

Finally, the developer must create the RPC functions required by the server. These functions are usually easily created as the developer can reuse existing code (usually the code removed during client creation). The developer just has to a) use the RPC inputs to setup the appropriate state before the existing code is executed and b) ensure that the RPC outputs are correctly created after the code has finished executing.

2.8 Step 4: Compile, Link, and Run

The final step in the RapidRe process is to compile the client and server components of the application together with the generated code stubs (that provide the functionality for the application-specific APIs) and the Chroma libraries. This creates the final client and server application binaries that are completely cyber foraging enabled.

2.9 What's Missing?

Unfortunately, RapidRe doesn't solve all the problems associated with retargeting applications to work on mobile devices.

Application GUI : RapidRe allows developers to quickly and easily retarget the computationally intensive portions of the application. However, before these applications can be used on mobile devices, the user interfaces of these applications usually need to also be retargeted for the capabilities of the device. This is not an easy task as mobile devices usually have very small displays and very limited I/O capabilities. For example, a mobile wristwatch may have only a small 60 by 60 DPI display with no keyboard and only 5 input buttons whereas the application being retargeted was developed for a desktop computer with a 1600 by 1200 DPI display with a full keyboard and a mouse. In this dissertation, I do not address this problem of retargeting the user interfaces. Fortunately, the problem of GUI retargeting is being actively researched and there is ongoing work on automated user interface generation [67, 157] that I can defer to.

Application Hint Modules : For every operation, Chroma will determine the optimal application settings that best match the available resources and user preferences. Before it can do this, Chroma needs to know how choosing different fidelity variable settings and tactics affects the application's quality. For example, Chroma needs to know that choosing tactic `a` will result in a low quality output while choosing tactic `b` will result in a high quality output. This information is necessary for Chroma to choose an optimal tactic. Currently, this information has to be hand written by the developer in the form of a very short and simple C file (called an application hint module) that gets linked into Chroma. In the future, I plan to expand the Vivendi syntax to allow developers to specify the crucial pieces of information needed for these hint modules. This extension will allow the stub generator to automatically create the hint modules for the application.

Adding Fidelity Adaptation Support to Applications : If an application already supports fidelity adaptation, the developer can easily specify the fidelity variables that control the adaptation. However, if it doesn't, RapidRe does not provide any assistance in adding fidelity adaptation to the application. If fidelity adaptation is required, it has to be manually added, a potentially lengthy and complicated task, by the developer. For example, adding fidelity adaptation to a graphics application such that the resolution of the image can be changed dynamically.

2.10 Summary

In this chapter, I presented the characteristics of a cyber foraging environment. Namely, that mobile devices are small, light, and transient and that resource availability fluctuates immensely in mobile environments. I then identified the specific category of computationally-intensive interactive application that this thesis is concentrating on. Next, I identified the key requirements of any solution that attempts to make it easy to retarget these applications to support cyber foraging. Finally, I presented my solution, called RapidRe, that allows even novice developers to quickly, easily, and effectively retarget computationally-intensive interactive applications for cyber foraging.

Chapter 3

Applications Studied

To validate my dissertation, it is necessary to retarget a large number of real, useful mobile applications. In this chapter, I describe the applications used to validate my dissertation. The actual validation is presented in the subsequent chapters. Not all applications were used in every aspect of the validation. I used 10 real applications that have the following characteristics: a) They were all developed by other developers, b) they were not originally designed for cyber foraging, and c) they are all potentially useful mobile applications that fit the computationally-intensive interactive application classification presented in Section 2.2.3.

3.1 Face : Face Recognition

Face is a program that detects human faces in images. It was developed by Henry Schneiderman [191] and consists of $\approx 20,000$ lines of Ada code. For a given image, Face will draw a box around each face found in the image (as shown in Figure 3.1). It is representative of image processing applications of value to mobile users. For example, face detection could be a crucial component of a mobile device that allows the visually impaired to know who is near them. Surveillance personnel, with wearable computers, who use images to detect suspicious features in the environment are also likely to require this kind of application.

Face has a very simple tactics description file as shown in Figure 3.2. It has a single parameter ($input\_size$), no fidelity variables, and only one tactic ($just\_do\_it$) consisting of a single RPC ($detect\_frontal$). Even though Face has only one tactic, Chroma can use extra resources in the environment, through data decomposition (Section 2.5.2.2), to improve the performance of Face. For example, different parts of the image can be sent to different servers.

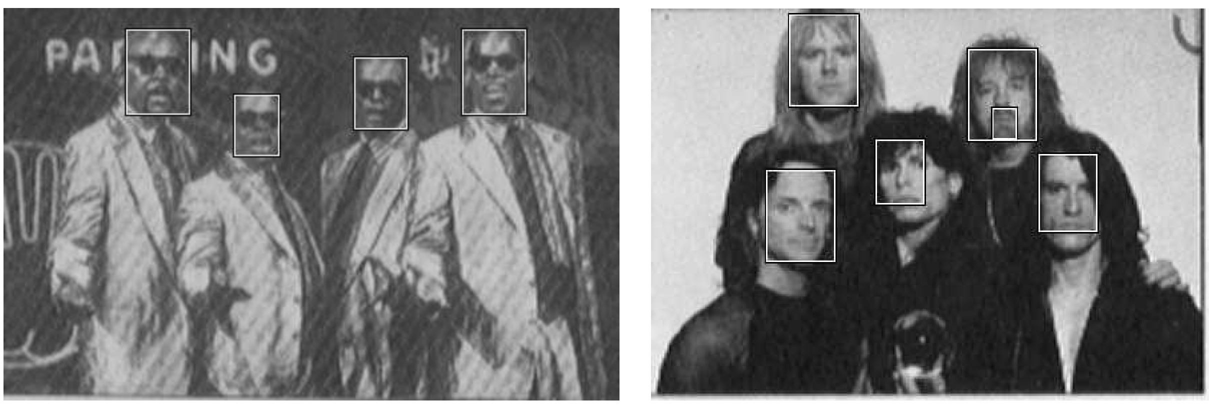


Figure 3.1: Detecting Faces in Images

```
APPLICATION face;
OPERATION   detect;

IN          int         input_size;

RPC detect_frontal (IN file in_image_name, OUT file out_image_name);

TACTIC just_do_it = detect_frontal;

// The data decomposable tactic is shown below. It
// replaces the normal tactic
// TACTIC just_do_it = %split:(detect_frontal):join;
```

Figure 3.2: Tactics Description for Face

```
APPLICATION flite;
OPERATION   synthesize;

IN  int  file_size  FROM 0 TO 1000000;

RPC synthesize (IN file text_file, OUT file wav_file);

TACTIC just_do_it = synthesize;

// The data decomposable tactic is shown below. It
// replaces the normal tactic
// TACTIC just_do_it = %split:(synthesize):join;
```

Figure 3.3: Tactics Description for Flite

3.2 Flite : Text to Speech Synthesis

Flite, short for Festival Lite, is a speech synthesis program developed by the University of Edinburgh and Carnegie Mellon University (the principal developers are Alan Black and Kevin Lenzo [32, 33]). It is written in C and consists of $\approx 570,000$ lines of code. Flite transforms text files into spoken output. This is useful as mobile device users may prefer having files read to them instead of having to view them on the devices' small displays. Spoken text is also a viable output modality that can be used in mobile devices designed for the visually impaired.

Flite, similar to Face, also has a very simple tactics description file (shown in Figure 3.3) with a single parameter ($file\_size$), no fidelity variables, and a single RPC and tactic. It can also use data decomposition to exploit extra resources in the environment. For example, different parts of the input file can be sent to different servers to be synthesized. The final audio output is then created by simply combining the synthesized partial outputs.

3.3 c2dfft : Image Filtering

c2dfft is a C program ($\approx 1,000$ lines of code), written by David O'Halloran [164], that calculates the 2-dimensional fast Fourier transform (FFT) of an array. Calculating FFTs of an array is a crucial component of any image filtering algorithm. In particular, image filters perform a number of FFTs followed by a transpose. Hence, this application represents image filters that mobile users are likely to use. For example, a mobile user takes a picture of a scene and wants his mobile device to detect if there are any houses in the scene. The first step in this process might involving detecting all the edges in the scene. This edge

```
APPLICATION c2dfft;
OPERATION   fft;

IN int size;

RPC fft_stage1 (IN int start, IN int stop, IN int width,
                IN FILE buffer, OUT FILE outbuf);

RPC transpose_stage1 (IN int start, IN int stop, IN int width,
                      IN FILE outbuf, OUT FILE outbuf1);

RPC fft_stage2 (IN int start, IN int stop, IN int width,
                IN FILE outbuf1, OUT FILE outbuf2);

RPC transpose_stage2 (IN int start, IN int stop, IN int width,
                      IN FILE outbuf2, OUT FILE out buf_final);

TACTIC do_it_all = %split1:(fft_stage1):join1 & transpose_stage1 &
                   %split2:(fft_stage2):join2 & transpose_stage2;
```

Figure 3.4: Tactics Description for c2dfft

detection performs a number of FFTs and transposes.

I use c2dfft to simulate a 2-pass image filtering operation. This consists of performing two consecutive FFTs where each FFT is followed by a transpose. The FFT algorithm independently processes each row of an array and is thus highly parallelizable. The transpose algorithm however, is very hard to parallelize as it processes the entire array at the same time. Overall, the two FFT operations make c2dfft highly data decomposable.

The tactics description file for c2dfft is shown in Figure 3.4. There is a single parameter called $size$ and separate RPC functions for the two FFT and transpose operations. Separate RPCs are used as the current stub generator requires that each RPC in a tactic must have a different name. The single tactic has two decomposable FFT stages (denoted by the $\%$ tag). Each of the FFT stages is followed sequentially by a non-decomposable transpose stage.

3.4 GLVU : 3D Model Viewing

GLVU is a graphics application, developed by the Walkthrough group at University of North Carolina [224], that allows users to virtually "walk through" (by changing the camera position) a three dimensional model of a building or object. GLVU is written in C++ using the OpenGL toolkit and consists of $\approx 25,000$ line of code.

Narayanan [152] added multi-fidelity support to GLVU. This allowed GLVU to dynamically change the resolution of the model being navigated. However, even with multi-fidelity support, GLVU was still not designed to be remotely executable. For this dissertation, I used the multi-fidelity version of GLVU.

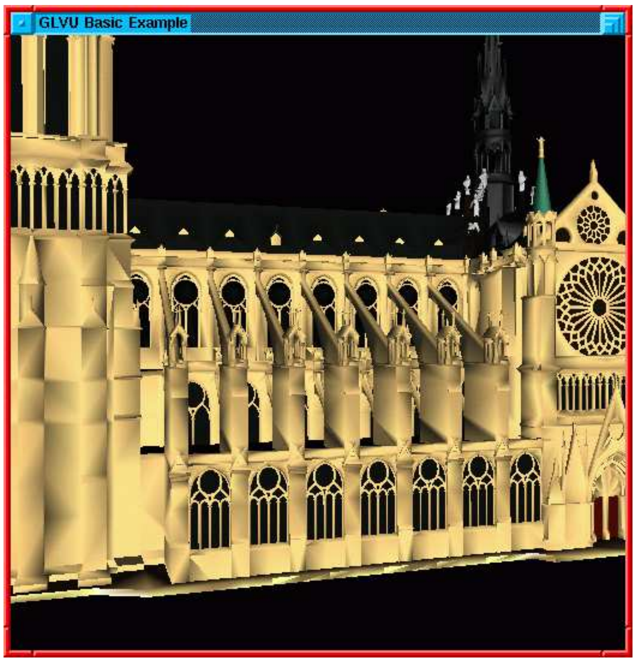
GLVU is representative of augmented reality [17] (AR) applications that could be useful to mobile users. In AR, a user looks through a transparent heads-up display connected to a wearable computer. Any displayed image appears to be superimposed on the real-world scene before the user. AR thus creates the illusion that the real world is visually merged with a virtual world. AR has already proved useful in domains such as tourist guides [70], power plant maintenance [62], architectural design [228], and computer-supported collaboration [29].

Figure 3.5 shows the tactics description for GLVU. GLVU has a large number of parameters (22) that affect its resource usage. This is because every application value that affects the viewing position and the size of the model has an effect on the resource usage of GLVU. GLVU has a single fidelity variable ($resolution$) that determines the resolution of the model. Changing the resolution changes the number of polygons that are used to render the 3D model. The lower the number of polygons used to render the model, the lower the CPU and memory usage. However, lowering the number of polygons greatly changes the quality of the model as perceived by the user. Figure 3.6 shows the effect of changing the resolution in GLVU.

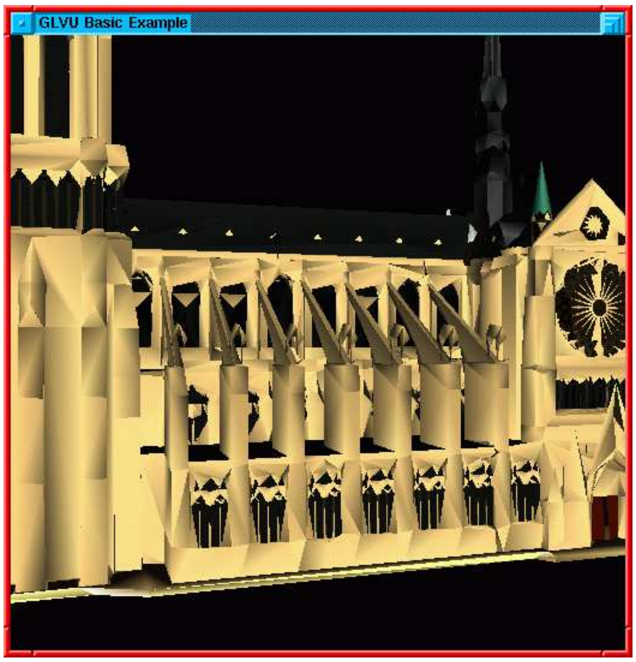
Finally, GLVU has a single tactic consisting of just one RPC. The argument list for the RPC is fairly long as a number of values need to be sent to the application server before the current scene can be rendered. These values state the exact position of the user and the current model settings.

```
APPLICATION     glvu;
OPERATION       draw;

IN    double            polygons     FROM 0     TO infinity;
IN    double            width        FROM 0     TO infinity;
IN    double            height       FROM 0     TO infinity;
IN    double            min_x        FROM 0     TO infinity;
IN    double            min_y        FROM 0     TO infinity;
IN    double            min_z        FROM 0     TO infinity;
IN    double            max_x        FROM 0     TO infinity;
IN    double            max_y        FROM 0     TO infinity;
IN    double            max_z        FROM 0     TO infinity;
IN    double            eye_x        FROM 0     TO infinity;
IN    double            eye_y        FROM 0     TO infinity;
IN    double            eye_z        FROM 0     TO infinity;
IN    double            vrp_x        FROM 0     TO infinity;
IN    double            vrp_y        FROM 0     TO infinity;
IN    double            vrp_z        FROM 0     TO infinity;
IN    double            vup_x        FROM 0     TO infinity;
IN    double            vup_y        FROM 0     TO infinity;
IN    double            vup_z        FROM 0     TO infinity;
IN    double            yfov         FROM 0     TO infinity;
IN    double            aspect       FROM 0     TO infinity;
IN    double            near         FROM 0     TO infinity;
IN    double            far          FROM 0     TO infinity;

OUT   double            resolution   FROM 0     TO 1;

RPC glvu_draw (IN uint w, IN uint h, IN string filename,
             IN float eye_x, IN float eye_y, IN float eye_z,
             IN float lookat_x, IN float lookat_y, IN float lookat_z,
             IN float up_x, IN float up_y, IN float up_z,
             IN float yfov, IN float aspect, IN float ndist,
             IN float fdist, IN uint numpolygons, OUT ustring buf);

TACTIC do_render = glvu_draw;
```

Figure 3.5: Tactics Description for GLVU



Full Resolution (1.0)



Low Resolution (0.1)

Images obtained from Dushyanth Narayanan

Figure 3.6: Effect of Resolution on GLVU Quality

3.5 GOCR : Optical Character Recognition

GOCR is an optical character recognizer (OCR) developed by Joerg Schulenburg [192] that is written in C ($\approx 30,000$ lines of code). It takes as input images containing characters in them (for example, the image shown in Figure 3.7). It then produces an ascii text file containing all the characters found in the image. Optical character recognition is useful for mobile users. For example, a foreign traveller can take an image of a street sign, perform OCR on it to extract the words, and then translate the words into a known language using a language translator.

GOCR has a very simple tactics description file (shown in Figure 3.8) consisting of a single parameter ($image\_size$) and a single tactic with just one RPC. However, GOCR can benefit from data decomposition. In particular, each line of text in the input image can be sent to a different server to be processed.

3.6 Janus : Speech Recognition

Janus is a speech recognition program that can identify spoken text in an audio file (in wav format). This is useful to mobile users as converting speech to text is required for any user

Xfig−font – generated for OCR testing – Mai2001 JS using fig2dev −L png
ABCDEF abcdef, MNOPQR mnopq XYZÄÅÆÖÜ. !?−+=/ " ’’ ft ff
GHIJKL ghijkl; STUVWr stuvw xyz (01234 56789) <> &$
ABCDEF abcdef, MNOPQR mnopq XYZ. !?−+=/ " ” ft ff
GHIJKL ghijkl; STUVWr stuvw xyz (01234 56789) <> &$
ABCDEF abcdef, MNOPQR mnopq XYZ. !?−+=/ " ’’ ft ff
GHIJKL ghijkl; STUVWr stuvw xyz (01234 56789) <> &$
ABCDEF abcdef, MNOPQR mnopq XYZ. !?−+=/ " ’’ ft ff
GHIJKL ghijkl; STUVWr stuvw xyz (01234 56789) <> &$
ABCDEF abcdef, MNOPQR mnopq XYZ. !?−+=/ " ” ft ff
GHIJKL ghijkl; STUVWr stuvw xyz (01234 56789) <> &$
ABCDEF abcdef, MNOPQR mnopq XYZ. !?−+=/ " ′′ ft ff
GHIJKL ghijkl; STUVWr stuvw xyz (01234 56789) <> &$
Special chars: àáâãäåæçÀÅÆ ß &$#% øØ«»¢µ

Figure 3.7: Example Image File that GOCR Can Process

```
APPLICATION gocr;
OPERATION   recognize;

IN  int  image_size  FROM 0 TO 1000000;

RPC recognize (IN file image_file, OUT file out_text_file);

TACTIC just_do_it = recognize;

// data decomposable tactic
// TACTIC just_do_it = %join:(recognize):split;
```

Figure 3.8: Tactics Description for GOCR

```
APPLICATION        janus;
OPERATION          recognize;

IN   int           file_size;

OUT  int           model_size;

RPC  Full          (IN string model, IN model_size,
                    IN FILE infile, OUT string recognition);

RPC  Hybrid_Stage1 (IN string model, IN model_size,
                    IN FILE infile, OUT FILE intermediate);

RPC  Hybrid_Stage2 (IN string model, IN model_size,
                    IN FILE intermediate, OUT string recognition);

TACTIC Janus_Full   =  Janus_Full;
TACTIC Janus_Hybrid =  Hybrid_Stage1@local & Hybrid_Stage2@remote;
```

Figure 3.9: Tactics Description for Janus

interface that allows users to speak their commands to the device.

Janus is already written as a client-server application. The client (written by Flinn [76]) is written in C (≈ 1000 lines of code) using the Motif toolkit. The server was developed by Carnegie Mellon University and the University of Karlsruhe. It was written using a mix of Tcl/Tk and C and consists of $\approx 125,000$ lines of code. Even though Janus is a client-server application, it is not adaptive and does not change fidelity or server location dynamically.

Figure 3.9 shows the tactics description for Janus. The $recognize$ operation has a single parameter, $file\_size$, and a single fidelity variable, $model\_size$. $model\_size$ is used by Chroma to tell the client whether to use a large or small language model to perform the recognition. The larger the model, the more accurate the recognition. However, larger models require more memory and CPU cycles. The $recognize$ operation has three RPCs ($Janus\_Full$, $Janus\_Hybrid\_Stage1$, and $Janus\_Hybrid\_Stage2$) that are combined to form two tactics ($Full$ and $Hybrid$). The $Full$ tactic performs the recognition, either locally or remotely, in one step. The $Hybrid$ tactic is an energy efficient recognition, using functionality added by Flinn, that performs the first part of the recognition locally ($Janus\_Hybrid\_Stage1$) and the second remotely ($Janus\_Hybrid\_Stage2$). Janus can use data decomposition as the the input wav file can be split into independent bits (using silence detection techniques) that can be recognized on different servers.

```
APPLICATION music;
OPERATION categorize;

IN int nsamples FROM 0 TO 1000000;
IN int freq;

RPC categorize (IN string packed_input_samples,
                OUT string packed_output_results);

TACTIC do_it = categorize;
```

Figure 3.10: Tactics Description for Music

3.7 Music : Identification of Music Pieces

Music is a music recognition program developed by Yan Ke, Derek Hoiem, and Rahul Sukthankar [129]. Given a short sample of music, Music will identify other pieces of music that contain the sample. This kind of sample identification application is useful to mobile users. For example, a lost mobile user may take a photo of his current location. This photo is matched against other images in a database to figure out where the user is. Music is an example of such an application.

Music is already designed as a client server application with the client written in Java (≈ 2000 lines) and the server in C++ (≈ 7000 lines). However, it was not designed to adapt in a mobile environment and cannot dynamically change servers to adapt to the resource conditions or user preferences. The tactics description of Music is shown in Figure 3.10. Music has two parameters ($nsamples$ and $freq$) and has a single tactic with one RPC.

3.8 Panlite : Natural Language Translation

Panlite (short for Pangloss-Lite) is a natural language translator that converts Spanish to English and vice versa. Panlite was developed by Robert Frederking and Ralf Brown and consists of $\approx 150,000$ lines of C++ code. Table 3.1 shows examples of the translations performed by Panlite. Language translation is a useful application for mobile users as demonstrated by the motivating example in Section 2.1.

Panlite translates sentences by using up to three different translation engines (the glossary, dictionary, and example-based machine translation (EBMT) engines). Each of these translation engines use different algorithms, can be run independently, and produce different quality partial outputs. The EBMT engine has the highest quality output followed by the glossary engine, and finally the dictionary engine. These partial outputs are then com-

Original Spanish	Translated English
Muchas veces dispararon contra aldeas	Often fired versus villages
También llevaron refuerzos hasta la frontera, incluidas unidades de tanque	Only take reinforcements until the border, including units for tank
En enero de 1993, la composición de la FPNUL era la siguiente	In January of 1993, the composition of the [FP-NUL] was the next :H
La razón principal es que desde hace varios años el Gobierno del Líbano no ha reembolsado a los propietarios el valor de los terrenos y locales utilizados por la FPNUL	The main discretion exist that ever since various years The Lebanon Government do not consider possess to reimburse to the owned the million from the terrestrial and premises utilized by you [FPNUL]

Table 3.1: Examples of Converting Spanish to English with Panlite

bined by a language modeler, that factors in the quality of each engine, to create the final translation.

Figure 3.11 shows the tactics description for Panlite. The only parameter is $nwords$ and there are no fidelity variables. There are 4 RPCs ($server\_gloss$, $server\_dict$, $server\_ebmt$, and $server\_lm$) corresponding to the three translation engines and the language modeler (lm). There are seven tactics as there are seven ways to translate a sentence. This corresponds to the seven ways of combining the RPCs such that a sentence is processed by at least one translation engine before the final translation is produced by the language modeler. To facilitate this, the $server\_lm$ RPC has one input for each translation engine. The stub generator will set any unused inputs to NULL automatically.

Each of these seven tactics use different amounts of resources and have different quality outputs. The lowest quality tactic is the $dict$ tactic where the sentence is processed only by the dictionary engine. The highest quality tactic is the $gloss\_dict\_ebmt$ tactic where all three engines are executed in parallel. The partial results from the three engines are then combined by the language modeler. Panlite is also data decomposable as individual lines can be translated on different servers.

3.9 Radiator : Lighting for 3D Models

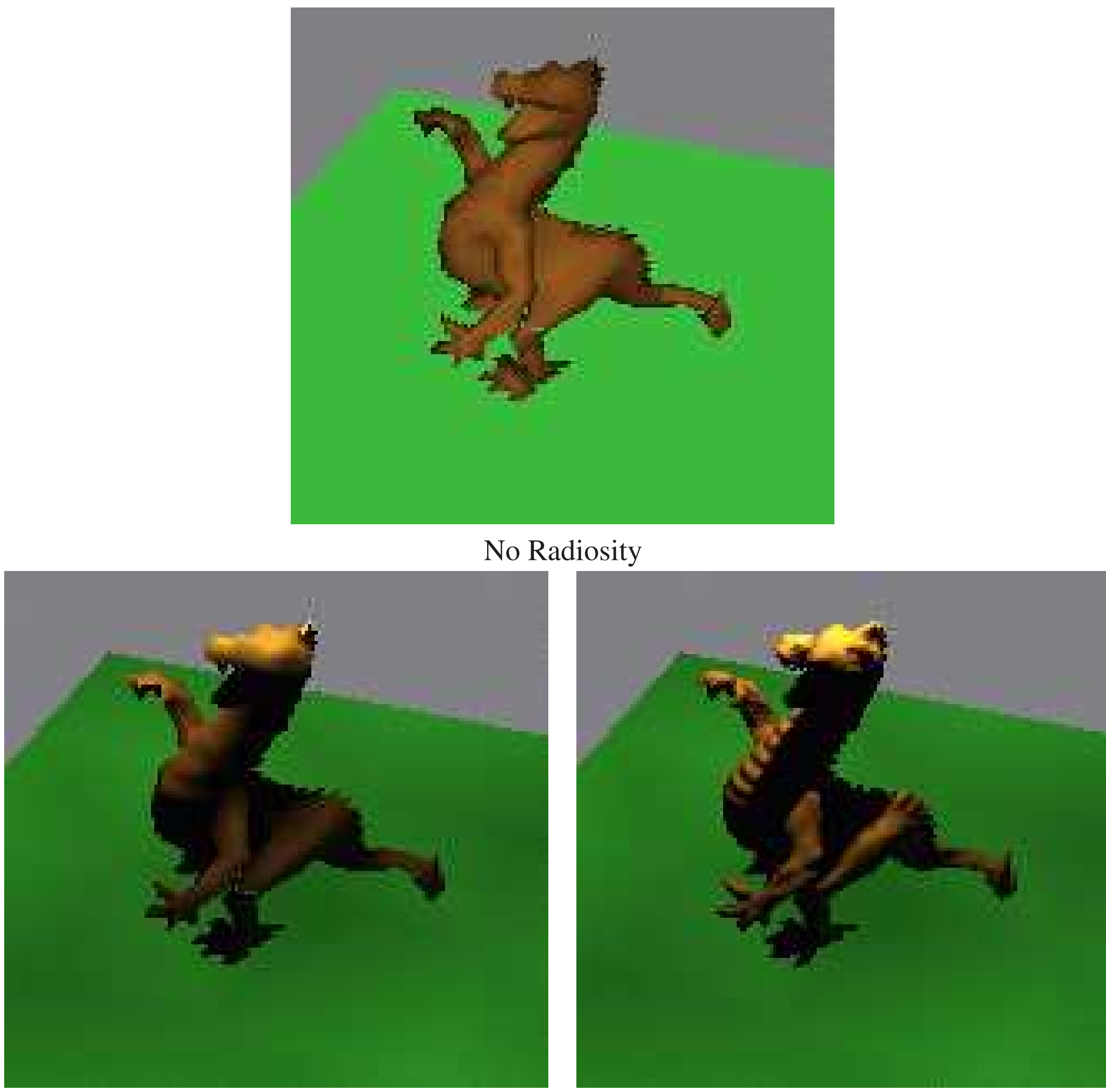
Radiator is a program, written in C++ ($\approx 65,000$ lines of code) using the OpenGL library by Andrew Willmott [233], that performs radiosity computations on 3D models. Radiosity [49] computations colour and shade a 3D scene according to the light sources present in the scene. This computation needs to be redone every time the scene changes, i.e., when objects or light sources are added, removed, or modified. However, radiosity is view-independent and does not need to be recomputed when the camera position or orientation changes. Radiosity is useful for augmented reality applications as it adds the proper lighting details to the scene. Since radiosity computations are extremely CPU and memory

```
APPLICATION     panlite;
OPERATION       translate;

IN  int  nwords  FROM 0  TO infinity DEFAULT 1;

RPC server_dict  (IN string line, OUT string dict_out);
RPC server_gloss (IN string line, OUT string gloss_out);
RPC server_ebmt  (IN string line, OUT string ebmt_out);
RPC server_lm    (IN string line, IN string ebmt_out,
                  IN string dict_out, IN string gloss_out,
                  OUT string translation);

TACTIC gloss           = server_gloss & server_lm
TACTIC dict            = server_dict & server_lm;;
TACTIC ebmt            = server_ebmt & server_lm;
TACTIC gloss_dict      = (server_gloss, server_dict) & server_lm;
TACTIC gloss_ebmt      = (server_gloss, server_ebmt) & server_lm;
TACTIC dict_ebmt       = (server_dict, server_ebmt) & server_lm;
TACTIC gloss_dict_ebmt = (server_gloss, server_dict, server_ebmt) &
                          server_lm;
```

Figure 3.11: Tactics Description for Panlite

intensive, Radiator already has built-in fidelity adaptation support. In particular, the resolution of the model can be changed dynamically. Figure 3.12 shows an example of the multi-fidelity radiosity computations performed by Radiator.

The tactics description for Radiator is shown in Figure 3.13. There is one parameter ($size$) and two fidelity variables ($resolution$ and $algorithm$). The $resolution$ changes the number of polygons used to render the scene. The lower the number of polygons, the lower the output quality but less CPU and memory is consumed. The $algorithm$ fidelity variable determines which algorithm is used to perform the radiosity computation. Different algorithms may result in different output qualities. They may even have the same quality but use different amounts of resources. For example, the $progressive$ algorithm and the $hierarchical$ algorithms result in the same quality output. However, the $progressive$ algorithm uses more CPU cycles while the $heirarchical$ algorithm uses more memory. Radiator has a single tactic comprising of a single RPC.



No Radiosity

Low Quality (0.1)

High Quality (1.0)

The top image shows the image before any radiosity computations: we see that the lighting has no effect on the observed image. The next two images show the same image, after radiosity computation, at low fidelity (0.1), and at high fidelity (1.0). The light source is above the dragon in all three cases.

Figure 3.12: Radiosity: Adding Lighting to 3D Models

```
APPLICATION radiator;
OPERATION   radiosity;

IN   int     size  FROM 0 TO 1000000;

OUT  float   resolution FROM 0.0 TO 1.0;

OUT  string  algorithm;

RPC radiosity (IN file scene_file, IN float resolution,
               IN string algorithm, OUT file rendered_image);

TACTIC just_do_it = radiosity;
```

Figure 3.13: Tactics Description for Radiator

The top left image is the original 2D image of the Carnegie Mellon University University Center. The remaining three images are screen captures of the 3D scene created by PopUp from that single image. Additional 2D images would improve the quality of the 3D scene. Images obtained from Derek Hoiem.

Figure 3.14: Example of PopUp Creating 3D Scenes from 2D Images

```
APPLICATION popup;
OPERATION vrml;

IN INT num_images;

RPC test_directory_s (IN STRING cluster_density, IN STRING image_dir,
                      IN STRING vert_density, IN STRING horz_density,
                      IN STRING testseg, IN STRING varargin);

TACTIC do_it = % split:(test_directory_s):join;
```

Figure 3.15: Tactics Description for PopUp

3.10 PopUp : Creation of 3D Scenes from 2D Images

PopUp creates 3D VRML [227] (an industry standard 3D scene specification language) scenes from 2D images. This can be used by mobile devices with a camera to create 3D scenes for use with augmented reality applications. PopUp was written in Matlab ($\approx$ $20,000$ lines of code) by Derek Hoiem [107]. Figure 3.14 shows examples of PopUp at work.

Since Matlab is too large for most mobile devices, I created simple Perl and Python client interfaces (less than 100 lines of code each) to PopUp. Both these interfaces were identical in functionally. They just read a list of files supplied by the user and submitted a request to the PopUp application to process those files. Creating these simple client interfaces allowed us to a) avoid needing Matlab on mobile devices, and b) test RapidRe with more languages. Figure 3.15 shows the tactics description for PopUp. PopUp has a single parameter (num_images) and just one tactic with a single RPC. However, the tactic is data decomposable as PopUp can process multiple 2D images in parallel to create the final VRML 3D scene.

3.11 Summary

In this Chapter, I described the applications used to validate this dissertation. Table 3.2 gives an overview of these applications. The applications covered a broad range of functionality ranging from language translation to face detection to creating lighting models for 3D displays. They also varied tremendously in their sizes and in the languages they were written in. Finally, the applications covered a broad range of parameters, fidelities, and tactics. In the next two chapters, I show that it possible for novice developers to quickly, easily, and effectively retarget these application for cyber foraging.

Application	Lines of Code	Number of Files	Language	Fidelities	Parameters	RPCs	Total RPC Args	Tactics	Decomposable?
Face (Face Recognizer)	20K	105	Ada w/C interface	0	1	1	2	1	Yes
Flite (Text to Speech)	570K	182	C	0	1	1	2	1	Yes
c2dfft (Image Filtering)	570K	182	C	0	1	1	2	1	Yes
GLVU (3D Visualizer)	1K	155	C++, OpenGL	1	15	1	18	1	No
GOCR (Char. Recognizer)	30K	71	C++	0	1	1	2	1	Yes
Janus (Speech Recognizer)	126K	227	C, Tcl/Tk, Motif	1	1	3	9	2	Yes
Music (Music Finder)	9K	55	Java, C++	0	2	1	2	1	No
Panlite (Lang. Translator)	150K	349	C++	0	1	4	11	7	Yes
Radiator (3D Lighting)	65K	213	C++, OpenGL	2	1	1	4	1	No
PopUp (3D Scene Generation)	20K	213	Perl, Python, Matlab	2	1	1	4	1	Yes

Table 3.2: Overview of the Applications Used to Validate This Dissertation

Chapter 4

Validation : Easy To Retarget

In this Chapter, I provide validation that RapidRe makes it easy for application developers to retarget large computationally intensive applications for use on resource limited mobile devices.

4.1 Success Criteria

Before this validation can be performed, it is necessary to determine what the success criteria are. For this thesis, RapidRe is successful if it allows developers to do the following:

- *Face complex applications confidently with little training*. Less required training is always better, of course, but some training will be needed before a developer can use my solution. About an hour of training is acceptable in commercial settings, and is probably close to the minimum time needed to learn anything of substance.

- *Modify complex applications quickly*. It is not easy to become familiar with the source code of a complex new application, and then to modify it for adaptation and cyber foraging. Based on my own experience and that of others I expect the typical time for this to be on the order of multiple weeks. Shortening this duration to a day or less would be a major improvement.

- *Modify complex applications with few errors*. Because programming is an error-prone activity, it is unrealistic to expect a developer to produce error-free code with my solution. A more realistic goal is a solution that avoids inducing systematic or solution-specific coding errors by developers. The few errors that do occur should only be ordinary programming errors that are likely in any initial coding attempt.

- *Produce modified applications whose quality is comparable to those produced by an expert*. When fidelity and performance metrics are carefully examined under a variety of cyber foraging scenarios, the adaptive applications produced by developers using my solution should be indistinguishable from those produced by an expert.

4.2 Validation Plan

The primary goal of this validation was to assess how well RapidRe meets the success criteria laid out in Section 4.1. A secondary goal was to gather detailed process data to help identify a) the reasons for RapidRe's success or failure, and b) areas for future research. To effectively satisfy both goals, I performed a laboratory-based user study that combined well-established user-centric and system-centric evaluation metrics. I used user-centric metrics for programmers such as ease-of-use, ease-of-learning, and errors committed [196] and system-centric metrics such as application latency and lines of generated code.

The laboratory study consisted of two parts. In the first part, novice developers modified a variety of real applications for cyber foraging. I used novice (and not regular) developers as a) this was a harder goal to achieve, and b) novice developers are more likely to be tasked with retargeting work at companies. I describe the experimental setup for this part in Section 4.2.1 and report its results in Sections 4.3 to 4.5. In the second part, I compared the performance of these modified applications to the performance of the same applications when modified by an expert. I describe this part and report its results in Section 4.6

4.2.1 User-Centric Evaluation

In this section, I describe the setup for the user study conducted to evaluate the effectiveness of RapidRe. Following the lead of Ko et al. [135] and Klemmer et al [134], I took user-centric evaluation methods originally developed for user interface investigations and adapted them to the evaluation of programming tools.

4.2.1.1 Control Group

In designing the user study, a major decision was whether to incorporate a control group in my design. When there is substantial doubt about whether a tool or process improves performance, it is customary to have one condition in which the tool is used and a control condition where subjects perform the task without the tool. This allows reliable comparison of performance. However, the practicality and value of control groups is diminished in some situations. For example, it is difficult to recruit experimental subjects for more than a few hours. Further, the value of a control group is negligible when it is clear to task experts that performing a task without the tool takes orders of magnitude longer than with it.

My own experience, and that of other mobile computing researchers, convinced us that modifying real-world applications for adaptive mobile use is a multi-week task even for experts. Given this, my goal of one day is clearly a major improvement. Running a control condition under these circumstances would have been highly impractical and of little value. I therefore chose to forego a control group.

Application	Lines of Code	Number of Files	Language	Fidelities	Parameters	RPCs	Total RPC Args	Tactics	Decomposable?
Face (Face Recognizer)	20K	105	Ada w/C interface	0	1	1	2	1	Yes
Flite (Text to Speech)	570K	182	C	0	1	1	2	1	Yes
GLVU (3D Visualizer)	1K	155	C++, OpenGL	1	15	1	18	1	No
GOCR (Char. Recognizer)	30K	71	C++	0	1	1	2	1	Yes
Janus (Speech Recognizer)	126K	227	C, Tcl/Tk, Motif	1	1	3	9	2	Yes
Music (Music Finder)	9K	55	Java, C++	0	2	1	2	1	No
Panlite (Lang. Translator)	150K	349	C++	0	1	4	11	7	Yes
Radiator (3D Lighting)	65K	213	C++, OpenGL	2	1	1	4	1	No

This figure is similar to Figure 3.2. The main difference is that PopUp and c2dfft are omitted as they were not used for the user study.

Table 4.1: Summary of Applications Used for User Study

4.2.1.2 Test Applications

I used eight of the ten applications (the remaining two were obtained after the user study was conducted) described in Chapter 3 for this user study. The applications were: *GLVU* (Section 3.4), a virtual walkthrough application that allows users to navigate a 3D model of a building; *Panlite* (Section 3.8), an English to Spanish translator; *Radiator* (Section 3.9), a 3D lighting modeler; *Face* (Section 3.1), a face recognition application; *Janus* (Section 3.6), a speech recognizer; *Flite* (Section 3.2), a text to speech converter; *Music* (Section 3.7), an application that records audio samples and finds similar music on a server; and *GOCR* (Section 3.5), an optical character recognizer. Table 4.1 shows the salient characteristics of these eight applications.

None of these applications was written by us. Some of these applications, such as GLVU, Janus, and Radiator, already had support for fidelity adaptation. Other applications, such as Music, and Janus, were already designed as client – server applications. However, none of the eight applications were designed for use in dynamic mobile environments where the exact runtime operation setting will change as the environment or user preference changes. As Table 4.1 shows, the applications ranged in size from 9K to 570K lines of code, and were written in a wide variety of languages such as Java, C, C++, Tcl/Tk, and Ada. The application GOCR, was used only for training participants; the others were assigned randomly.

Participant	Application 1	Application 2	Application 3
1	GLVU	Face	–
2	Panlite	Janus	–
3	GLVU	Janus	Music
4	Panlite	Face	Radiator
5	GLVU	Music	–
6	Panlite	–	–
7	GLVU	Janus	–
8	Panlite	Flite	–
9	GLVU	–	–
10	Panlite	Radiator	–
11	Flite	Face	–
12	Radiator	Flite	–
13	Music	–	–

The assignment of applications to participants is shown in this table. Each participant was given an initial application (Application 1). Some participants returned, on a different day, for a second application (Application 2). Two participants (3 and 4) even returned, again on a different day, to modify a third application (Application 3).

Table 4.2: Assignment of Participants to Applications

4.2.1.3 Participants and Setup

In many companies, the task of porting code falls to junior developers. I modeled this group by using undergraduate seniors majoring in computer science. In addition, I used a group size large enough to ensure the statistical validity of my findings. While the exact number depends upon the variability within the participants and the overall size of the effects, widely accepted practices recommend between 12 and 16 users [158]. I used 13 participants, which falls within this range and represents the limit of my resources in terms of time (six hours were needed for each data point).

The participants were all between 18 to 25 years of age. There were 12 male and 1 female participant. 12 participants were students at Carnegie Mellon University. The remaining participant was a student at the University of Pittsburgh. 11 of the participants was computer science majors. The remaining 2 were electrical engineering majors. 11 of the participants were seniors. The remaining 2 were masters students (in the 5th year senior thesis program).

The selection criteria required them to know C programming and be available for a contiguous block of six hours. None of them were familiar with my research, any of the tools under development, or any of the test applications. Table 4.2 shows the assignment of participants to applications. As the table shows, several participants returned for additional applications. In keeping with standard human-computer interaction (HCI) practice, I counter-balanced the assignment of participants to applications to avoid any ordering effects. Using the same participants to retarget multiple applications allowed us to investigate learning effects and to determine whether my one-time training was adequate. Each application was retargeted by 3 participants. The exceptions were Panlite and GLVU which were retargeted by 5 participants each. I used more participants for just two applications as a) I did not have enough resources to conduct 5 experiments for every application, and b) Panlite and GLVU had the most complicated tactics descriptions. I thus expected them to be the most difficult applications to retarget. Hence, I assigned more participants to them as I expected the most interesting user study results from them. To ensure that these results were not tainted due to learning effects, Panlite and GLVU were never assigned as a second or third application.

Participants were compensated at a flat rate of $120 for completion of a task. I stressed that they were not under time pressure, and could take as long as they needed to complete the task. I made certain that they understood the motivation was quality and not speed. This was a deliberate bias against my goal of short modification time.

The participants worked alone in a lab for the duration of the study. I provided them with a laptop and allowed them to use any editor of their choice. The displays of the participants were captured throughout the study using Camtasia Studio [210]. This provided us with detailed logs of user actions as well as accurate timing information.

4.2.1.4 Experimental Procedure

The user study consisted of 25 separate experiments involving 13 participants. Each experiment lasted up to 6 hours and required a participant to take an unmodified application and retarget it, using RapidRe, for cyber foraging. All 25 experiments were conducted over a 1 month period and no participant participated in more than 1 experiment on any given day. In the rest of this section, I describe the exact procedure that was used for each experiment.

Training Process: Upon arrival, participants were given a release form and presented with a brief introduction to the user study process. They were told that they were going to be making some existing applications work on mobile devices, and that they would be learning to use a set of tools for making applications work within an adaptive runtime system. The participants were then introduced to the concepts of operations, parameters, fidelities, RPCs and tactics. I then conducted a hands-on training session using the GOCR application where I demonstrated how to identify and describe these concepts in Vivendi. The participants were provided with documentation on Vivendi syntax, with many examples. The training then showed them how to create, using the automatically generated APIs, the retargeted client and server components of GOCR. They were also provided with extensive example-filled documentation explaining how to create the client and server component. The entire training session lasted less than one hour in all cases.

Testing Process: After training, each participant was randomly assigned to an application to be modified. They were given all accompanying documentation for the application written by the original application developer that explained how the application worked and explained the functional blocks that made up the application. This documentation did not mention anything about making the application adaptive as that was not the original developer's intention. The participants were also provided with domain information from which it was possible to extract the parameters and fidelity variables. For example, the domain information might say that for 3D graphics applications, the name of the model, the size of the model, the current viewing position and current perspective affect the resource usage of the application. It was up to the participants to determine exactly which application variables these general guidelines mapped to.

Task Structure: I provided participants with a structured task and a set of general instructions. The task structure consisted of three stages, as shown in Table 4.3. In Stage A, the primary activity was creating the tactics file; in Stage B, it centered on creating the client code component; in Stage C, it centered on creating the server component. I wanted to cleanly isolate and independently study the ability of novices to perform each of these stages. I therefore provided participants with an error-free tactics file for use in Stages B and C. This ensured that errors made in Stage A would not corrupt the analysis of Stages B and C. For Stages B and C, the success criteria was that the retargeted client and server components should compile cleanly. I did not allow participants to test their retargeted applications as that would have required additional infrastructure support (remote servers, correct data files etc.). This would have also increased the length of each already-lengthy experiment as participants would have a) needed extra training, and b) would have needed extra time to ensure that the applications worked. Given the limits of my resources, I decided to end the experiment as soon as the retargeted code compiled cleanly. A concern with this strategy is that the resulting code would be filled with runtime bugs and that this would affect the overall results. I explain in Section 4.5 how I overcame this problem.

As Table 4.3 shows, each stage consists of a structured sequence of subtasks. For each subtask, participants were given a general set of instructions, not customized in any way for specific applications. After completion of each subtask, I asked participants to answer a set of questions about it.

The exact procedure, documentation, and questionnaires used for the user study are shown in Appendixes C, D, and E.

4.2.1.5 Data Collected

I collected four different kinds of data. In the rest of this section, I use correlations to help interpret this data. These correlations are interpreted as follows: the correlation coefficient,

Stage A *Tactics file*	Stage B *Client component*	Stage C *Server component*
Read docs	Read docs	Read docs
Application	Include file	Include file header
In	Register	service_init API call
Out	Cleanup	Create RPCs
RPC	Find Fidelities	run_server API call
Tactic	Do Tactics	Compile and fix[1]
	Compile and fix[1]	

This table shows the task stages and the subtasks within each stage. [1] Note that in Stages B and C, the participants compiled their code, but did not run it.

Table 4.3: Task Stages

usually denoted as r, denotes the slope of the line that best fits the data. A positive coefficient means that an increase in the x-axis corresponds to an increase in the y-axis. A negative coefficient means that an increase in the x-axis corresponds to a decrease in the y-axis. If the coefficient is zero, the data cannot be approximated by a straight line (there is no correlation between the x values and the y values).

The likelihood indicator, usually denoted as p, gives the probability that this correlation would have occurred if we had just selected values at random. In particular, a high (> 0.5) value indicates that picking random values would have resulted in the same correlation. A small p value indicates a true correlation that could not have occurred by chance.

The data I collected were:

Timing: I used the Camtasia recordings to obtain exact completion times for each subtask. This allowed us to determine the completion times for stages or for the overall task.

Task Process: The Camtasia recordings were also used to collect process data on how participants completed all of the subtasks. In particular, I noted where they had trouble, were confused, or made mistakes.

Self-Report: I collected questionnaire data of several types, including quality of training, ease of use of my solution, and performance in each subtask. The questionnaires comprised mostly of multiple choice questions with a few free-form open-ended questions. The multiple choice questions were used to precisely identify how participants felt about various aspects of the process. The possible answers to each multiple choice question were chosen to form a Likert scale [141]. The Likert method is one of the most popular attitude measuring tools. This method consists of a series of statements, each with an evaluative scale consisting (usually) of five positions running from strongly agree, through neutral, to strongly disagree. A respondent is required to read the given statement and then indicate, on the corresponding scale, the degree to which they agree or disagree with the statement. To prevent learning effects from tainting the results, the ordering of the answers was randomized. The free-form questions were used to allow participants to express any feedback that was not captured by the multiple choice questions.

Solution Errors: I noted all errors in the participants' solutions. I fixed only trivial errors that kept their code from compiling and running. This allowed us to collect performance data from their solutions.

4.3 Results: Little Training

The first criterion for a good solution relates to training duration, as listed in Section 4.1: "Can novices face complex applications confidently with little training?" The training process was presented in Section 4.2.1.4. As stated there, the training session was one hour or less for all participants, thus meeting the above criterion. What is left to be determined is whether this training was adequate. The ultimate test of adequate training is task performance, as shown by the success the participants have in actually modifying applications. These results are reported in the rest of this chapter. A secondary test is the subjective impression of participants. I asked participants several questions after task completion to help us judge whether they felt adequately prepared. In particular, I asked the participants questions that required them to rate the value of the training and training materials.

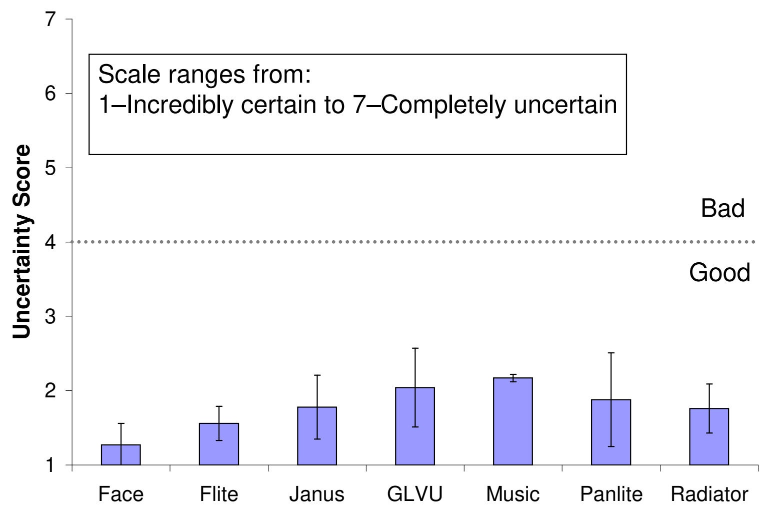
After completing an experiment, each participant was asked the following question: "Was the training helpful?"?. Overall, on a 5-point Likert scale (1 – Helped immensely, 2 – Quite a lot, 3 – Somewhat, 4 – A little bit, 5 – Didn't help at all), the average participant response fell between 1 (Helped immensely) and 2 (Quite a lot), with a mean value of 1.33 and a standard deviation of 0.48. The results were similar for the question "Was the documentation helpful?" The mean response was 1.64 and the standard deviation was 0.76. Figure 4.1 shows the breakdown for these questions on a per-application basis.

In addition, after every subtask of Table 4.3, I probed participants' confidence in their work through the question, "How certain are you that you performed the subtask correctly?" Responses were provided on a 7-point Likert scale (1 – Incredibly certain to 7 – Completely uncertain). As shown in Figure 4.2, participants reported a high degree of confidence across the range of applications. The mean response ranged from 1.3 for Face, to 2.2 for Music.

These self-report ratings correlate (explained in Section 4.2.1.5) highly with the task performance times presented in Section 4.4. The correlation coefficient (r) is 0.88, indicating a strong positive correlation. The p value of 0.009 indicates that it is highly unlikely this correlation would occur by chance. I will discuss these results in more detail in Section 4.7.2, where I identify opportunities for improving my solution. Overall, these results

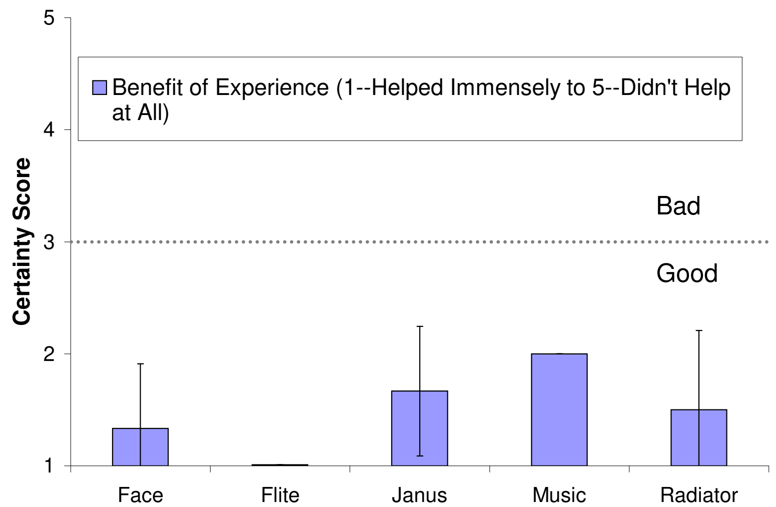
For each application, the height of its bar is the mean score on the Likert scale shown in the legend, averaged across all participants who retargeted that application. Error bars show the standard deviation.

Figure 4.1: Self-Reported Usefulness of Training and Documentation Scores



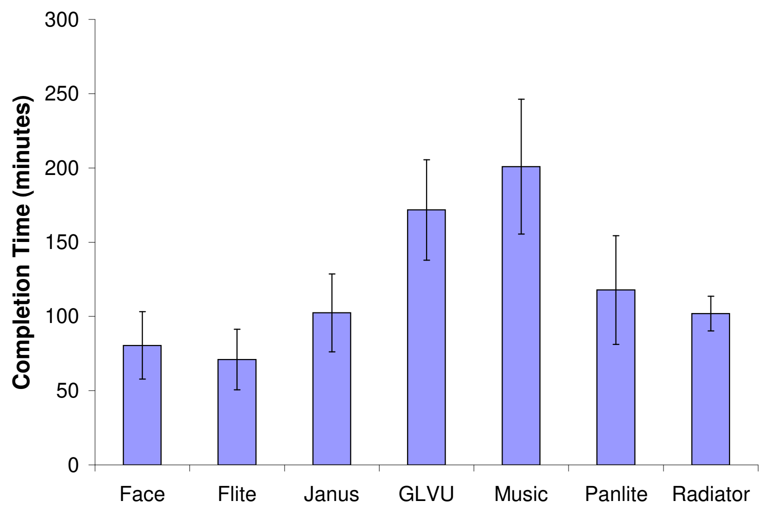
For each application, the height of its bar is the mean uncertainty score on the Likert scale shown in the legend, averaged across all participants who retargeted that application. Error bars show the standard deviation. Participants calibrated the scale as follows: 1 was "Incredibly certain. I would bet my house that I'm correct." while 7 was "Completely uncertain. I would be wasting money betting on my correctness even given 1000-1 odds."

Figure 4.2: Self-Reported Uncertainty Scores



For each application, the height of its bar is the mean certainty score on the Likert scale shown in the legend, averaged across all participants who retargeted that application. Error bars show the standard deviation.

Figure 4.3: Self-Reported Benefit of Experience Scores



For each application, the height of its bar is the mean completion time averaged across all participants who retargeted that application. Error bars show the standard deviation.

Figure 4.4: Measured Application Completion Times

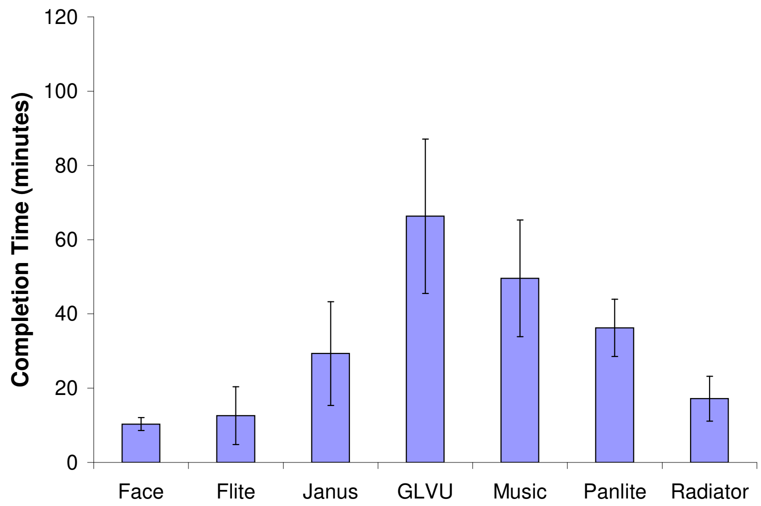
suggest that the participants believed their training prepared them well for the modification tasks they faced.

RapidRe is also an easy process to learn. I asked the 10 participants who retargeted more than one application to rate where experience with RapidRe helped when retargeting a subsequent application. On a 5-pt Likert scale (1–Helped Immensely to 5–Didn't Help at All), the mean answer was 1.67 with a standard deviation of 0.89. Figure 4.3 shows the breakdown on a per-application basis. Note that GLVU and Panlite are not shown as these applications were always the first applications modified by a participant.

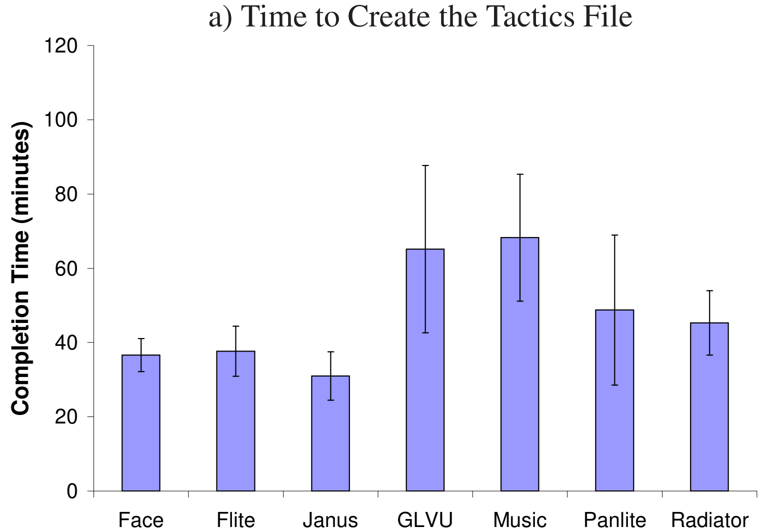
4.4 Results: Quick Modifications

In this section, I address the second criterion listed in Section 4.1: "Can novices modify complex applications quickly?" To answer this question, I examined overall task completion times across the range of applications in validation suite. I found that the average completion time was just over 2 hours, with a mean of 2.08 and a standard deviation of 0.86. Figure 4.4 shows the distribution of task completion times. These data show mean completion times ranging from 70 to 200 minutes, with no participant taking longer than 4 hours for any application. For two applications, some participants only needed about an hour. Figure 4.5 and Table 4.4 presents the breakdown of these times across task stages.

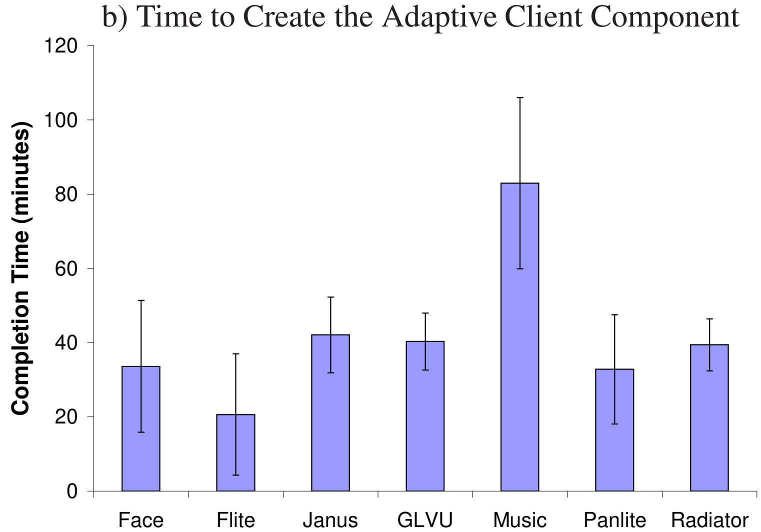
The proportion of the original code base that was modified is another measure of task simplicity. Table 4.5 shows the relevant data. These data show that only a tiny fraction of the code base was modified in every case, and that there was roughly ten times as much automatically generated code as hand-written code. In addition to the reduction in cod-



a) Time to Create the Tactics File

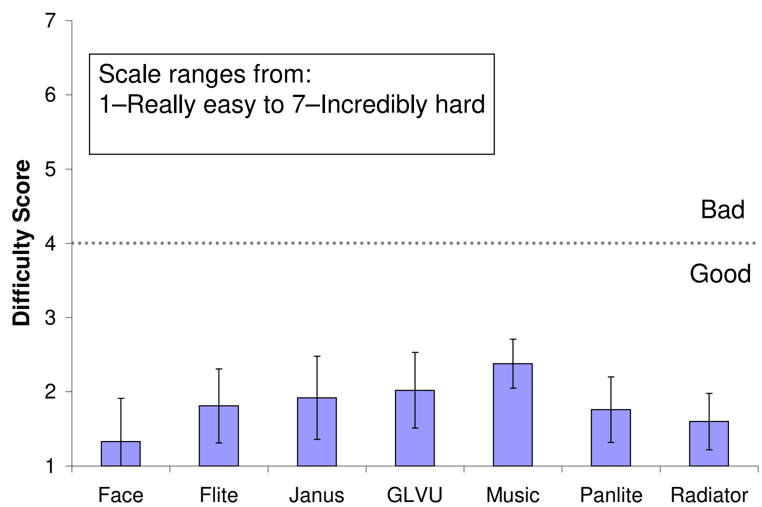


b) Time to Create the Adaptive Client Component



b) Time to Create the Adaptive Server Component

Figure 4.5: Breakdown of Time Needed to Retarget Each Application



For each application, the height of its bar is the mean difficulty score on the Likert scale shown in the legend, averaged across all participants who retargeted that application. Error bars show the standard deviation. Participants calibrated the scale as follows: 1 was "Really easy. Equivalent to writing a simple hello world program" and 7 was "Incredibly hard. Equivalent to adding concurrency support to an operating system."

Figure 4.6: Self-Reported Task Difficulty Scores

App	Stage A	Stage B	Stage C	Total
Face	10.3 (1.7)	36.6 (4.5)	33.6 (17.8)	80.5 (22.7)
Flite	12.6 (7.8)	37.7 (6.7)	20.6 (16.4)	70.9 (20.4)
Janus	29.3 (14.0)	31.0 (6.5)	42.1 (10.2)	102.4 (26.2)
GLVU	66.3 (20.8)	65.1 (22.5)	40.3 (7.7)	171.7 (33.8)
Music	49.6 (15.7)	68.2 (17.1)	83.0 (23.0)	200.8 (45.4)
Panlite	36.2 (7.7)	48.7 (20.2)	32.8 (14.7)	117.8 (36.6)
Radiator	17.2 (6.0)	45.3 (8.7)	39.4 (7.0)	101.9 (11.7)

Each entry gives the completion time in minutes for a task stage, averaged across all participants who were assigned that application. Values in parentheses are standard deviations.

Table 4.4: Completion Time by Task Stage

ing effort, the use of automatically generated stubs allowed participants to get away with minimal knowledge of Chroma.

Finally, I asked participants the question “How easy did you find this task?” Responses were provided on a 7-point Likert scale (1 – Really easy to 7 – Incredibly hard). As Figure 4.6 shows, the responses were heavily weighted toward the easy end of the scale for all applications. These self-report ratings also correlate highly with the task completion times reported earlier (r = 0.82, p = 0.02), increasing the confidence that these results are meaningful. As an additional validation, the self-reported confidence and task difficulty scores were also strongly correlated (r = 0.88, p = 0.01). Taken together, these pieces of evidence converge to suggest that the participants were able to quickly and easily modify the complex applications represented in the validation suite.

4.5 Results: Low Error Rate

In this section, I examine the third criterion listed in Section 4.1: “Can novices modify complex applications with few errors?” Because programming is an error-prone activity, I expect novice-modified applications to contain ordinary programming errors of the types described by Pane et al. [168]. In addition, I expect a few additional simple errors because participants could not test their solution, except to verify that it compiled cleanly. I divide the analysis into two parts; errors in creating tactics files (Stage A); and errors in modifying application code (Stages B and C). An expert scored both parts through code review.

Table 4.6 shows the errors for Stage A. The parameter, RPC, and tactic errors were due to specifying too few parameters, RPC arguments, and tactics respectively. Too few parameters can lead to poor predictions by Chroma. Too few tactics could hurt application performance because the tactics-fidelity space is too sparse. Too few RPC arguments results in a functionally incorrect solution. There were also 4 harmless errors that would not have caused any performance problems. In particular, the participants specified extra fidelities that Chroma would ignore.

For Stages B and C, I classified the errors found as either *trivial* or *non-trivial*. Trivial errors are those commonly occurring in programming assignments. Examples include being off by one on a loop index, or forgetting to deallocate memory. Trivial errors also include those that would have been detected immediately if my procedure allowed participants to test their modified applications. An example is forgetting to insert a `register_API` call to Chroma. All other errors were deemed non-trivial.

Table 4.7 shows the error distribution across applications. A total of 25 trivial errors were found, yielding an average incidence rate of one trivial error per modification attempt. The bulk of these errors (80%) were either a failure to register the application early enough or an incorrect specification of the output file. The register error was due to participants not placing the register call at the start of the application. This prevented the application from connecting to Chroma. The output file errors were due to incorrect use of string functions (a common programming error); this resulted in the application exiting with an error when performing an RPC. Both of these errors would have been discovered immediately if the

App	Lines of Code	File Count	Tactic File Size	Stage B: Client Modifications			
				Lines Added	Lines Removed	Stub Lines	Files Changed
Face	20K	105	10	31 – 68	12 – 15	556	2
Flite	570K	182	10	29 – 39	1 – 5	556	2
GLVU	25K	155	38	62 – 114	3 – 21	1146	2
Janus	126K	227	25	28 – 47	2 – 7	1538	3
Music	9K	55	11	61 – 77	4 – 6	1127	2
Panlite	150K	349	21	30 – 66	1 – 39	1481	3
Radiator	65K	213	15	41 – 51	1 – 47	643	2

				Stage C: Server Modifications			
				Lines Added	Lines Removed	Stub Lines	Files Changed
Face	20K	105	10	26 – 45	15 – 24	186	2
Flite	570K	182	10	13 – 30	3 – 87	186	2
GLVU	25K	155	38	88 – 148	12 – 32	324	2
Janus	126K	227	25	59 – 130	7 – 70	434	4
Music	9K	55	11	131 – 269	23 – 147	203	2
Panlite	150K	349	21	12 – 73	18 – 39	406	3
Radiator	65K	213	15	49 – 106	17 – 32	202	2

Any a–b value indicates a lower bound of a and an upper bound of b. Lines of Code and File Count show the size and number of files in the application. Tactic File Size gives the number of lines in the application's tactics file. The Lines Added and Lines Removed columns show how many lines were added and removed when performing the task. Stub Lines gives the number of automatically generated lines of code. Files Changed gives the maximum number of files that were actually modified by the participants.

Table 4.5: Application Modifications

Apps	Params	RPCs	Tactics	Harmless	# Apps	Okay
Face	0	0	0	0	3	3
Flite	1	0	0	0	3	2
GLVU	1	1	0	3	5	4
Janus	0	0	0	1	3	3
Music	0	1	0	0	3	2
Panlite	0	0	2	0	5	3
Radiator	0	2	0	0	3	1
Total	2	2	0	4	25	18

The # Apps column lists the no. of tactics files created for each app. Okay lists how many tactic files had no harmful errors.

Table 4.6: Total Errors for Stage A Across All Participants

participants had been able to test their applications.

A total of 10 non-trivial errors were found, giving an incidence rate of 0.4 per modification attempt. These took two forms: incorrectly setting parameter values, or incorrectly using fidelities. The parameter errors appeared across many applications while the fidelity errors occurred only in GLVU. Neither of these errors would be immediately apparent when running the application. I examine the performance impact of these errors in Section 4.6.

In summary, I achieved a good success rate with 72% (18 of 25) of the Stage A tactics files having no harmful errors and 64% (16 of 25) of the Stage B and C novice-modified applications having no non-trivial errors. At first glance, these numbers may seem unimpressive. However, no novice-modified application had more than 1 non-trivial error. This is very low given that the applications being modified consisted of thousands of lines of code and hundreds of files. I am confident that any manual attempt, even by experts, to modify these applications would result in far larger numbers of non-trivial errors. This low error rate is also an upper bound as the participants were not able to actually test their modified applications — they only confirmed that it compiled cleanly. The low error rate also substantially improves standard testing phases as the applications are mostly correct. In addition, any errors caught during testing can be rapidly traced to the offending code lines, because relatively few lines of code were inserted or deleted. In Section 4.7.2 I examine ways to reduce this error rate even further.

4.6 Results: Good Quality

The fourth criterion listed in Section 4.1 pertains to the quality of modified applications: "Can novices produce modified applications whose quality is comparable to those pro-

	Trivial Errors					Non-Trivial Errors	
Apps	Reg. Late	Output File	Output Space	Mem. Freed	Other	Params	Fids
Face	0	3	0	0	0	1	0
Flite	0	3	0	0	1	0	0
GLVU	3	0	1	0	1	1	4
Janus	1	2	0	0	0	0	0
Music	1	0	0	2	0	1	0
Panlite	4	0	0	0	0	1	0
Radiator	2	1	0	0	0	2	0
Total	11	9	1	2	2	6	4

Observed trivial errors include: did not register application early enough; did not create output file properly; did not allocate enough space for output; freed static memory. Observed non-trivial errors include: did not set parameters correctly; did not use fidelities to set application state properly

Table 4.7: Total Errors for Stages B and C Across All Participants

duced by an expert?". To answer this question, I conducted the system-centric evaluation described in Section 4.6.1.

4.6.1 System-Centric Evaluation

The goal of the system-centric evaluation was to understand whether developer retargeted applications created using RapidRe had adequate performance. For each application, I asked an expert who had a good understanding of my solution and the application to create a hand-tuned adaptive version of the application. In essence, I hand retargeted each application and ensured that they were fully tuned to achieve optimal performance. The performance measurements from this expert-modified application were then used as a reference against which to compare the performance of novice-modified applications under identical conditions.

4.6.1.1 Testing Scenarios

Ideally, one would compare novice-modified and expert-modified applications for all possible resource levels and user preferences. Such exhaustive testing is clearly not practical. Instead, I performed the comparisons for six scenarios that might typically occur in cyber foraging.

ID	Load	BW	User Prefs	Typical Scenario
Q	Low	High	Highest quality result	Conducting an important business meeting using a language translator
T	Low	High	Lowest latency result	Field engineer just wanting to navigate a quick 3D model of a building to understand the building's dimensions
LH	Low	High	Highest quality result within X s	Sitting in an empty cafe with plentiful bandwidth and unused compute servers
HH	High	High	Highest quality result within X s	Bandwidth is available in cafe but long lived resource intensive jobs are running on the compute servers
LL	Low	Low	Highest quality result within X s	Cafe's compute servers are unused but other cafe users are streaming high bitrate multimedia content to their PDAs
HL	High	Low	Highest quality result within X s	The cafe is full or people either streaming multimedia content or using the compute servers for resource intensive jobs

Load is the compute server load. BW is the available bandwidth. User Prefs are the User Preferences. X is 20s for Face, 25s for Radiator, and 1s for the rest.

Table 4.8: Scenario Summary

These six scenarios are shown in Table 4.8. I used two values of load on compute servers: light (1% utilization) and heavy (95% utilization). I used two bandwidth values: high (5 Mb/s) and low (100 Kb/s), based on published measurements from 802.11b wireless networks [137]. This yielded four scenarios (labeled "LH," "HH, "LL" and "HL" in Table 4.8). All four used the same user preference: return the highest fidelity result that takes no more than X seconds, where X is representative of desktop performance for that application. X was 1 second except for Face (20 s) and Radiator (25 s). The other two scenarios are corner cases: scenario "Q," specifying highest fidelity regardless of latency; and scenario "T," specifying fastest result regardless of fidelity. In all cases, Chroma was provided with correct utility functions that captured the user preferences for the scenario being tested.

4.6.1.2 Experiment Setup

To model a resource-poor mobile device, I used an old Thinkpad 560X laptop with a Pentium 266 MHz processor and 64 MB of RAM. I modeled high and low end compute servers using two different kinds of machines: *Slow*, with 1 GHz Pentium 3 processors and 256 MB of RAM, and *Fast*, with 3 GHz Pentium 4 processors and 1 GB of RAM. The mobile client could also be used as a very slow fallback server if needed. All machines used the Debian 3.1 Linux software distribution, with a 2.4.27 kernel for the client and a 2.6.8 kernel for the servers. To avoid inconsistent behaviour due to Chroma's history-based mechanisms, I initialized Chroma with the same history before every experiment.

4.6.1.3 Procedure

Each novice-modified and expert-modified application was tested on 3 valid inputs in each of the 6 scenarios above. These 18 combinations were repeated using fast and slow servers, yielding a total of 36 experiments per application. Each experiment was repeated 5 times, to obtain a mean and standard deviation for metrics of interest.

For each novice-modified application, I conducted 36 experiments comparing its performance to that of the same application modified by an expert. As explained in Section 4.6.1.1, these 36 experiments explored combinations of compute server loads, network bandwidths, user preferences, and server speeds. For each experiment, I report fidelity and latency of the result. Fidelities are normalized to a scale of 0.01 to 1.0, with 0.01 being the worst possible fidelity, and 1.0 the best. Fidelity comparisons between different versions of the same application are meaningful, but comparisons across applications are not. I report latency in seconds of elapsed time.

I deemed applications to be indistinguishable if their performance on *all 36 experiments came within 1% of each other on both fidelity and latency metrics.* This is obviously a very high bar. Data points differing by more than 1% were deemed anomalies. I evaluated the performance of client and server components of each application separately.

4.6.2 Server Component Results

All 25 retargeted server components achieved indistinguishable performance from their expert-modified counterparts. I thus do not show further results for the retargeted server components.

4.6.3 Client Component Results

Table 4.9 presents the client component results. The table entry for each participant, and application modified by that participant, gives the percentage of the 36 experiments for which novice-modified and expert-modified applications were within 1%. A score of 100% indicates indistinguishable applications; a lower percentage indicates the presence of anomalies. Table 4.9 shows that novice- and expert-modified applications were indistinguishable in 16 out of 25 cases.

Tables 4.10 to 4.14 show details for the five anomalous applications (Face, Flite, GLVU, Panlite, and Radiator). I only show the performance of one anomalous version of GLVU as the other 3 anomalous versions were similar. For each application, I provide the relative fidelity and latency obtained for all 3 inputs in all 6 scenarios. The relative fidelity is

Participant No.	Application						
	Face	Flite	GLVU	Janus	Music	Panlite	Radiator
1	100%		44%				
2				100%		100%	
3			44%	100%	100%		
4	100%					83%	94%
5			44%		100%		
6						100%	
7			100%	100%			
8		100%				100%	
9			44%				
10						100%	100%
11	67%	100%					
12		67%					78%
13					100%		

A score of 100% indicates that the participant's client version matched the performance of the expert in all 36 experiments. A blank entry indicates that the participant was not asked to create a modified version of that application.

Table 4.9: Relative Performance of Novice-Modified Client Component

expressed as H (Higher), S (Same), or L (Lower) than the expert-modified version. Latency is given as a ratio relative to the expert. For example, a value of 11.9 indicates that the novice-modified application had 11.9 times the latency of the expert-modified application, for the same input.

GLVU was the source of most of the anomalies. The novices' solutions selected an inappropriately high fidelity resulting in their solutions exceeding the latency goals for the T, LH, HH, LL, and HL scenarios. Code inspection of the anomalous versions of GLVU revealed that all 4 anomalous versions made the same mistake. To successfully modify GLVU, participants needed to use a fidelity value returned by Chroma to set the application state before performing the chosen tactic. In all 4 cases, the participants read the value of the fidelity but forgot to insert the 2 lines of code that set the application state. As a result, these 4 applications always performed the chosen tactic using the default fidelity, and were unable to lower fidelity for better latency.

The other 5 anomalies (1 Face, 1 Flite, 1 Panlite and 2 Radiator versions) were due to mis-specified parameters. In 4 of the 5 cases, the participants set a parameter value that

Scenarios

	Q	T	LH	HH	LL	HL
Slow	...	S, 5.24	S, 5.22	...	...	...
	...	S, 5.26	S, 5.24	...	...	...
	...	S, 5.20	S, 5.25	...	...	...
Fast	...	S, 14.21	S, 14.22	...	...	...
	...	S, 14.37	S, 14.29	...	...	...
	...	S, 14.17	S, 14.25	...	...	...

Participant 11

Each entry consists of a relative fidelity followed by a relative latency for a single input. The relative fidelity is either L–lower than expert, S–same as expert, or H–higher than expert. The relative latency gives the ratio between the participant's version versus the expert. E.g., a latency of 11 indicates the participant's version had 11 times the latency of the expert. Only the anomalous values are presented. All other values are replaced by the symbol "..." to avoid visual clutter.

Table 4.10: Detailed Results for Anomalous Retargeted Face Client Component

was too small. For Panlite, the parameter was set to the length of the entire input string instead of just the number of words in the input string. For Flite, the participant forgot to set the parameter value, which then defaulted to a value of 0. For Face, the parameter was set to input file name length instead of file size. For Radiator (participant 12), the parameter was set to a constant value of 400 instead of the number of polygons in the lighting model. These mis-specifications of parameter values led Chroma to recommend fidelity and tactic combinations that exceeded the scenario latency requirements.

In the last case (Participant 4's version of Radiator), the parameter was set to a far higher value than reality. In particular, it was set to the size of the model file on disk instead of just the number of polygons in the model being used. This caused Chroma to be more pessimistic in its decision making than it should have been. So this application version achieved lower fidelity than it could have.

4.6.4 Summary

In summary, the results confirm that the novice-modified application code is of high quality. All 25 of the server components, and 16 of the 25 client components modified by participants were indistinguishable from their expert-modified counterparts. Where there was divergence, analysis of the anomalies provided ideas for improving my solution. I discuss these improvements in Section 4.7.2.

Scenarios

	Q	T	LH	HH	LL	HL
Slow	...	...	S, 2.33	...	S, 2.45	...
	...	...	S, 2.77	...	S, 2.74	...
	...	...	S, 2.51	...	S, 2.42	...
Fast	...	...	S, 2.91	...	S, 2.97	...
	...	...	S, 3.56	...	S, 3.23	...
	...	...	S, 3.16	...	S, 3.38	...

Participant 12

Each entry consists of a relative fidelity followed by a relative latency for a single input. The relative fidelity is either L–lower than expert, S–same as expert, or H–higher than expert. The relative latency gives the ratio between the participant's version versus the expert. E.g., a latency of 11 indicates the participant's version had 11 times the latency of the expert. Only the anomalous values are presented. All other values are replaced by the symbol "..." to avoid visual clutter.

Table 4.11: Detailed Results for Anomalous Retargeted Flite Client Component

Scenarios

	Q	T	LH	HH	LL	HL
Slow	...	H, 11.26	...	H, 3.04	...	H, 3.06
	...	H, 13.29	H, 1.16	H, 4.65	H, 1.15	H, 4.61
	...	H, 8.31	...	H, 2.47	...	H, 2.45
Fast	...	H, 11.34	...	H, 3.06	...	H, 3.02
	...	H, 13.40	...	H, 4.59	...	H, 4.67
	...	H, 7.85	...	H, 2.46	...	H, 2.48

Participant 1

Each entry consists of a relative fidelity followed by a relative latency for a single input. The relative fidelity is either L–lower than expert, S–same as expert, or H–higher than expert. The relative latency gives the ratio between the participant's version versus the expert. E.g., a latency of 11 indicates the participant's version had 11 times the latency of the expert. Only the anomalous values are presented. All other values are replaced by the symbol "..." to avoid visual clutter.

Table 4.12: Detailed Results for Anomalous Retargeted GLVU Client Component

Scenarios

	Q	T	LH	HH	LL	HL
Slow	…	…	H, 7.68	…	H, 7.57	…
	…	…	H, 6.89	…	H, 6.93	…
	…	…	H, 7.54	…	H, 7.49	…
Fast	…	…	…	…	…	…
	…	…	…	…	…	…
	…	…	…	…	…	…

Participant 4

Each entry consists of a relative fidelity followed by a relative latency for a single input. The relative fidelity is either L–lower than expert, S–same as expert, or H–higher than expert. The relative latency gives the ratio between the participant's version versus the expert. E.g., a latency of 11 indicates the participant's version had 11 times the latency of the expert. Only the anomalous values are presented. All other values are replaced by the symbol "…" to avoid visual clutter.

Table 4.13: Detailed Results for Anomalous Retargeted Panlite Client Component

4.7 Analysis of Results

Sections 4.3 to 4.6 have shown that RapidRe is able to achieve the four success criteria posed in Section 4.1. In this section, I take a deeper look at both the RapidRe process and the user study results and ask the following two questions:

1. Why is RapidRe so successful at reducing the time needed for developers to retarget large unfamiliar applications? What magic is at work here? This question is answered in Section 4.7.1.

2. How can RapidRe be improved even further? This question is answered in Section 4.7.2.

4.7.1 Why My Solution Works

At first glance, the results of the previous sections seem too good to be true. Modifying a complex application for cyber foraging, a task that one expects will take a novice multiple weeks, is accomplished in just a few hours. The modified application performs close to what one could expect from an expert. Yet, it is not immediately clear what accounts for this success. Vivendi, Chroma and the stub generator are each quite ordinary. Somehow, their combined effect is greater than the sum of the parts. What is the magic at work here?

Scenarios

	Q	T	LH	HH	LL	HL
Slow	…	…	…	…	…	…
	…	…	…	…	…	…
	…	…	L, 0.17	…	…	…
Fast	…	…	…	…	…	…
	…	…	…	…	…	…
	…	…	L, 0.05	…	…	…

Participant 4

	Q	T	LH	HH	LL	HL
Slow	…	…	…	…	…	…
	…	…	…	…	…	…
	…	…	H, 3.98	H, 1.14	H, 1.10	H, 1.15
Fast	…	…	…	…	…	…
	…	…	…	…	…	…
	…	…	H, 1.11	H, 1.12	H, 1.16	H, 1.14

Participant 12

Each entry consists of a relative fidelity followed by a relative latency for a single input. The relative fidelity is either L–lower than expert, S–same as expert, or H–higher than expert. The relative latency gives the ratio between the participant's version versus the expert. E.g., a latency of 11 indicates the participant's version had 11 times the latency of the expert. Only the anomalous values are presented. All other values are replaced by the symbol "…" to avoid visual clutter.

Table 4.14: Detailed Results for Anomalous Retargeted Radiator Client Components

The key to explaining the success is to recognize, as mentioned in Section 2.3.4, the existence of a deep architectural uniformity across modified applications. This is in spite of diversity in application domains, programming languages, modular decompositions, and coding styles. It arises from the fact that, at the highest level of abstraction, we are dealing with a single genre of applications: mobile interactive resource-intensive applications.

In a mobile environment, all sensible decompositions of such applications place interactive code components on the mobile client, and resource-intensive components on the compute server. This ensures low latency for interactive response and ample compute power where needed. This space of decompositions is typically a tiny fraction of all possible procedure-level decompositions. The challenge is to rapidly identify this “narrow waist” in an unfamiliar code base.

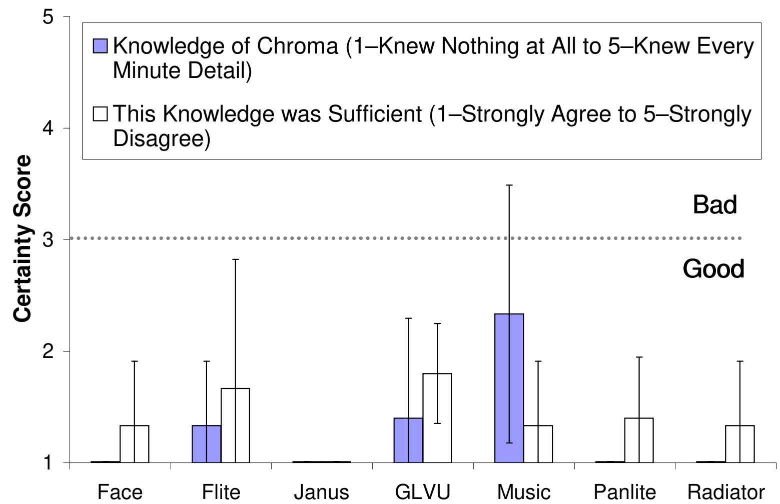
In examining a broad range of relevant applications, I was surprised to observe that every unmodified application of interest to us was already structured to make such decomposition easy. In hindsight, this is not so surprising. Code to deal with user interaction is usually of a very different flavor from code that implements image processing, speech recognition, and so on. Independent of mobile computing considerations, a capable programmer would structure her application in a way that cleanly separates these distinct flavors of code. The separation would be defined by a small procedural interface, with almost no global state shared across that boundary — exactly the criteria for a narrow waist.

In addition to this similarity of code structure, there is also similarity in dynamic execution models. First, there is a step to obtain input. This could be a speech utterance, a natural language fragment, a scene from a camera, and so on. Then, resource-intensive processing is performed on this input. Finally, the output is presented back to the user. This may involve text or audio output, bitmap image display, etc.

In modifying such an application for mobile computing, the main change is to introduce an additional step before the resource-intensive part. The new step determines the fidelity and tactic to be used for the resource-intensive part. It is in this step that adaptation to changing operational conditions occurs. A potential complication is the need to add the concept of fidelity to the application. Fortunately, this has not been necessary for any of the applications. Most applications of this genre already have “tuning knob” parameters that map easily to fidelities — another pleasant surprise.

RapidRe exploits these similarities in architecture and execution model. The architectural similarity allows us to use a “little language”(Vivendi) to represent application-specific knowledge relevant to cyber foraging. This knowledge is extracted by a developer from the source code of an application and used to create the tactics file. The similarity in execution model allows us to use a common runtime system (Chroma) for adaptation across all applications. The use of stubs raises the level of discourse of the runtime system to that of the application. It also hides many messy details of communication between mobile device and compute server.

The net effect of executing the solution steps using a checklist is to quickly channel attention to just those parts of application source code that are likely to be relevant to cyber foraging. At each stage in the code modification process, the developer has a crisp and



For each application, the height of its bar is the mean certainty score on the Likert scale shown in the legend, averaged across all participants who retargeted that application. Error bars show the standard deviation.

Figure 4.7: Self-Reported Chroma Knowledge Scores

narrow goal to guide his effort. This focused approach allows a developer to ignore most of the bewildering size and complexity of an application.

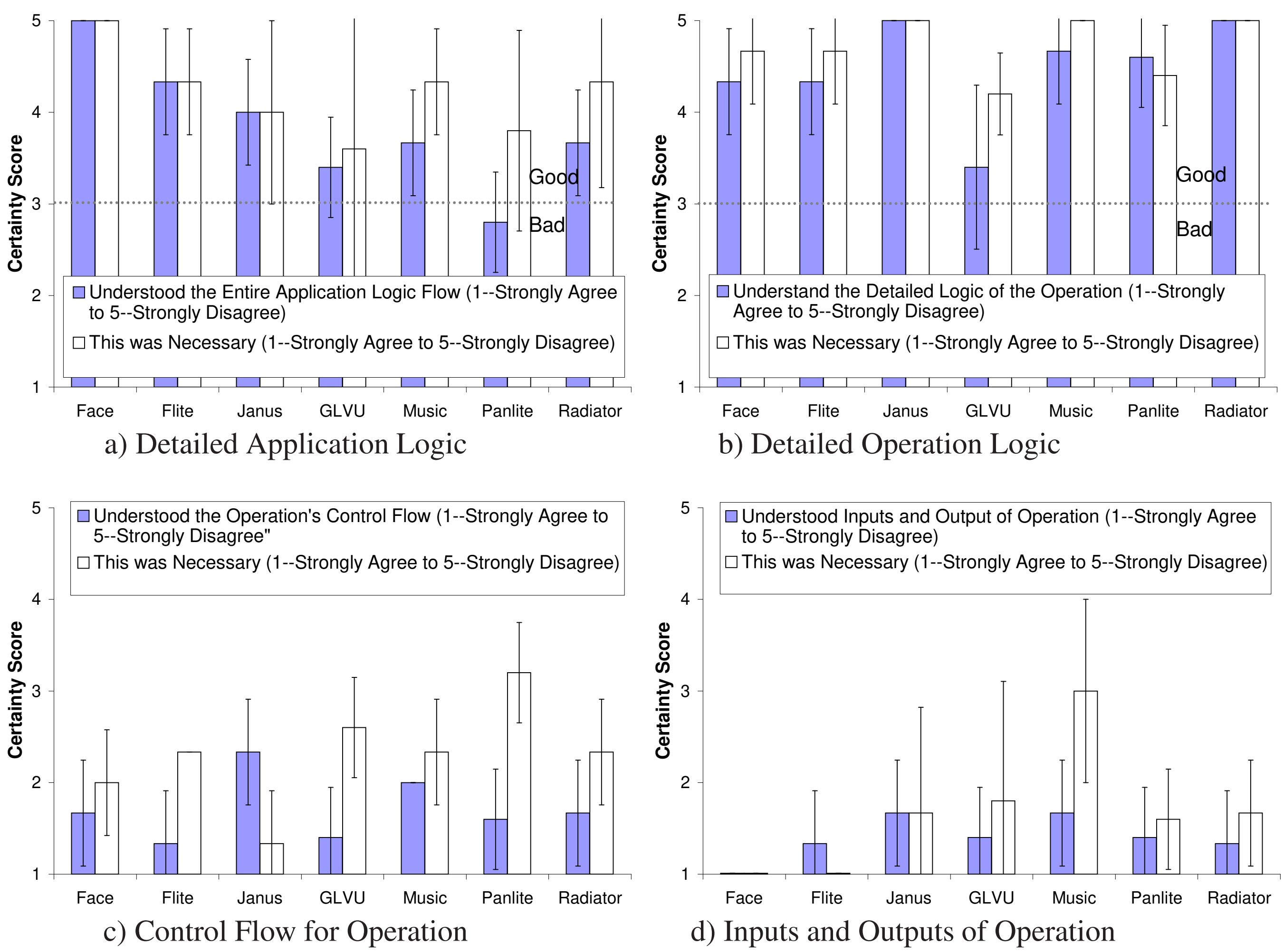
In addition to reducing programmer burden, there is also a significant software engineering benefit to the clean separation of concerns implicit in my design. The application and the runtime system can be independently evolved, with many interface changes only requiring new stubs.

It should be noted that RapidRe is not a magic bullet. There are applications for which it will not be useful. As mentioned above, RapidRe does not help developers in adding fidelity variable support to applications. In addition, applications that do not enforce clean separation between the modules that make up a tactic will need to be reworked before they can be used. For example, applications that use global variables to communicate between modules.

4.7.1.1 Information Isolation

A key reason, listed above, why RapidRe works is because it dramatically reduces the amount of information that developers need to know before they can retarget an application. In this subsection, I present process data that quantifies the effect of this information isolation. I obtained this process data by asking each participants in the user study to answer a number of questions after they had finished retargeting an application.

The first two questions I asked were “How much about Chroma did you learn during the user study?” and “Was this level of Chroma knowledge sufficient to complete the task?”. The mean response for the first question was 1.28 (standard deviation of 0.68) on a 5-pt



For each application, the height of its bar is the mean certainty score on the Likert scale shown in the legend, averaged across all participants who retargeted that application. Error bars show the standard deviation.

Figure 4.8: Self-Reported Application Knowledge Scores

Likert scale (1–Knew Nothing at All to 5–Knew Every Minute Detail). For the second question, the mean was 1.44 (standard deviation of 0.58) on a 5-pt Likert scale (1–Strongly Agree to 5–Strongly Disagree). Figure 4.7 shows the breakdown for these two questions on a per-application basis. Overall, every participant felt that they did not need to know much about Chroma and that this was still sufficient to successfully retarget an application.

I then asked each participant four questions that asked them to state how much application knowledge they needed to successfully retarget the application. The four questions were 1) Did you need to understand the entire detailed application logic?, 2) Did you need to understand the detailed logic of the operation? For example, exactly how the language translation algorithms worked, 3) Did you need to understand the control flow of the operation?, and 4) Did you need to understand the inputs and outputs of the operation? For each question, I also asked each participant (similar to the question about Chroma knowledge) to rate how important that particular piece of information was in allowing them to successfully retarget the application. Each question was answered on a 5-pt Likert scale (1–Strongly Agree to 5–Strongly Disagree).

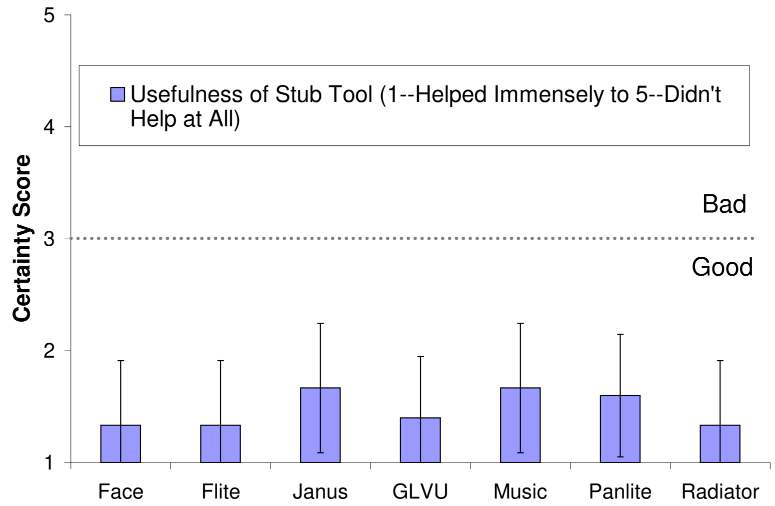
Overall, the participants said that they did not need to know the detailed application logic (mean of 3.52, standard deviation of 1.48) and that they also did not need to know the detailed logic to perform the operation (mean of 4.36, standard deviation of 0.91). They also felt that knowing the detailed application logic (mean of 4.00, standard deviation of 1.08) and the detailed operation logic (mean of 4.64, standard deviation of 0.49) was not needed to successfully retarget the application. On the other hand, they did say that they needed to know the control flow of the operation (mean of 1.76, standard deviation of 0.72) and the inputs and outputs of the operation (mean of 1.4, standard deviation of 0.5). They also felt that knowing the operation control flow (mean of 1.64, standard deviation of 0.7) and the operation's inputs and outputs (mean of 1.84, standard deviation of 1.14) was crucial in retargeting the application. Figure 4.8 shows a breakdown of these answers on a per-application basis

This process data provides strong evidence of the information isolation provided by RapidRe. Participants do not need to understand the complicated and detailed runtime system and they only need to understand a small bit of the application being retargeted. In particular, they just needed to understand the application control pertaining to the operation and the inputs and outputs of the operation. This validates the RapidRe process as these are the only portions of the application that it tells developers to find and understand (described in Section 2.7).

4.7.1.2 Usefulness of Stub Generator

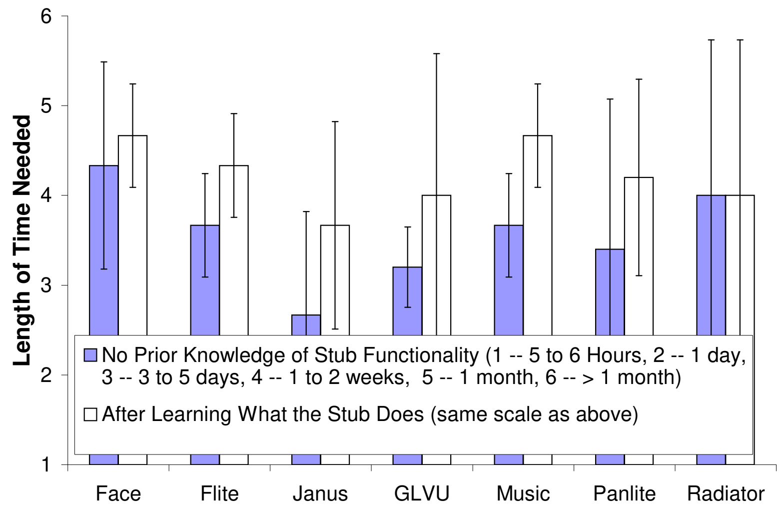
A key component that facilitates the information isolation properties of RapidRe is the stub generator. In particular, the stub generator is the reason that developers do not need to know anything about the runtime system.

I asked participants how useful they thought the stub generator was. The mean response, on a 5-pt Likert scale (1–Helped Immensely to 5–Didn't Help at All), was 1.48 with a standard deviation of 0.51. Figure 4.9 shows this response on a per-application basis. The



For each application, the height of its bar is the mean certainty score on the Likert scale shown in the legend, averaged across all participants who retargeted that application. Error bars show the standard deviation.

Figure 4.9: Self-Reported Usefulness of Stub Scores



For each application, the height of its bar is the mean certainty score on the Likert scale shown in the legend, averaged across all participants who retargeted that application. Error bars show the standard deviation.

Figure 4.10: Self-Reported Estimated Manual Modification Time Scores

results indicate that the participants thought that the stub generator was highly useful.

To further quantify the benefits of the stub generator, I asked participants to estimate how long they would need to retarget their application if there was no stub generator. I asked this question two times. The first time I asked this question, I provided participants with no information about what exactly the stub generator did. I then asked the participants a number of other questions (to try to make them forgot their answer). I then asked them this same question again without letting them see their previous answer. However, this time, I explained to them exactly what the stub generator was doing (I told them a summary of Section 2.7). Both times, I provided them with 6 ascending options. They were "5 to 6 hours", "1 day", "3 to 5 days", "1 to 2 weeks", "1 month", and "greater than 1 month". On average, without detailed knowledge of the stub, participants said that they would need between "3 to 5 days" to "1 to 2 weeks" (mean of 3.52, standard deviation of 1.12). After learning what the stub did, participants said that they would need between "1 to 2 weeks" to "1 month" (mean of 4.2, standard deviation of 1.08) to retarget the application. Figure 4.10 shows the breakdown of both these questions on a per-application basis.

Overall, the participants answers demonstrated the benefits provided by the stub generator. The stub, in turn, is made possible because of the concept of tactics and the tactics file (created using Vivendi). In particular, participants estimated that they would need up to 2 weeks to retarget an application without the stub generator. This time matches my prior experience of retargeting applications without RapidRe.

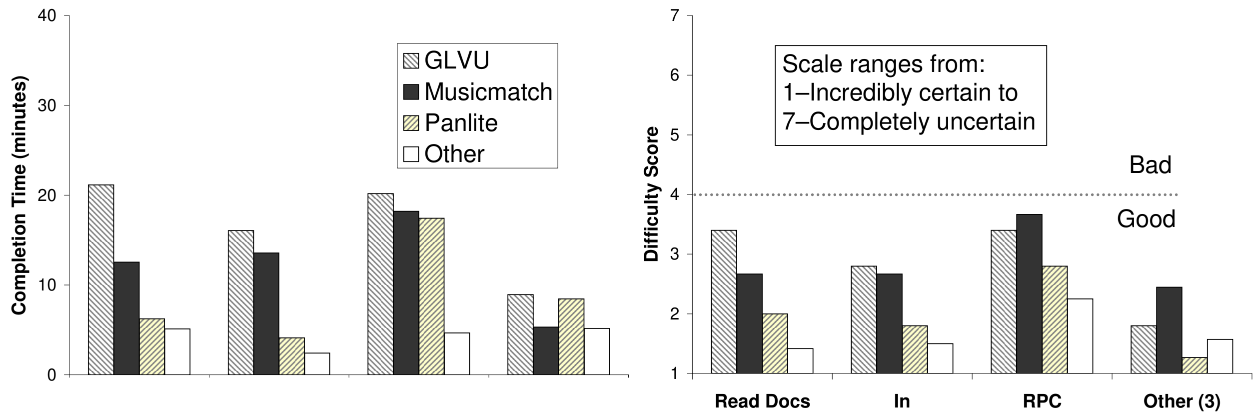
4.7.2 Improving RapidRe

In this section, I identify areas in which my solution, RapidRe, can be improved. RapidRe could be improved in several ways: eliminating all errors, further reducing the time required, and ensuring it applies to the widest possible range of potential mobile applications. In order to chart out these future directions, I analyzed all the non-trivial errors, examined how the subjects spent their time, and examined the differences in applying the solution to the range of applications. Because it is already fast, I focused on improving RapidRe's quality.

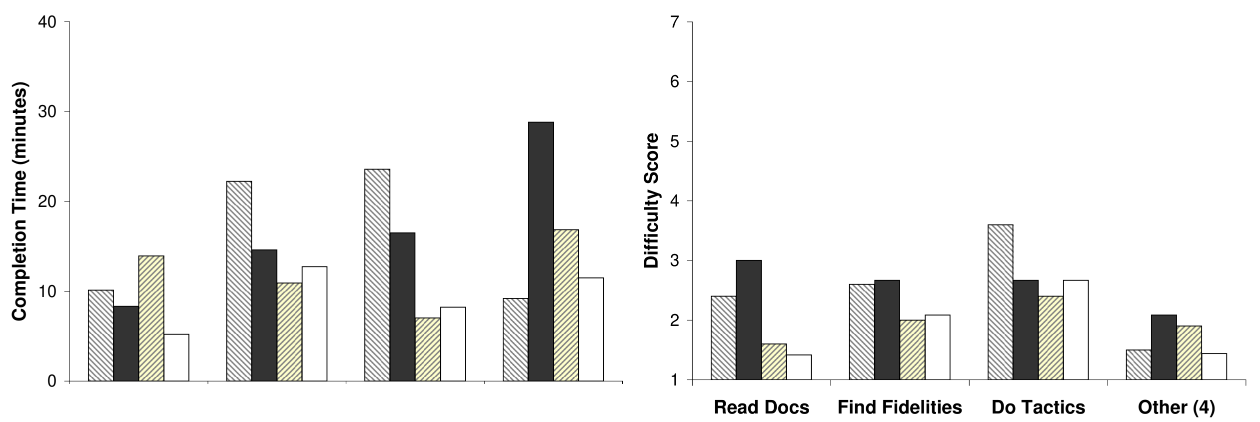
From the results displayed in Sections 4.5 and 4.6, I identified the errors made by the participants. The non-trivial errors in Stage A took 3 forms; specifying too few parameters, specifying too few RPC arguments, and specifying too few tactics. These errors were distributed randomly across participants and applications.

All of the non-trivial errors in Stages B and C occurred in one subtask, "Find Fidelities", while creating the client, and were of only two types. In one type of error, all for GLVU, novices successfully read the fidelity values returned by Chroma, but failed to use those values to set the application state. In the other cases, novices failed to set the parameters correctly to reflect the size of the input. There were no errors associated with any other subtask involved in creating either the client or server.

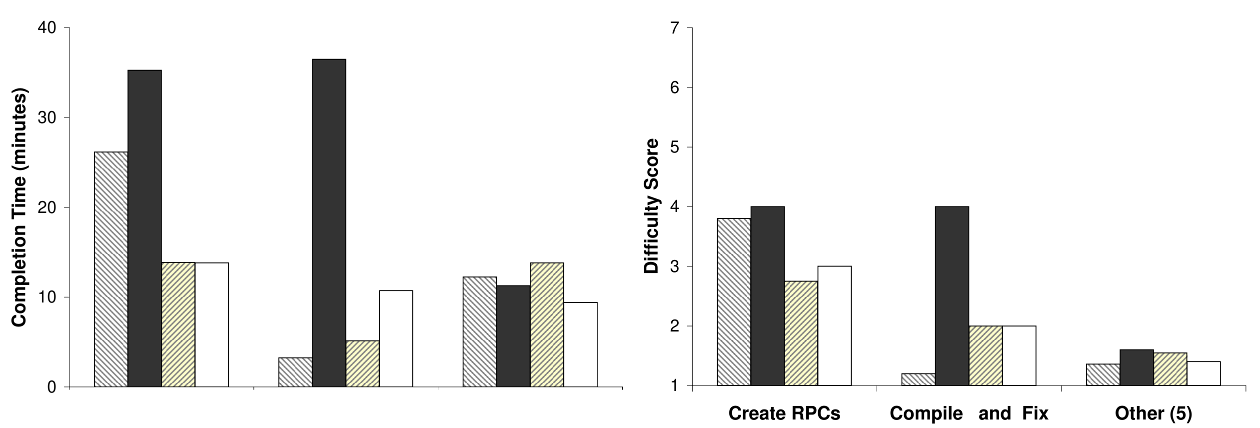
In order to eliminate these errors, I needed to determine whether the programmers were unable to understand the task or simply forgot to complete all necessary steps. If the latter,



Stage A: Creating the Tactics File



Stage B: Creating the Client Component



Stage C: Creating the Server Component

Only the largest time values (left column) and self-reported difficulty scores (right column) are shown. The Other bar presents either the sum (for times) or the average (for difficulty) of the remaining subtasks (no. of subtasks shown in parentheses on the x-axis).

Figure 4.11: Time and Difficulty of Each Individual Subtask

straightforward improvements in the instructions may be sufficient to eliminate all observed errors. An examination of the evidence summarized in Figure 4.11 suggests that forgetfulness is the likely cause. Subjects did not report that the “Find Fidelities” subtask was particularly difficult, rating it only 2.6 on a 7-point difficulty scale where 4 was the midpoint. They also did not report a high degree of uncertainty (not shown) in their solution, giving it a 1.7 on a 7-point uncertainty scale (midpoint at 4). Table 4.9 shows that, of the seven programmers who made at least one non-trivial error, five successfully modified a second application with no errors. Of the other two, one modified only a single program, and the other made non-trivial errors on both programs they modified. Together, these results suggest that nearly all the subjects were capable of performing all tasks correctly. This implies forgetfulness was the problem. This analysis leads us to believe that forcing developers to pay more attention to these error-prone parts of the “Find Fidelities” task, perhaps with an extended checklist, will eliminate most of the errors.

Figure 4.11 also suggests that the difficult and time-consuming tasks vary considerably across application types. For example, GLVU required more time in the “In” and “RPC” subtasks of Stage A as it had a large number of parameters and RPC arguments as shown in Table 4.1. It also had larger times for the “Find Fidelities” and “Do Tactics” subtasks of Stage B as “Find Fidelities” required participants to set each of the parameters while “Do Tactics” required participants to manage each of the RPC arguments. Similarly, Panlite required more time during the “Tactic” subtask of Stage A as it had a large number of tactics that had to be identified and described. In each of these cases, I suspect that instructing programmers on how to keep track of the minutiae of these subtasks, and ensuring that each is completed, would be of substantial benefit.

Finally, Music had a very large “Compile and fix” time for Stage C. This was because Music was originally written as a non-adaptive, desktop oriented client–server application. Thus it already used a specific on-wire data format that participants had to reuse, requiring them to write large amounts of relatively simple buffer manipulation code. Trivial errors in this code led to the increased subtask times. This suggests that there will be issues specific to some types of applications that may make them less amenable to my solution.

4.7.3 Applicability of RapidRe

In this section, I discuss the applicability of RapidRe. This discussion first identifies the types of applications that RapidRe is most suitable for. It then discusses exactly which parts of the application retargeting process RapidRe helps to speed up.

4.7.3.1 Applications That Can Benefit From RapidRe

I first describe exactly what kinds of applications can fully benefit from RapidRe. This knowledge is a combination of my basic assumptions with the knowledge obtained from developing and evaluating RapidRe. For RapidRe to be most effective, applications should have the following properties:

- **Interactive Nature** : Applications should be interactive in nature. In particular, they must have the following mode of operation; the user provides input to the application and the application then performs some computation on the input. This sequence of user input followed by computation can occur as many times as necessary.

- **Resource Constrained on Mobile Devices** : A suitable application should be resource constrained when running on a mobile device. In particular, it should not be possible to achieve acceptable application performance, when running on a mobile device, without adapting the application. In addition, Chroma was designed for CPU and/or memory constrained applications and has only limited support for bandwidth constrained applications. Hence, bandwidth constrained applications may not work well with RapidRe (this was explained in more detail in Section 2.2.3).

- **Independent and Stateless RPCs** : Chroma assumes that the individual RPCs that make up an application's tactics are independent. In particular, a RPC should not depend on state created by a previous RPC. Chroma has, by using the server constraint syntax of Vivendi, limited support for RPCs that need to be co-located on particular servers. However, using this feature limits the ability of Chroma to pick the best runtime solution. Chroma also has a very simple error handling mechanism which assumes that RPCs are stateless; i.e., it is possible for any RPC to fail and then be retried without any negative effects. If this assumption is violated, then the developer will have to manually add the appropriate code to the application server such that it can recover from a failed RPC request.

- **No Global Variables Used Between RPCs** : A key requirement for any RPC mechanism is that all arguments needed for the RPC to successfully execute should be specified as inputs to the RPC. In particular, an RPC should not depend on the value of a global variable that is set in some other RPC (or the client code). An example of such a global variable is the *errno* variable commonly used in Unix programs. If an RPC uses such global variables, it must be manually fixed to either not use these variables or to receive the current value of these variables as an input to the RPC. Note that it is okay for an RPC to use a global variable that is fully self-contained within that RPC. However, even in this special case, care must be taken to not violate the stateless assumption mentioned above.

- **Written in a C-friendly Language** : RapidRe achieves a large part of its language-independent characteristic because Chroma is written in C. In my experience, all the languages I tested RapidRe with have an easy and well-documented way to interface with a C library (the stub code is linked into the application as a C library). In particular, it was possible to use the Simplified Wrapper and Interface Generator (SWIG) tool [206] to automatically create API wrappers for the non-C languages used in this thesis. This eliminated the need to change the stub generator to generate code in different languages. If an application is written in a language that doesn't have an easy C interface, then it will be necessary to change the stub generator. This change would take at least a few days to complete.

4.7.3.2 The Application Retargeting Process

There are many steps involved in retargeting an application from one domain to another. These include a) identifying if the application is suitable for the other domain, b) converting the application compilation routines to work in the other domain, c) converting the application itself to support the other domain, and d) testing the retargeted application in the other domain.

RapidRe only helps with step c). It significantly reduces the time needed to convert a legacy application and make it work on a mobile device. This time savings can be on the of one to two weeks per application. However, it doesn't help with the other three retargeting steps. In particular, developers still have to identify if a legacy application is suitable for running on a mobile device (i.e., it supports the requirements stated in Section 4.7.3 above). From my personal experience, this can take a few hours per application. The developer also has to modify the application's build process to support the new environment. My experience is that converting the build process can take a few minutes to one day depending on how complicated and well written the application's build process is.

Finally, RapidRe decreases some of the time needed to test the retargeted application and verify that it accurately works in the new domain. In particular, RapidRe decreases the unit-testing time as the automatically generated stub code can be unit tested separately from the retargeted applications. As shown in Figure 4.5, the amount of code added to the application is minimal. Hence, RapidRe also makes it easier to unit-test the individual client and server components of the retargeted applications. However, RapidRe does not speed up the final domain-specific overall application testing. In the context of mobile computing, these are the tests that verify that the retargeted application can actually work effectively on a mobile device in a variety of mobile environments.

4.8 Summary

In this chapter, I presented the results of a user study that was designed to measure the effectiveness of RapidRe. In the user study, thirteen undergraduate students retargeted seven large applications using RapidRe. The user study showed that this group of novice developers were able to retarget each application in under 4 hours. The quality of the retargeted applications was also excellent; comparable to expert-modified applications. A closer look at the process data obtained during the user study shows that RapidRe is excellent at isolating developers from having to know anything at all about Chroma. RapidRe also requires developers to only know very little about the applications they are retargeting. In particular, only the control flow of the operation and the inputs and outputs of the operation need to be known. Finally, an analysis suggested that RapidRe's error rate might be reduced by drawing developer attention, through a checklist or other means, to certain error-prone portions of the retargeting process. Overall, this chapter effectively validates that RapidRe allows even novice developers to quickly and easily retarget computationally-intensive applications for cyber foraging. Furthermore, these applications have comparable quality to expert-retargeted applications. In the next chapter, I show that these expert-retargeted applications achieved excellent performance in a variety of mobile environments.

Chapter 5

Validation: Effectiveness

In the previous chapter, I showed that the relative performance of novice-retargeted applications was excellent. In particular, they achieved equivalent performance, in many cases, to to expert-retargeted applications. In this chapter, I show that these expert-retargeted applications achieve excellent absolute performance in a variety of mobile scenarios. More precisely, I show that Chroma can achieve excellent application performance in mobile environments.

5.1 Validation Strategy

The validation goal of this chapter can be separated into six smaller questions. Successfully answering these six questions will provide excellent validation of the top-level goal.

1. **Can Chroma correctly pick the optimal runtime settings for an application when the user preference and resource environment is fixed?** This is the basic requirement that Chroma has to provide. A runtime setting consists of precise fidelity variable settings, the tactic to run, and precise server selections for each RPC in the chosen tactic. The validation for this question is provided in Section 5.2.

2. **Can Chroma work in a dynamic mobile environment?** In particular, can Chroma pick the best application runtime settings even when user preferences and the resource environment changes? Satisfying this dynamic requirement is vital for Chroma to successfully support the needs of a mobile user. This question is validated in Section 5.3. The validation also shows that for Chroma to achieve good performance, it needs to use dynamic utility functions that capture the current user preferences.

3. **Is the overhead of Chroma's decision making process reasonable?** In particular, it should not add substantially to the total latency of the application. This question's validation is presented in Section 5.4.

4. **Is Chroma able to use extra server resources in the environment to automatically improve application performance?** The validation of this question, shown

in Section 5.5, demonstrates that Chroma can effectively handle cases where either overprovisioning or underutilization results in extra server resources being available for use.

5. **Are there simple client-side mechanisms that Chroma can use to protect against malicious servers?** Answering this question will provide evidence that Chroma can be used in an untrusted mobile environment. Section 5.6 provides the results of this validation.

6. **How does Chroma work in real environments?** This question removes some of the assumptions made by previous sections and attempts to investigate how Chroma would behave in real environments. In particular, it asks the following two subquestions: 1) how would Chroma work in an environment that had heterogeneous servers (i.e., the servers had different computing capabilities), and 2), how would Chroma work in an environment where multiple Chroma clients are all vying for the same set of server resources. Section 5.7 provides the results of this validation.

5.1.1 Client and Server Setup

For the rest of this section, except where stated otherwise, I used HP Omnibook 6000 notebooks with 256 MB of memory, a 20 GB hard disk and a 1 GHz Mobile Pentium 3 processor as the remote servers. I used two different clients that represent the range of computational power available in today's mobile devices. The *fast client* is the above mentioned HP Omnibook 6000 notebook. The *slow client* is an IBM Thinkpad 560X notebook with 96 MB of memory and a 233 MHz Mobile Pentium MMX CPU. The computational power of the Thinkpad 560X is representative of today's most powerful handheld devices. The clients and servers ran Linux and were connected via a 100 Mb/s Ethernet network. I used the Coda [133, 190] distributed file system to share application code between the clients and servers.

5.2 Q1: Determining the Optimal Operation Setting

Since Chroma automatically determines how to remotely execute an application based on the current resources, it is possible that the decisions it makes are not as good as a careful manual remote partitioning of the application. I allay this concern by showing that Chroma's partitioning comes close to the optimal partitioning possible for a number of different applications and operating conditions.

5.2.1 Experiment Setup

To demonstrate this, I compared the decision making of Chroma with that of an ideal runtime system. This ideal runtime system was achieved by manually testing every possible application runtime setting for a given experiment and then choosing the best one. Chroma, on the other hand, has to figure out the best application runtime setting dynamically at runtime.

I defined the best runtime setting as the one that maximizes the given utility function. For this experiment, I used the following simple utility function.

$$utility = \frac{fidelity}{latency} \tag{5.1}$$

Fidelity has no units and represents the application quality. It ranges from 0.0 (worst quality) to 1.0 (best quality). Latency, given in seconds, is the time needed to perform the operation.

I used Panlite (Section 3.8), Janus (Section 3.6), and Face (Section 3.1) as the applications for this experiment. Each experiment was repeated five times and the results are shown with 90% confidence intervals where applicable. Since Chroma uses history-based demand prediction, I created history logs for each application before running the experiments using training data that was not used in the actual experiments. These logs provide the system with the proper prediction values for the application. Without these logs, the system would have to slowly learn the correct prediction values online and this could take a long time and lead to incorrect results. The results for each application is presented separately in the next few sections.

5.2.2 Panlite

5.2.2.1 Description

As mentioned in Section 3.8, Panlite translates text from one language to another. It can use up to three translation engines: EBMT (example-based machine translation), glossary-based, and dictionary-based. Each engine returns a set of potential translations for phrases within the input text. A language modeler combines their output to generate the final translation.

Panlite's fidelity increases with the number of engines used for translation. I assigned the EBMT engine a fidelity of 0.5. The glossary and dictionary engines produce subjectively worse translations— I assigned them fidelity levels of 0.3 and 0.2, respectively. When multiple engines are used, I added their individual fidelities since the language modeler can combine their outputs to produce a better translation. For example, when the EBMT and glossary-based engines are used, I assigned a fidelity of 0.8. The seven possible combinations of the engines are captured by the seven tactics (shown in Fig 3.11).

Each translation engines and the language modeler may be executed either locally or remotely. While execution of each engine is optional, the language modeler must always execute. Thus, there are at least 52 tactic plans (a tactic plan is a specific tactic selection with precise server locations for each RPC) for Chroma to choose from when at least

Sentence Length (No. of Words)	Ideal Runtime chosen tactic	Ideal Runtime utility	Chroma chosen tactic	Chroma utility	Ratio
11	gloss_dict_ebmt	1.00	gloss_dict_ebmt	1.00	1.00
23	gloss_dict_ebmt	1.00	dict_ebmt	0.70	0.70
35	gloss_dict_ebmt	1.00	dict_ebmt	0.70	0.70
47	gloss_dict_ebmt	1.00	gloss_dict_ebmt	1.00	1.00
59	dict_ebmt	0.70	dict_ebmt	0.70	1.00

(a) Fast Client

Sentence Length (No. of Words)	Ideal Runtime chosen tactic	Ideal Runtime utility	Chroma chosen tactic	Chroma utility	Ratio
11	gloss_dict_ebmt	1.00	gloss_dict_ebmt	1.00	1.00
23	gloss_dict_ebmt	1.00	gloss_dict_ebmt	1.00	1.00
35	gloss_dict_ebmt	1.00	dict_ebmt	0.70	0.70
47	dict_ebmt	0.70	dict_ebmt	0.70	1.00
59	dict_ebmt	0.70	dict_ebmt	0.70	1.00

(b) Slow Client

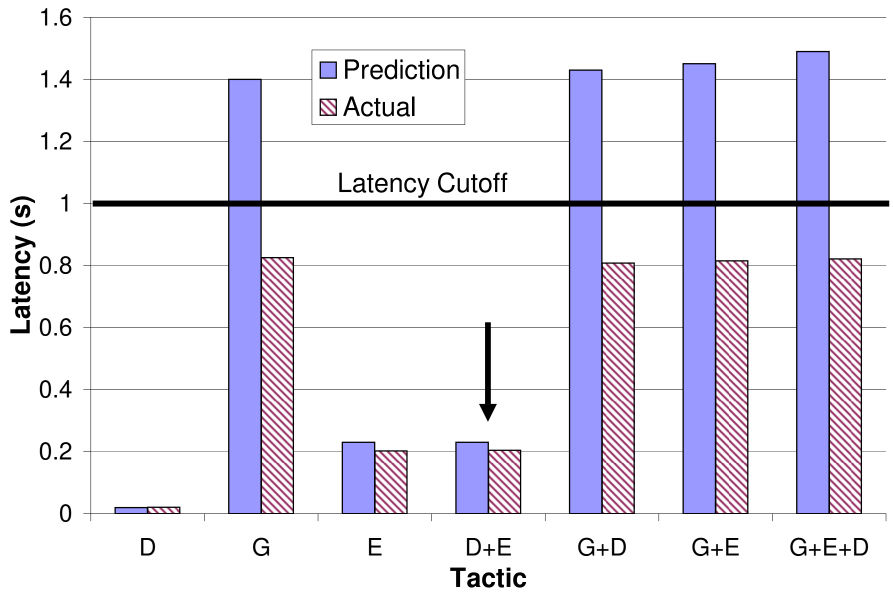
The two tables show the exact tactic and the utility value of the runtime setting chosen by Chroma and the ideal runtime. The locations chosen by Chroma and the ideal runtime were identical in all cases and are thus omitted from the table. The ratio ($\frac{Chroma}{Ideal}$) between the ideal system's utility value and Chroma's is shown in the Ratio column. Any value less than 1.0 indicates that Chroma chose an inefficient runtime setting.

Table 5.1: Comparison Between the Ideal Runtime and Chroma for Pangloss-Lite

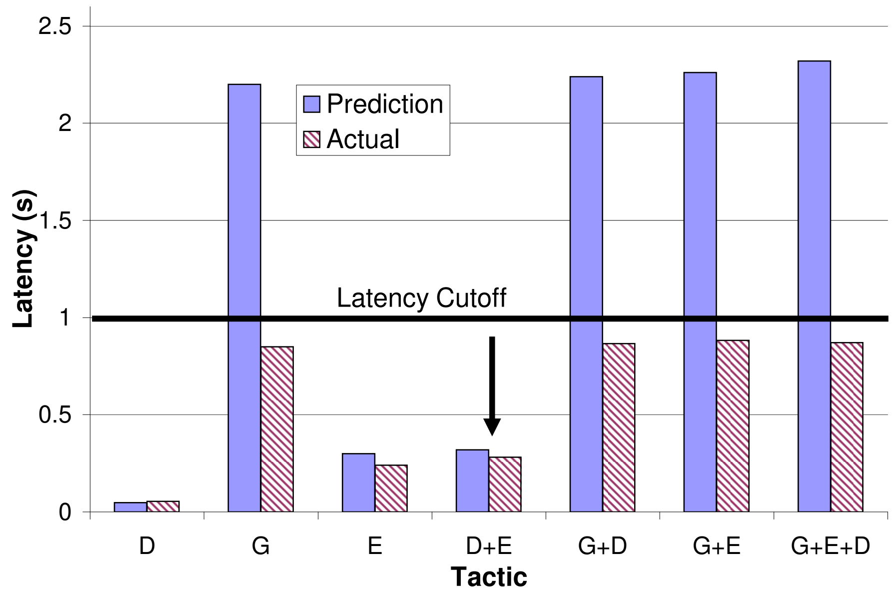
one remote server is available. Since Panlite has no fidelity variables, these tactic plans represent the possible runtime settings.

I used the simple utility function (Equation 5.1) to determine the runtime setting to use for Panlite. However, to model the preferences of an interactive user, I specified that all predicted latencies of one second or lower are equally good and that all predicted latencies larger than five seconds are impossibly bad. Thus if the prediction is greater than five seconds, I set the utility to zero and if the prediction is one second or lower, I changed its value to one (the final utility thus depends solely on the fidelity). All other predicted latency values were left unchanged.

I used as input five sentences with different number of words (ranging from 11 words to 59 words) as inputs for the baseline experiments. The input sentences were in Spanish and were translated into English. There were three remote servers available and both the servers and the clients were unloaded for the purposes of this experiment.



(a) Fast Client



(b) Slow Client

These 2 figures show the predicted latency for each of the 7 tactics (D = dictionary, E = ebmt, G = gloss) as well as the actual latency for each of those tactics. From the figures, we see that for both clients, the predicted latency for executing the glossary engine was much higher than the actual latency. Thus, Chroma chose the best tactic (shown by the arrow) that it thought would maximize the utility function. The 1s latency cutoff is the point below which all latencies were considered equally good. I.e., the point below which only fidelity values affected the final utility score.

Figure 5.1: Difficulty with Predicting Sentences with 35 Words

5.2.2.2 Results

Table 5.1 shows the decisions made by Chroma and the ideal runtime, for all five sentences, on the fast and slow clients respectively. From the results, we see that Chroma made decisions that approximated the decisions made by the ideal runtime system. In the cases where Chroma made a different decision, it was off by 30%. This difference in decision making was due to incorrect resource estimations by Chroma. From the results, we see that Chroma decided not to run the `glossary` engine in the cases where it differed from the ideal runtime. The time needed for the glossary engine to complete a translation was hard for Chroma to predict as it was not a simple function of the length of the input sentence. This prediction difficulty is shown in Figure 5.1. Chroma's decision to drop the `glossary` engine incurred a 30% reduction in fidelity and this resulted in the final 30% difference in Ratio. The latencies used to calculate the utility value were below 1 second for both Chroma and the ideal runtime system in all the cases where the utility values differed.

5.2.3 Janus

5.2.3.1 Description

As mentioned in Section 3.6, Janus performs speech recognition and converts spoken phrases to text. This recognition can be performed at either full or reduced fidelity. The reduced fidelity uses a smaller, more task-specific vocabulary that limits the number of phrases that can be successfully recognized but requires less time to recognize a phrase. Janus has a fidelity variable that determines which vocabulary files to use. I assigned the reduced fidelity a utility of 0.5 and the full fidelity a utility of 1.0 to reflect this behavior. Similar to Panlite, I modeled an interactive user by making all predicted latencies less than or equal to one second equally good (we set the predicted latency value to one) and all latencies greater than five seconds horribly bad (I set the utility value to zero). All other latency values were left unchanged.

Janus has two tactics, as shown in Figure 3.9, that can be executed either locally or remotely. The tactic `Janus_Full` uses a single RPC while the `Janus_Hybrid` uses two RPCs. `Janus_Full` has a lower latency than `Janus_Hybrid` but uses more battery energy.

I used as input ten different utterances containing different numbers of spoken words (ranging from 3 words to 10 words) as inputs for the baseline experiments. One remote server was used for this experiment and both the server and the clients were unloaded.

5.2.3.2 Results

Table 5.2 shows the decisions made by the ideal runtime and Chroma. In all cases, the ideal runtime and Chroma both chose the `Janus_Full` tactic as it had the lower latency (and battery conservation was not a factor). The table lists, for both the fast and slow client, the fidelity variable setting chosen by both Chroma and the ideal runtime along with the actual execution latency for each of the 10 input sentences.

Utterance	Ideal Runtime			Chroma			Ratio
	Fidelity Value	Latency	Utility	Fidelity Value	Latency	Utility	
1	reduced		0.50	reduced		0.50	1.00
2	reduced		0.50	reduced		0.50	1.00
3	reduced		0.50	reduced		0.50	1.00
4	reduced		0.50	reduced		0.50	1.00
5	reduced		0.50	reduced		0.50	1.00
6	reduced		0.50	reduced		0.50	1.00
7	full		0.53	reduced		0.50	0.94
8	reduced		0.50	reduced		0.50	1.00
9	reduced		0.50	reduced		0.50	1.00
10	reduced		0.50	reduced		0.50	1.00

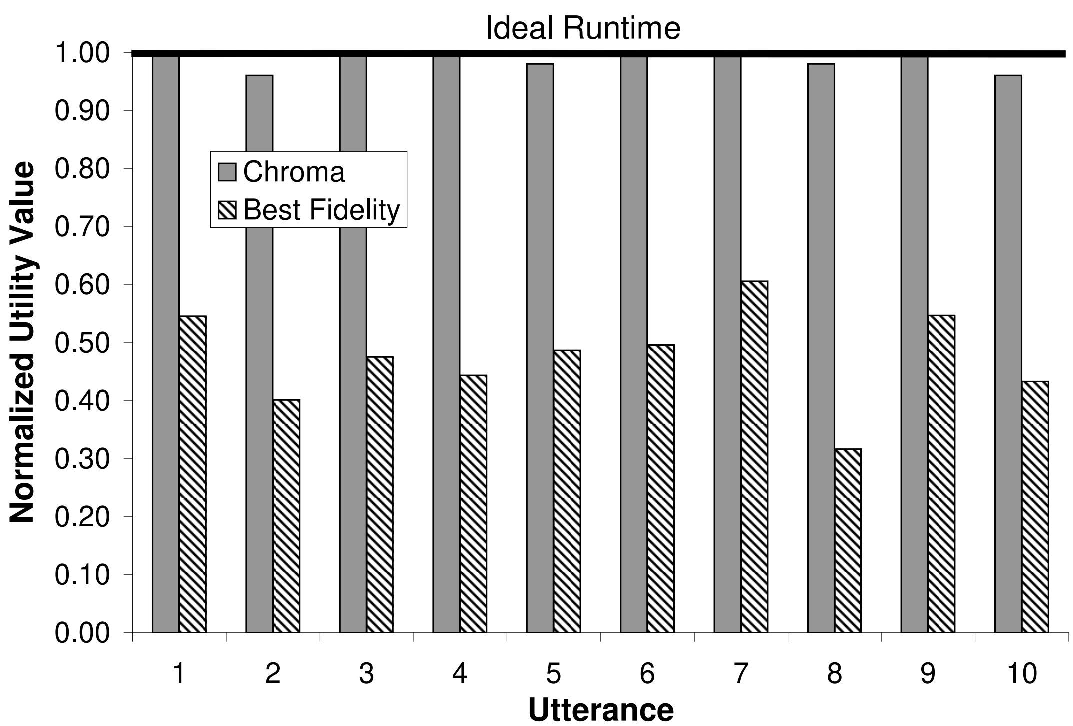
(a) Fast client

Utterance	Ideal Runtime			Chroma			Ratio
	Fidelity Value	Latency	Utility	Fidelity Value	Latency	Utility	
1	reduced	0.71	0.50	reduced	0.73	0.50	1.00
2	reduced	1.00	0.50	reduced	1.00	0.48	0.96
3	reduced	0.76	0.50	reduced	0.80	0.50	1.00
4	reduced	0.78	0.50	reduced	0.79	0.50	1.00
5	reduced	0.99	0.50	reduced	1.03	0.49	0.98
6	reduced	0.95	0.50	reduced	0.96	0.50	1.00
7	reduced	0.70	0.50	reduced	0.71	0.50	1.00
8	reduced	1.20	0.42	reduced	1.22	0.41	0.98
9	reduced	0.77	0.50	reduced	0.77	0.50	1.00
10	reduced	0.99	0.50	reduced	0.99	0.48	0.96

(b) Slow Client

These two tables show the exact tactic and the utility value of the runtime setting chosen by Chroma and the ideal runtime. The locations chosen by Chroma and the ideal runtime were identical in all cases and are thus omitted from the table. The ratio ($\frac{Chroma}{Ideal}$) between the ideal system's utility value and Chroma's is shown in the Ratio column. Any value less than 1.0 indicates that Chroma chose an inefficient runtime setting.

Table 5.2: Comparison Between the Ideal Runtime and Chroma for Janus



This figure shows the normalized utility ratio of the choices made, for the slow client, by the Ideal Runtime, Chroma, and a theoretical runtime that always uses the runtime settings that have the best fidelity. As can be seen, picking the highest fidelity choice results in a far lower utility.

Figure 5.2: Relative Utility of Different Operation Settings for Janus

We see that Chroma picked the optimal choice in almost all cases on the fast client. Even in the case where Chroma picked a different tactic plan, the utility value of the plan picked by Chroma was very close to optimal (94% of optimal). On the slow client, Chroma performed as well as the ideal runtime. In all cases, Chroma picked the same tactic plan as the ideal runtime system and the differences in the utility value were due to experimental errors in the latency measurements. In particular, as Figure 5.2 demonstrates (for the slow client), Chroma did not pick the runtime setting with the best fidelity as this would have been suboptimal.

5.2.4 Face

5.2.4.1 Description

Face (Section 3.1) is a program that detects human faces in images. It is representative of image processing applications of value to mobile users. Face can potentially change its fidelity by degrading the quality of the input image. However, for the purposes of this experiment, all experiments were run with full fidelity images.

Face has a single tactic, as shown in Figure 3.2, with a single RPC that can be run either entirely locally or entirely remotely. Even though Face has only one tactic, this does not mean that it cannot benefit from tactics. I show in Section 5.5.3 how Chroma can use this single tactic to improve the performance of Face by using extra resources in the environment.

I used as input three different image files of different size (ranging from 133 KB to 621 KB in size) as inputs for the baseline experiments. There was one remote server available and both the server and the clients were unloaded.

5.2.4.2 Results

Figure 5.3 shows the latency that can be achieved when doing the face recognition locally and remotely for both configurations. Since the fidelity was constant (full quality images) in all the experiments, maximizing the utility function would require Chroma to pick the option that minimized the latency. We see that in all cases, Chroma chose the correct option.

The graphs show that Face has extremely high latencies; on the order of tens of seconds per image. In Section 5.5.3, I show how data decomposition can be used to reduce this latency.

5.2.5 Q1: Summary

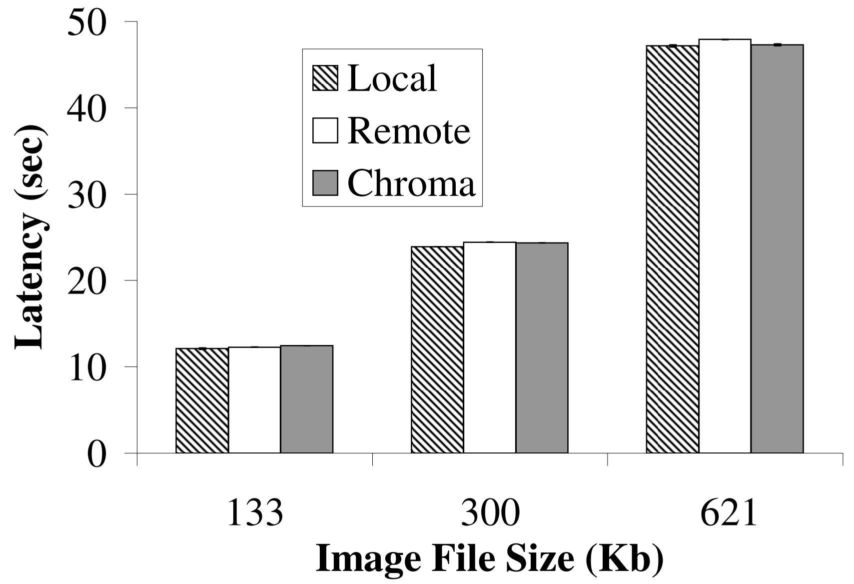
Sections 5.2.2, 5.2.3 and 5.2.4 described the performance of Chroma relative to an ideal runtime system for Panlite, Janus and Face respectively. We see that while Chroma is not perfect, its performance is still comparable to an ideal runtime system. Hence, Chroma is able to pick the appropriate runtime setting for an application.

5.3 Q2: Can Chroma Perform Well in Dynamic Environments?

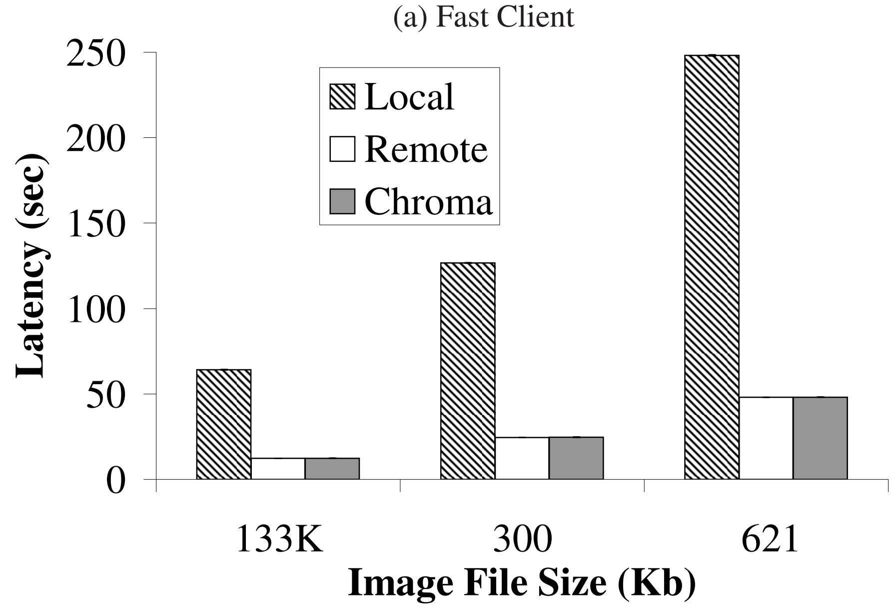
In this section, I answer the second question posed at the start of this chapter (Section 5.1). Namely, can Chroma work effectively in a dynamic environment? To effectively answer this question, I answered the following two more focused questions.

1. **Why is a default utility function not good enough?** In this question, I investigated if it was really necessary for Chroma to dynamically change its optimization functions to accommodate changing user goals. In particular, would it be possible to satisfy changing user goals by using just a single good default optimization function? This question is answered in Section 5.3.2

2. **Can Chroma provide the correct behavior even when user preferences dynamically change?** In this question, I investigated whether Chroma, when provided with accurate representations of the current user preferences, can pick a runtime setting



(a) Fast Client



(b) Slow Client

The latency that was achieved by executing Face remotely and locally for all inputs on both clients is shown. In all cases, Chroma picked the option that minimized latency. This maximized the utility function as the fidelity was constant in all cases.

Figure 5.3: Relative Latency for Face

that satisfies the user and the given resource environment. I also measured if the runtime setting chosen is optimal. This question is answered in Section 5.3.3.

The rest of this section is structured as follows; Section 5.3.1 presents the experimental setup used for the system validation. The answers to the two subquestions posed above are presented in Sections 5.3.3 and 5.3.2 respectively.

5.3.1 Experiment Setup

To answer the two focused questions, I used the following scenario: I assumed that a user is performing language translation using a computationally limited PDA. To overcome the PDA's computational limitations, Chroma is able to use faster remote servers, if they are available, to perform the language translations. Remote servers are available for use in a *rich* resource environment while a *poor* environment has no available servers. To effectively simulate these two environments, I used a single remote server and the slow client (hardware described in Section 5.1.1). When the resources were poor, the remote server was unavailable.

I used Panlite as the language translation application. Panlite is capable of performing both English to Spanish and Spanish to English translations. For the experiment, I used 3 randomly selected sentences, of between 10-12 words in length. Each sentence was translated from English to Spanish and then back into English. I measured the time taken to complete the two translations and the accuracy of the resulting English re-translation. Each sentence was translated five times for every experiment.

The accuracy of the translation was computed based on which tactic was used to perform the translation. Each tactic has a different fidelity which affects the output quality. However, the difference in quality between tactics cannot be easily discerned by a user. To characterize this, I mapped the seven possible fidelity level (one for each tactic) into just three user-perceived accuracy levels. Using just a single translation engine (the `gloss`, `dict`, and `ebmt` tactics) resulted in low accuracy translations. Using a tactic with 2 translation engines (the `gloss_dict`, `gloss_ebmt`, and `dict_ebmt` tactics) resulted in medium accuracy translation while picking the single tactic (`gloss_dict_ebmt`) that used all three translation engines resulted in high accuracy translations. Higher accuracy translations require Panlite to do more work and hence, take longer.

At runtime, Chroma would pick a runtime setting that optimized the utility functions (which factored in user preferences and the available resources). The translation would then be characterized as low, medium or high depending on which tactic Chroma picked. For this experiment, once Chroma had decided, for a given sentence, the runtime setting to use for the English to Spanish translation, it would use the same choice for the Spanish to English translation.

User Preference	Response Time	Accuracy
Accurate	Upper - 3.2s Lower - 0.6s	High Accuracy - Highly Desired Medium Accuracy - Somewhat Desired Low Accuracy - Not Desired
Fast	Upper - 1.4s Lower - 0.3s	High Accuracy - Highly Desired Medium Accuracy - Highly Desired Low Accuracy- Somewhat Desired

The table shows the tradeoffs in the accurate and fast preferences used for these experiments. The response time column shows the response time below which the user would be very happy (lower value) and the response time above which the user would be completely unhappy (upper value). Response times in between these two points would be linearly interpolated to calculate how favorable they were to the user. For accuracy, the table lists how desired different accuracy translations were to the user. These preferences were formed into a utility function that was used to calculate a utility score for each possible runtime setting.

Table 5.3: Tradeoffs Used in User Preferences

	Avg. Response Time (s)	Accuracy
Fast Template	0.32	Medium
Accurate Template	0.49	High

(a) Rich Resource Environment

	Avg. Response Time (s)	Accuracy
Fast Template	0.72	Low
Accurate Template	2.86	Medium

(b) Poor Resource Environment

The tables show the average response time (in seconds) and accuracy for translations performed by Chroma under different resource conditions. For each condition, Chroma was first provided with the fast preference that preferred fast responses over accuracy. Chroma was then provided with the accurate preference that preferred accuracy over response time. For each preference, Chroma translated 3 sentences of between 10-12 words in length 5 times each.

Table 5.4: Performance of Chroma with Different Preference Tradeoffs

	Chroma's Selection	Lower Accuracy Answer	Higher Accuracy Answer
Fast Template (Poor Resources)	1.0	N/A[a]	0.0[b]
Accurate Template (Poor Resources)	1.0	0.37	0.0[b]
Fast Template (Rich Resources)	1.0	0.51	0.83
Accurate Template (Rich Resources)	1.0	0.5	N/A[c]

[a]Chroma selected the lowest accuracy result as its choice
[b]The response time exceeded the maximum allowable value for the template
[c]Chroma selected the highest accuracy result as its choice

The table compares Chroma's choices (the raw values are presented in Figure 5.4) against other possible choices (that would result in different accuracies and response times). I computed a utility value (using the appropriate user preference) for all the choices and normalized them against Chroma's choice. Any choice with a normalized value of greater than 1.0 would be better than Chroma's choice. For each case, Chroma translated 3 sentences of between 10-12 words in length 5 times each.

Table 5.5: Optimality of Chroma's Choices

5.3.2 Is A Default Utility Function Good Enough?

In this sub-section, I answer the question of whether Chroma can satisfy changing user preferences by using just a single default utility function (the first sub-question posed at the start of this section). If the answer to this question is "yes", then Chroma's design can be greatly simplified. However, if the answer is "no", then Chroma needs support for dynamic utility functions.

For this experiment, I used only the rich resource environment and provided Chroma with three different static utility functions. They were a) a function that preferred accuracy over response times (I used the logic of the *Accurate* preference from Section 5.3.3), b) a function that preferred response times over accuracy (I used the *Fast* preference logic from Section 5.3.3), and c) a function that chose the runtime setting that returned the fastest translation. For each case, I measured how well each static function was able to satisfy two different user goals. One goal, called *min-latency*, required response times of less than 0.2 seconds (accuracy was irrelevant). The other goal, called, *max-latency*, wanted high accuracy answers (response time was irrelevant). The results of this experiment are shown in Table 5.6. We see that none of the static preferences was able to optimally satisfy both user goals. I claim that, in general, it is impossible for an adaptive system to adequately support changing user goals if it uses a single fixed optimization function. Hence, for Chroma to be effective in mobile environments, it needs to use dynamic and correct utility functions that accurately capture the current user preferences.

5.3.3 Chroma Can Support Changing User Preferences

In the previous subsection (Section 5.3.2), it was shown that Chroma cannot effectively support changing user goals with a single default utility function. In this subsection, I answer the question of whether Chroma can be effective in a dynamic environment (the second sub-question posed at the start of this section) where user preferences can change dynamically.

The first challenge for Chroma is obtaining dynamic utility functions that encapsulate these changing user preferences. To obtain these dynamic functions, I integrated Chroma with Prism [199]. Prism is a user level program that monitors the user and generates utility functions that accurately capture the user's current preferences. Prism, and the integration of Chroma with Prism, is described in more detail in Appendix A.5.

For the rest of this section, I used two different user preferences. The first preference, called *Accurate* preferred accuracy over response time (latency) while the second preference, called *Fast*, preferred response time over accuracy. Table 5.3 lists the exact tradeoffs expressed in each preference. Chroma was provided, by Prism (Appendix A.5), detailed utility functions that encapsulated each of these preferences.

For each translation, Chroma would pick the runtime setting that had the highest utility value. Table 5.4 shows the raw results for the choices made by Chroma for both preferences and for both the poor and rich resource environments. In all cases, Chroma was able to pick a solution that satisfied the given preference and environment.

I then compared Chroma's choices against other possible choices to determine if Chroma was operating optimally. I obtained these other choices by translating the test sentences using runtime settings that would result in higher and lower accuracy answers than the choice made by Chroma. For example, if Chroma's choice resulted in a medium accuracy answer, I translated the same sentence using a runtime setting that would result in low and high accuracy answers. Effectively, I am comparing Chroma's choice against all possible other choices that could have been made. I ran this experiment using both resource environments and both preferences. For each preference, I computed a utility value for every choice (both Chroma's decision and the other choices). I normalized all the scores against the choice made by Chroma for that particular preference and resource environment (i.e., Chroma's choice is set to 1.0). Any score above 1.0 indicates a better choice that should have been chosen by Chroma. Table 5.5 shows the results of this experiment. As can be seen, Chroma selected the optimal choice, in each resource condition, for both preferences. These results verify that Chroma is effective at choosing the appropriate application settings even in a dynamic environment.

5.3.4 Q2: Summary

In this section, I first showed that in environments where user preferences can change dynamically, such as mobile environments, it is vital that Chroma use accurate utility functions that capture the current user preferences. Otherwise, Chroma's performance will not be optimal. I then showed that Chroma, when provided accurate utility functions from sys-

	Average Response Time (s)	Accuracy	Satisfies Max-Accuracy Goal?	Satisfies Min-Latency Goal?
Prefer Accuracy	0.49	High	Yes	No
Prefer Response Time	0.32	Medium	No	Yes
Lowest Response Time	0.09	Low	No	Yes

The table shows the average response time (in seconds) and accuracy for translations performed by Chroma using different static optimization functions. For each function, the table also lists if the function satisfied the two different user goals of max-accuracy and min-latency. For each function, Chroma translated 3 sentences of between 10-12 words in length 5 times each. This experiment was run with rich resources.

Table 5.6: Performance of Chroma with Different Static Optimization Functions

tems like Prism, can choose the optimal runtime setting even in a dynamic environment where the resources and the user preferences change.

5.4 Q3: Chroma's Overhead

In this section, I present the complete overhead of Chroma's decision making process. This includes the resource prediction and resource measurement overheads. This answers the third question posed in Section 5.1. Namely, is the overhead of Chroma's decision making process reasonable.

To decide on the appropriate runtime settings for an application, Chroma uses an exhaustive brute force solver algorithm (details presented in Appendix A.6). A key concern with this brute force exhaustive solver is that it could take too long to solve for an optimal runtime setting. This proves to be not the case. Figure 5.4 shows the time taken (in milliseconds) by the solver to decide the optimal runtime setting. To obtain these results, I extracted the core solver from Chroma and supplied it with synthetic inputs. Each synthetic tactic had 3 RPC stages and the experiment was conducted on a 1 Ghz Mobile Pentium 3 processor. The experiment showed that the solver is not a bottleneck as it takes less than 1ms even for a large number of tactics and servers.

This solver overhead does not include other Chroma components such as the resource measurers and resource predictor. The total end-to-end measured overhead of Chroma is presented next. In the rest of this section, I use Panlite as the test application as it had the most tactics and thus required Chroma to do the most decision making. I therefore do not present the end-to-end overhead results for other applications as they were all strictly less than the overhead incurred for Panlite.

Figure 5.5 shows the maximum overhead of Chroma's decision making. This overhead represents the maximum time that Chroma needs to determine the appropriate runtime setting. Minimizing this overhead is crucial as Chroma currently does not take its own

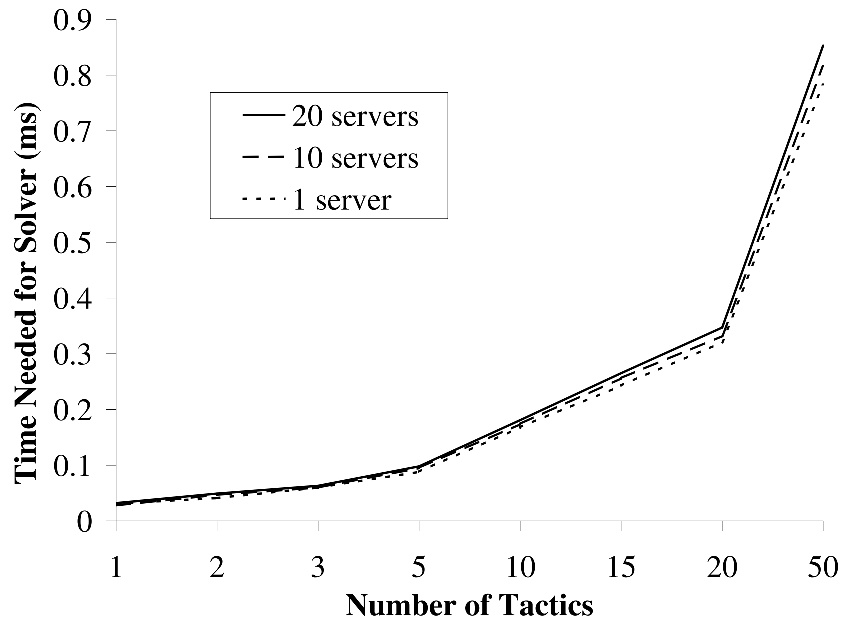


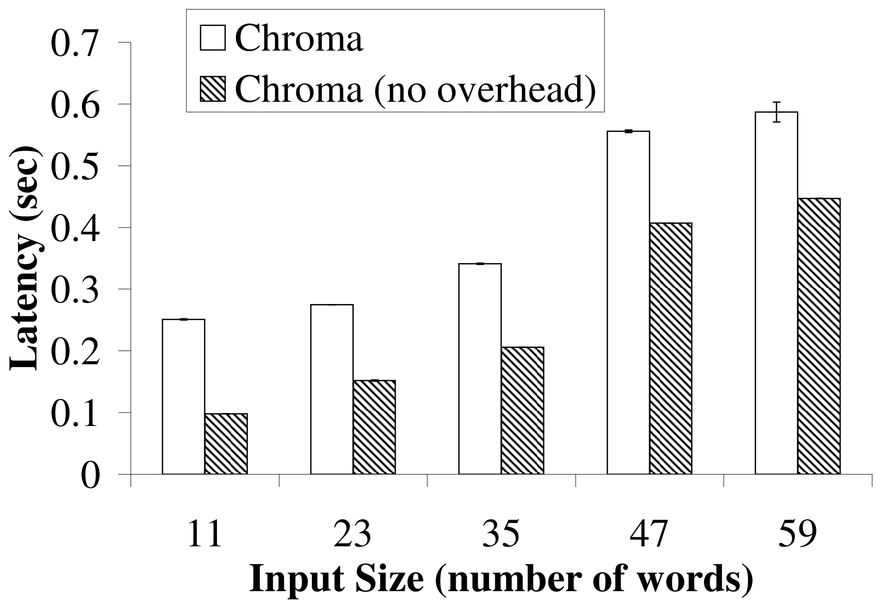
Figure 5.4: Overhead of the Chroma Solver

overhead into account when making placement decisions. Chroma can thus achieve longer latencies than expected. This is more apparent on slower clients as it takes longer for Chroma to make its decisions on these computationally weaker clients. From the figure, we see that Chroma's maximum overhead was ≈ 0.15 seconds for the fast client and ≈ 0.8 seconds for the slow client.

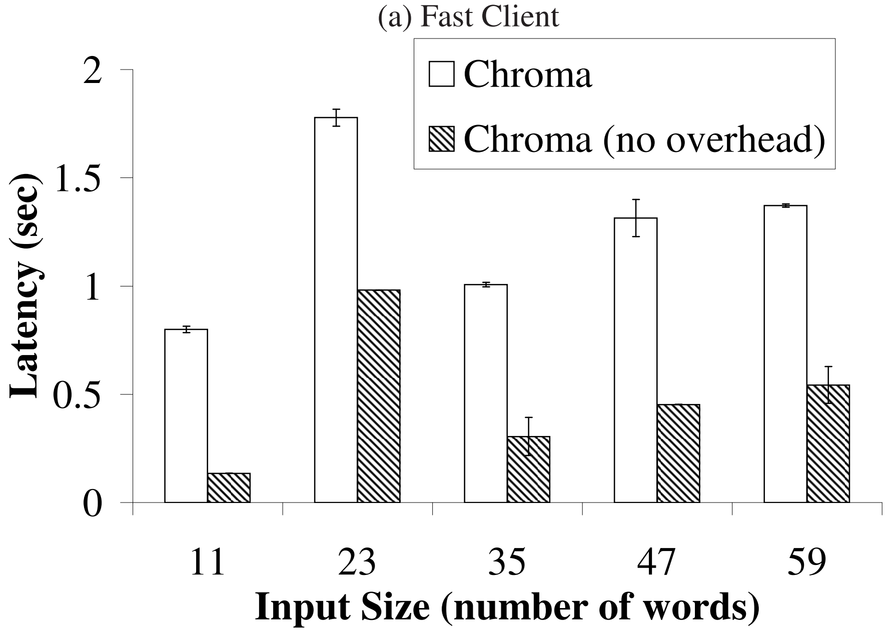
Figure 5.6 shows the breakdown of the overhead. We see that most of it ($\approx 80\%$) came from the resource measurers. In particular, the overhead arose from measuring the available resource on remote servers. This arose due to two factors; 1) Chroma uses a very inefficient sequential measuring mechanism. In particular, if Chroma needs to measure the resource availability on ten servers, it does them one by one instead of in parallel, and 2) Chroma doesn't cache resource values. Hence, every operation incurs the full resource measurement time. Both of these inefficiencies can be reduced. First, Chroma can be modified to perform resource measurements in parallel. Second, Chroma could use cached resource values for subsequent operations. The tradeoff with using cached results is that it may result in inefficient decisions if the resource availability has changed.

5.4.1 Q3: Summary

In summary, Chroma's decision making overhead adds, in the worst case, up to 0.8 seconds of latency (for slower clients) to application execution times. Fortunately, this high



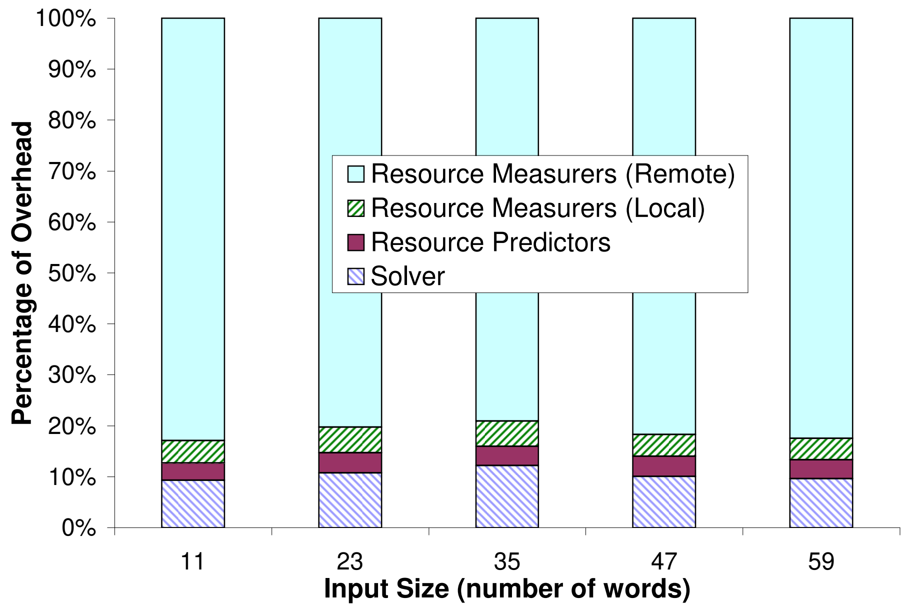
(a) Fast Client



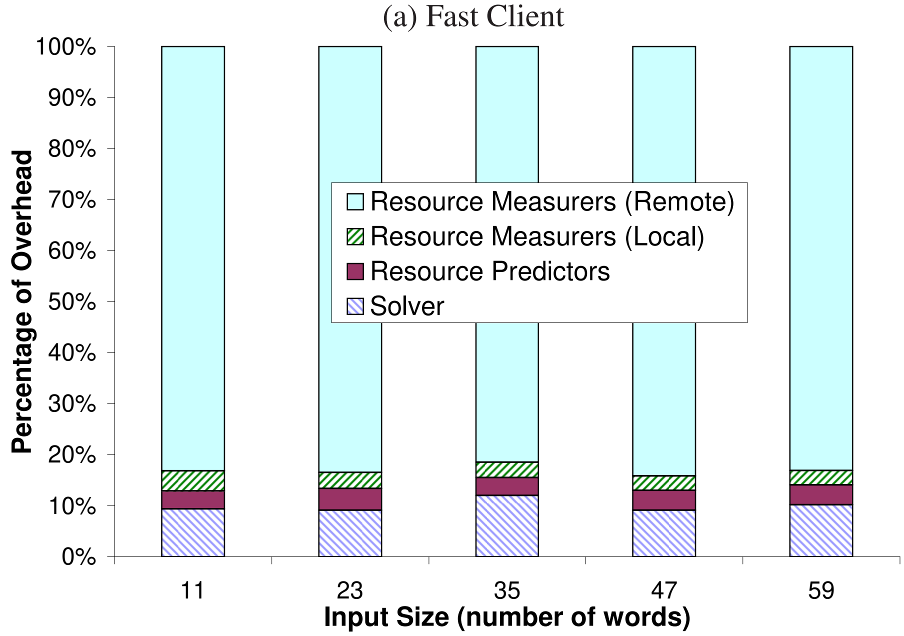
b) Slow Client

The bars show the time needed for different tactic plans to execute with and without Chroma's decision making process. The difference in time represents the overhead of Chroma's decision making process. The tactic plans used in this experiment are the same ones Chroma chose in Figure 5.1 for the different inputs. The results are the average of 5 runs and are shown with 90% confidence intervals.

Figure 5.5: Overhead of Decision Making for Panlite



(a) Fast Client



b) Slow Client

The bars show how the percentage contribution of each of Chroma's three major components to the total overhead. The solver overhead is the time needed to execute the nested loops of the solver (as described in Appendix A.6). The resource prediction overhead is the total time needed to determine the predicted resource usage of each runtime setting. The resource measuring overhead represents the time needed to accurately measure the available resource both locally and on remote machines. The bars for the solver include the time needed to discover the available servers in the environment. That is the reason why the solver time is an order of magnitude larger than the times shown in Figure 5.4.

Figure 5.6: Breakdown of Chroma's Decision Making Overhead

latency overhead can be significantly reduced if clients are willing to use cached resource measurements.

5.5 Q4: Overprovisioned Environments

In this section, I answer the third question posed in Section 5.1. Namely, can Chroma opportunistically use extra resources in the environment to improve application performance.

So far, my focus has been on environments that are resource constrained. However, there are cases where the assumption of resource constrained environments proves false. For example, there are already many environments, such as smart rooms and computer laboratories, that have an abundance of server resources by design. There may also be cases where due to unexpected underutilization, additional server resources become available. For example, a computing environment was designed to handle a peak load of X requests but the average load is just Y requests (where $Y < X$) which results in lucky occasions when the computing infrastructure is underutilized.

We refer to these kinds of environments as being *over-provisioned*. Over-provisioned environments are characterized as having more computing resources, either because of their nature or by lucky underutilization, than are actually needed for normal operation. For example, an application can use at most five servers but the environment provides 30. I would also like Chroma to work well in the anticipated common case where resources are scarce but also be able to automatically make use of over-provisioning if it becomes available. In particular, by using extra idle servers, I hope to improve the user's experience.

Tactics facilitate this dual modality by providing precise knowledge of the remote calls needed by a given operation and the data dependencies between them. Chroma can then use this information to opportunistically use extra resources in three different ways.

First, Chroma can make multiple remote execution calls (for the same operation) to different remote servers and use the fastest result. For example, Chroma can execute RPC $step\_1$ of tactic do_simple (Figure 2.4) on multiple servers and use the fastest result. Chroma knows that it can do this safely because the description of the tactics makes it clear that executing RPC $step\_1$ is a stand-alone operation and does not require any previous results or state. I call this optimization method *fastest result*.

Second, Chroma can perform the same operation but with different fidelities at different servers. Chroma can then return the highest fidelity result that satisfies the latency constraints of the application. For example, Chroma can execute multiple instances of RPC $step\_1$ in parallel at separate servers (all with different fidelity variable settings) and use the highest fidelity result that returns within a specified amount of time. I call this optimization method *best fidelity*.

Finally, Chroma can split the work necessary for an operation among multiple servers. It does this by decomposing the input data, using application provided logic, into smaller chunks and shipping each chunk to a different remote server. The partial results are then recombined, again using application provided logic, to form the final output. I call this optimization *data decomposition*.

Tactics allow us to use these optimizations on behalf of applications automatically without the applications needing to be re-compiled or modified in any way. There are other optimizations possible with tactics, but these are the ones I have explored so far and I present performance results for them in the following subsections.

These results used the same hardware setup described in Section 5.1.1. However, I only used the slow client as it could benefit the most from using extra resources. The results show that extra servers can be used in the three ways described above to:

- Hedge against load spikes at the remote servers: the same operation can be run on multiple servers using the "fastest result" method.
- Satisfy absolute latency constraints of an application while providing the best possible fidelity: the operation can be run at different servers (where each server runs the operation at a different fidelity) using the "best fidelity" method and the best fidelity result that returns within the latency constraint is returned to the application.
- Improve the total latency of an operation without sacrificing fidelity: the operation can be broken up into smaller parts using the "data decomposition" method and each smaller part run on a separate server.

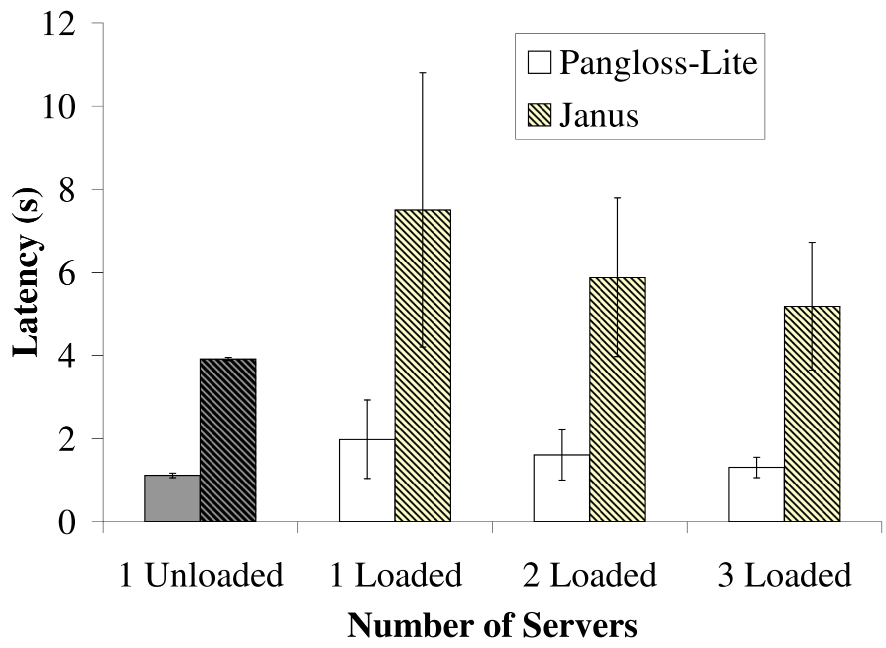
5.5.1 Hedging Against Load Spikes

5.5.1.1 Description

This experiment shows how opportunistically using extra servers in the environments provides protection against random load spikes at any particular remote server. The overall scenario I are assuming for this section is as follows; Chroma has decided where to remotely execute an application component. At the time it made the decision, Chroma noticed that the remote server was capable of satisfying the latency requirements of the operation. However, when the operation was actually executed, the actual average latency was much higher due to the random load on the server that Chroma was unaware of. I show results to quantify just how bad the average latency (and variance) becomes and how opportunistically using extra servers in the environment can help improve this.

To show the benefits of this approach, I introduced an artificial load on the server that Chroma selected to remotely execute application components. This artificial load has an average load of 0.2 (i.e., on average, each CPU was utilized only 20% of the time). However, the actual load pattern itself is random. I chose a random load pattern to model the uncertainty inherent in mobile environments where remote servers could suddenly perform worse than expected due to a variety of random reasons (such as bandwidth fluctuations, extra load at the server etc.). The average load was set at 0.2 to ensure that the servers were, on average, underutilized. In contrast, a load of 0.8 (the CPU was utilized 80% of the time) or higher would indicate a heavy load.

In this experiment, Chroma decides to execute the glossary engine of Panlite remotely to translate a sentence containing 35 words. I ran the translation of this sentence 100 times using a different number of remote servers (with different loads) in parallel and noted the



This figure shows the use of multiple loaded servers to improve the performance of Panlite and Janus. As the number of loaded servers is increased, the latency and standard deviation for both applications decrease significantly and converge towards the best-case value (1 unloaded server).

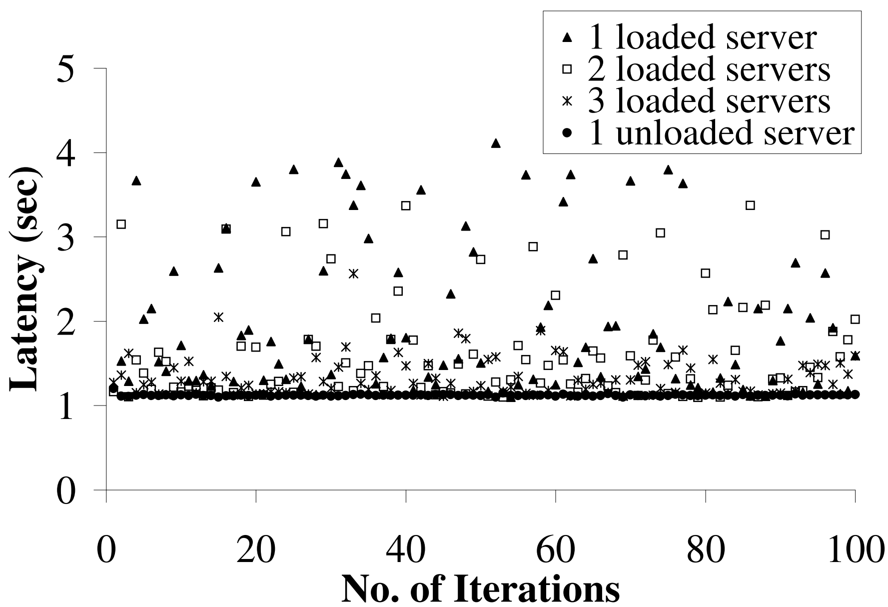
Figure 5.7: Using Extra Loaded Servers to Improve Latency

average latency achieved and the standard deviation. I also repeated this experiment using Janus.

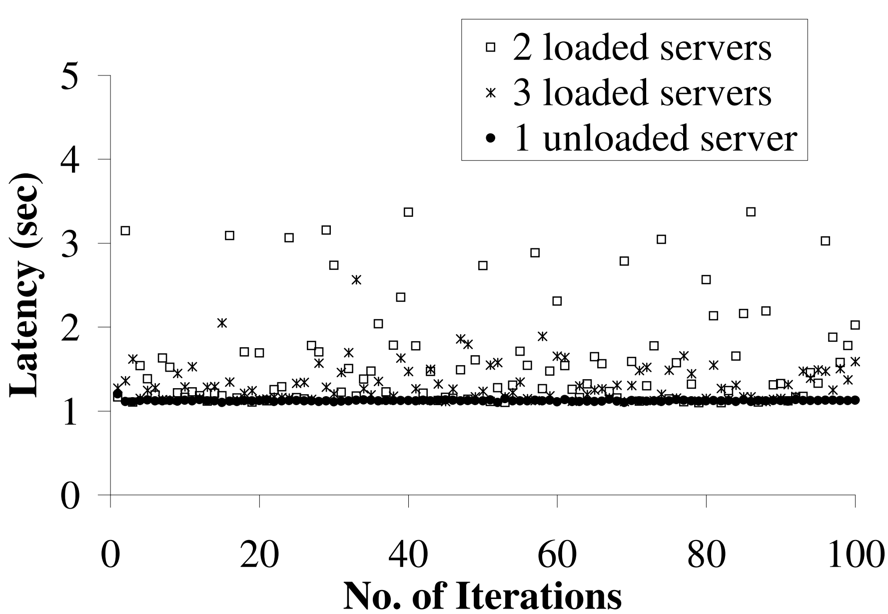
5.5.1.2 Results

Figure 5.7 shows the results I obtained from executing the glossary engine remotely on a totally unloaded server and from executing the glossary engine remotely on one, two and three servers respectively that had the artificial load explained earlier. Figure 5.7 also shows the results for Janus where the recognition of utterance 5 is performed multiple times on remote servers.

The results for the totally unloaded server present the best possible average latency and standard deviation. What we notice is that when the remote server is loaded, executing the glossary engine or recognition remotely at the server results in a much higher average latency and standard deviation. We also notice that executing the glossary engine or recognition on two remote servers that are randomly loaded (with the same average load) reduces the latency and standard deviation significantly compared with the single loaded server case. Executing the glossary engine or recognition on more loaded remote servers reduces the average latency and standard deviation even further and brings them closer to



(a) Results for 1, 2 and 3 extra loaded servers



(b) Results for 2 and 3 extra loaded servers

These 2 figures show the exact latency measurements when using loaded servers. The top figure shows all the measurements while the bottom figure removes the measurements obtained when using just 1 loaded server. The baseline result (using an unloaded server) is the almost solid black line at the bottom of both figures.

Figure 5.8: Use of extra loaded servers to improve latency for Panlite

the best possible results. This effect is shown in more detail in Figure 5.8. Each data point in the figure is a latency measurement. The top figure shows all the results while the bottom figure removes the results obtained when using just 1 loaded server. Comparison of the two figures clearly shows that using more loaded servers significantly reduces the average and standard deviation of the latency.

The reduction in latency caused by using extra servers with load was due to the load on the servers being uncorrelated. Hence, even though the average load on the servers was the same, when one server was experiencing a load spike, another server was unloaded and was able to service the request faster. This method of using extra servers thus maximizes the probability of being able to execute the application component at an unloaded server.

This assumption of uncorrelated load is reasonable in a mobile environment for the following reasons: if the remote servers are located in different parts of the network, it is quite likely that they experience different load patterns. This is also true for remote servers that are co-located but owned by different entities. In the case where the remote servers are co-located and owed by the same entity, it is possible that they experience the same load patterns. However, in this case, enabling some sort of Ethernet-like backoff system on the remote servers will ensure that the load on each server is uncorrelated.

Of course, if every Chroma client is sending extra requests to every available server, the assumption that the load on each server is uncorrelated will not be true. Cheng [45] showed that if the environment has enough resources, using a simple randomized server selection policy can significantly reduce the probability of clients using the same servers. The study of additional server selection and fair sharing mechanisms is beyond the scope of this dissertation.

5.5.2 Meeting Latency Constraints

5.5.2.1 Description

Chroma allows an application to specify a latency constraint for a given operation. This is frequently required for interactive applications to meet user requirements. Chroma looks at the tactics for the application and automatically decides how to remotely execute this operation in parallel with different fidelity values for each parallel execution. For example, for Panlite, Chroma could chose to execute the dictionary, gloss and ebmt translation engines on separate servers. When the latency constraint expires, Chroma picks the completed result with the highest fidelity and returns that to the application.

5.5.2.2 Results

I present results for Panlite to show experimentally the benefits of this approach. For this experiment, I assumed that the application has specified a latency constraint of 1 second. There were three remote servers available for Chroma to use. I used a sentence of 35 words as input. I loaded all the servers with a random load of average value 0.2. I ran each experiment 5 times.

	Fidelity	Latency		Metric
		Average (s)	Standard Deviation (s)	
Running to Completion	1.0	1.96	0.15	0.51
Taking Best Result after 1s	0.75	1.00	0.01	0.77

The table shows the latencies and fidelities obtained by running all three translation engines (dict, gloss, ebmt) on the input on loaded servers. We see that taking the best result that returns before 1 second results in a higher latency-fidelity metric than using the highest fidelity result.

Table 5.7: Achieving Latency Constraints for Panlite

Table 5.7 shows the results for this experiment. We see that by taking the best result after 1 second and returning that to the application, Chroma is able to achieve a higher latency-fidelity metric than by waiting for all the engines to finish and returning a full fidelity result. During this experiment, Chroma did the following: It performed the translation using a different translation engine (`ebmt, gloss, dict`) on each of the three servers. When the latency constraint expired, Chroma determined which engines had successfully finished translating. Chroma then consulted the tactics description to determine how best to combine the completed results to provide the highest fidelity output. All of these steps can be done automatically by Chroma without application knowledge.

5.5.3 Reducing Latency by Decomposition

5.5.3.1 Description

This experiment shows how decomposing an operation into smaller pieces and executing each piece on a separate remote server reduces the overall latency of the operation. The Vivendi syntax to describe data decomposition is presented in Section 2.5.2.2. To facilitate this decomposition, for each application validated in this section, I created functions to split and recombine the application data.

5.5.3.2 Results

As shown in Figure 5.3, Face had high latencies for the three input files. However, this latency can be reduced in two ways. Firstly, the input image can be reduced in size by scaling it. However, this method reduces the fidelity of the result. The second method is to break the image into smaller pieces and separately process each piece. This method has the potential of improving the latency without reducing the fidelity.

Table 5.8 shows the results obtained by using this method. I ran each experiment 5 times and measured the average latency and standard deviation. The servers used were unloaded. The results show that splitting the image into smaller pieces (allowing Chroma to parallelize

No. of Servers Used	Average (s)	Standard Deviation (s)	Latency Reduction
1	24.54	0.05	—
2	13.59	0.05	44.6%
3	9.73	0.07	60.4%

We see that splitting the input image for the operation into smaller pieces and sending these smaller pieces to different remote servers results in a dramatic reduction in total latency. The number of servers used corresponds to the number of pieces the image file was split into.

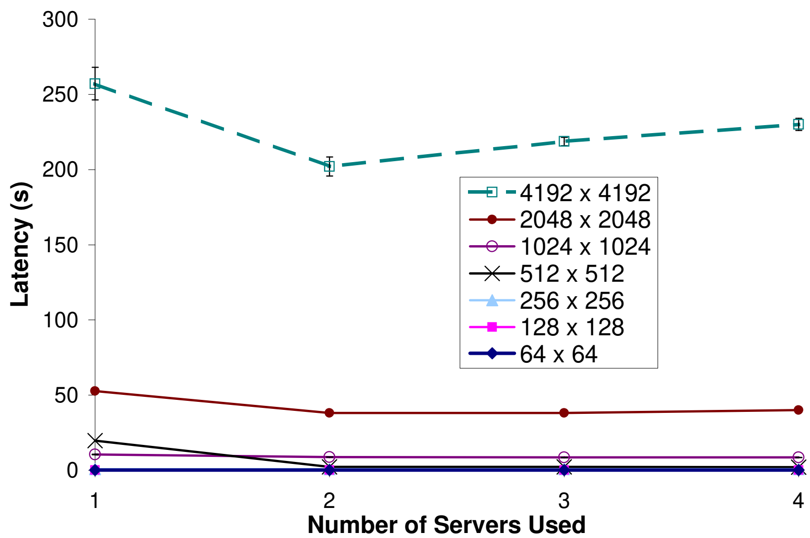
Table 5.8: Improvement in Face Latency by Decomposition

the operation) results in a substantial latency improvement (up to 60% reduction) over the original latency.

Figure 5.9 shows the benefit of data decomposition for the c2dfft application (Section 3.3). The two FFT stages of c2dfft can benefit from data decomposition by performing parallel FFTs of subsets of the original input matrix. The figure shows the benefits of this decomposition for different input matrix sizes. The benefits are marginal – a 20% improvement for the largest matrix when using two servers instead of one. Using more than two servers does not improve performance – in fact it hurts performance in many cases. The reason for this marginal improvement is that even though the two FFT stages do benefit tremendously from data decomposition, the latency of the transpose stages remains the same no matter how many servers are used. Additionally, a large amount of the total time is spent transferring the huge matrixes to the remote servers (a 4192 by 4192 matrix has about 500 MB of data). To fully realize the benefits of data decomposition for c2dfft, Chroma will have to transfer control to a remote server to minimize the amount of data that needs to be transferred. These results indicate that Chroma must take into account the amount of data transfered before deciding on the optimal number of servers to use for data decomposition. Currently, this smart logic has not been implemented into Chroma and is left for future work.

The PopUp application (Section 3.10) can also benefit from data decomposition. PopUp processes a number of independent image files and extra servers can be used to parallelize this process. Table 5.9 shows the benefits of doing this – up to a 65% reduction in latency compared to using just one remote server.

A more comprehensive evaluation of the benefits of data decomposition (using Chroma, Panlite, Janus, and Flite (Section 3.2)) is available in Cheng's [45] Master's dissertation. Cheng also evaluates various client-side algorithms for effectively sharing extra resources between multiple independent clients.



The x-axis is the number of servers that were available for data decomposition. The y-axis is the total application latency (in seconds). This is the time to complete 2 FFTs and 2 transposes of the input matrix. The input ranged from a 64 by 64 matrix to a 4192 by 4192 matrix (each input is shown as different lines).

Figure 5.9: Performance of c2dfft when using data decomposition

5.5.4 Q4: Summary

In this section, I presented three ways in which Chroma can use tactics to automatically improve user experience in over-provisioned environments. The improvement in each case was significant. Tactics allow us to obtain these improvements automatically at runtime without the application being aware of Chroma's decisions. The "data decomposition" method (Section 5.5.3), was the only method that required prior input from the application before it could be used. In this case, the application needed to tell Chroma how its data could be split into smaller pieces and recombined later. But even here, once Chroma had this information, it was able to use extra available resources to improve application performance at runtime without the application being aware of Chroma's optimizations.

In practice, the "fastest result" (Section 5.5.1) and "best fidelity" (Section 5.5.2) methods prove to be minor optimizations as they require very specific conditions, such as loaded servers for "fastest result" and applications with multiple tactics and/or fidelities for "best fidelity", before they become useful. However, data decomposition proves to be more useful, especially in applications such as image processing and image identification where the input data can be easily decomposed into multiple independent chunks. There were many examples of these types of applications in the computationally-intensive interactive domain. Hence, data decomposition was integrated into Vivendi (shown in Section 2.5.2.2) and Chroma (shown in Appendix A.6.1 and A.8.2).

No. of Servers Used	Average (s)	Standard Deviation (s)	Latency Reduction
1	1025.71	11.62	—
2	706.59	6.57	31.1%
3	354.67	4.00	65.4%

We see that using extra servers can greatly improve the PopUp latency. PopUp was given 3 image files to process. It could benefit from data decomposition by sending different image files to different servers. The partial VRML files were then recombined on the client. This experiment used different servers from the previous experiments as each server needed to run Matlab. In particular, three Pentium 3 700 Megahertz machines with 256 Megabytes of RAM were used as servers.

Table 5.9: Improvement in PopUp by Decomposition

5.6 Q5: Resistant Against Malicious Servers

In this section, I answer the last question posed in Section 5.1 and verify if Chroma can use simple client-side mechanisms to protect against malicious servers. These are servers that arbitrarily return wrong results to clients. Previous solutions to this problems used trusted hardware components, such as TPMs [184, 185], to detect malicious code. Cerium [44] is an example of such a system. Recently, there have also been systems, such as Pioneer [193], that are able to perform excellent software attestation (detection of malicious code) without the use of trusted hardware components.

For this dissertation, I used a simple verification mechanism where the mobile client verifies the correctness of a result, using MD5 checksums, returned from an untrusted server with a known trusted server. It is probabilistic in nature as the client does not do this verification all the time (otherwise this degenerates into using a trusted server all the time). The frequency of verification affects the probability with which the mobile client can discover a rogue server. The advantage of the scheme is that the mobile client does not have to always use the trusted server, which could be located fair away and/or be loaded. The mobile client could use unloaded nearby untrusted servers and probabilistically verify the results returned by those servers with the trusted server.

A key drawback of this mechanism is that it is fairly fragile as it cannot easily tolerate applications that are not idempotent. In particular, Pangloss-Lite cannot easily use this verification as the same input sentence can result in different non-deterministic output translations. One possible solution is for the trusted server to test the untrusted output with many possible correct output permutations. However, this process requires additional time and could result in incorrect results being classified as correct. In this section, the goal is to demonstrate, when using deterministic applications, the effectiveness of simple, easy to implement schemes in protecting against malicious servers.

Note that this mechanism does not protect the privacy of the client. A rogue server could easily leak or infer information about the client. In general, preserving privacy when performing RPCs is hard as the untrusted server is merely provided the inputs to the functions that need to be computed. If these inputs are encrypted or modified in any way (to preserve privacy), the server may not be able to compute a correct result. If privacy is important, a better solution would be to couple systems like Cerium or Pioneer, with code migration techniques (either at the application or virtual machine level) to perform the remote execution. This would allow the client to establish a completely trusted computing environment on a remote server and ensure that there is no possibility for a rogue process to see any client data.

This mechanism requires the mobile client to have access to at least one trusted server. However, no changes need to be made to untrusted servers. This mechanism also has the nice property that, depending on the implementation, the untrusted server will not be aware that the mobile client is checking its results. As such, it makes it much harder for the untrusted server to modify its behavior based on what the mobile client is doing.

This probabilistic mechanism is simple enough to actually be useful for a mobile client. The major drawback is that it requires access to a trusted server. However, I believe that this disadvantage is not enough to make this mechanism unusable in practice. The rest of this section will analyze the effectiveness of this mechanism. The different methods and frequency of checking results with the trusted server are described in Section 5.6.1 while Sections 5.6.2 and 5.6.3 evaluate the accuracy and resource usage of this probabilistic mechanism respectively.

5.6.1 Implementation of Probabilistic Verification Mechanism

The probabilistic verification mechanism described above can be implemented in a number of ways. Each of these methods has different strengths and weaknesses. I present three different methods below.

5.6.1.1 Verify Result Simultaneously

In this method, the mobile client contacts the trusted server at the same time as it sends the request for computation to the untrusted server. This has the advantage of minimizing the time needed to get a verified result from the trusted server. The mobile client then compares the result from the untrusted server with the result from the trusted server.

The main disadvantage of this method is that an untrusted server listening on the wireless link may discover that the mobile client is also talking to a remote trusted server. As such, the untrusted server may then decide to fool the client by not returning a wrong result this time. I call this method *Verify Simultaneously*.

5.6.1.2 Verify Result After

In this method, the mobile client contacts the trusted server after it sends the request for computation to the untrusted server. This method has the advantage in that it leaks no information to the untrusted server. By the time the untrusted server realizes that the mobile client is verifying its result, it is too late for the untrusted server to change its result.

The main disadvantage of this method is that it increases the latency of the computation. The mobile client first has to wait for the untrusted server to finish its computation and then it has to wait for the trusted server to finish its computation. I call this method *Verify After*.

5.6.1.3 Verify Result with Cached Known Results

In this method, the mobile client contacts the trusted server ahead of time and caches the results of some random number of known computations. Every now and then, the mobile client sends one of these known computations to an untrusted server and verifies the result against its cached result. This method has the advantage that the mobile client doesn't need to have contact with the trusted server. The untrusted server also never sees messages from the mobile client to the trusted server. Hence, the untrusted server cannot tell if the mobile client is checking its results.

The main disadvantage of this method is that the mobile client needs extra disk space to store the results of the known computations. This may not be viable for certain mobile clients. The cached results may also become stale. Finally, certain computations require runtime components, such as the system clock, and cannot be cached in advance. I call this method *Cache*.

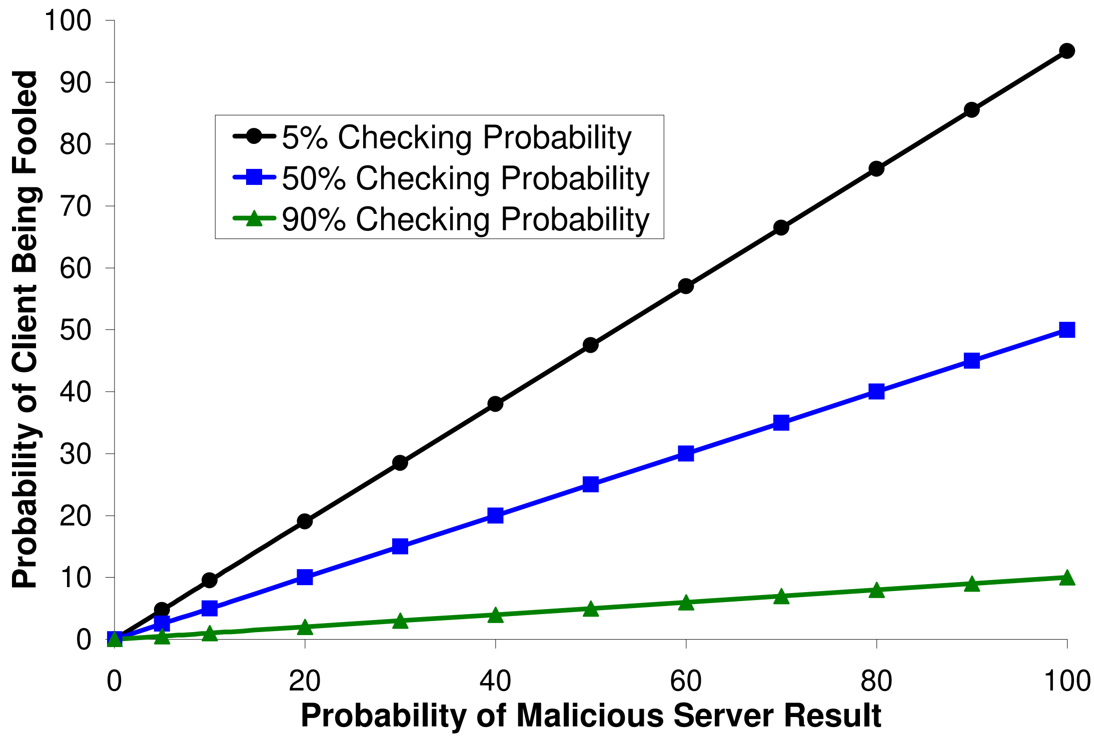
5.6.2 Accuracy of Scheme

In this section, I determine how effective the probabilistic mechanism is at detecting malicious servers that purposely return wrong results. The probability of a remote server returning wrong results and the probability of a client checking a result are independent as the actions of remote servers and mobile clients are very loosely coupled in real environments. As such, the probability of a mobile client detecting that a remote server is malicious is simply $x * y$, where x is the probability of the remote server returning a wrong result and y is the probability of the client checking the result.

Figure 5.10 provides a graphical representation of this simple equation, for easy viewing, for different client checking probabilities. As expected from the linear equation, the probability of a mobile client accepting a wrong result from a bad surrogate increases as the probability of the remote server lying increases. In such cases, the mobile client needs to increase the probability with which it verifies results. However, increasing this probability has resource and latency implications as shown in Section 5.6.3.

5.6.3 Resource Usage of Various Methods

In this section, I determine the resource usage of the three different methods (as described in Section 5.6.1) of the probabilistic verification mechanism. The bandwidth, battery, disk and latency impact of each of these three methods is evaluated. The bandwidth, battery, and disk



This figure shows the probability that a mobile client accepts a wrong result from a remote server. The x-axis is the probability of the remote server returning a wrong result while the y-axis is the probability that the mobile client actually accepts the wrong result (these are the results that the mobile client thinks is good because they were not checked). Each line is a different client checking probability. For example, 50% means that the client verifies a result 50% of the time.

Figure 5.10: Probability of a Client Accepting a Wrong Result

usage of each method was determined using simple "back of the envelope" calculations. The latency impact was determined experimentally.

5.6.3.1 Bandwidth / Battery Usage

For most mobile clients, the main source of energy consumption is the transmission and reception of network packets. Hence, any increase in bandwidth usage will also lead to an increase in energy usage.

Analytically, the bandwidth usage of each method is shown in Table 5.10. Both *Verify After* and *Verify Simultaneously* require twice the normal bandwidth needed for as computation as they also have to do the same computation at the trusted server. *Cache* requires no extra bandwidth and is thus the best method if reducing bandwidth usage becomes a significant concern.

Verify Simultaneously	2 * Bandwidth Needed for Computation
Verify After	2 * Bandwidth Needed for Computation
Cache Results	Bandwidth Needed for Computation

Table 5.10: Bandwidth Needed by Each Implementation

Verify Simultaneously	None
Verify After	None
Cache Results	Disk Space Needed to Store Previously Computed Results

Table 5.11: Disk Space Needed by Each Implementation

5.6.3.2 Disk Usage

The disk usage of each method is shown in Table 5.11. Both *Verify After* and *Verify Simultaneously* require no additional disk space.

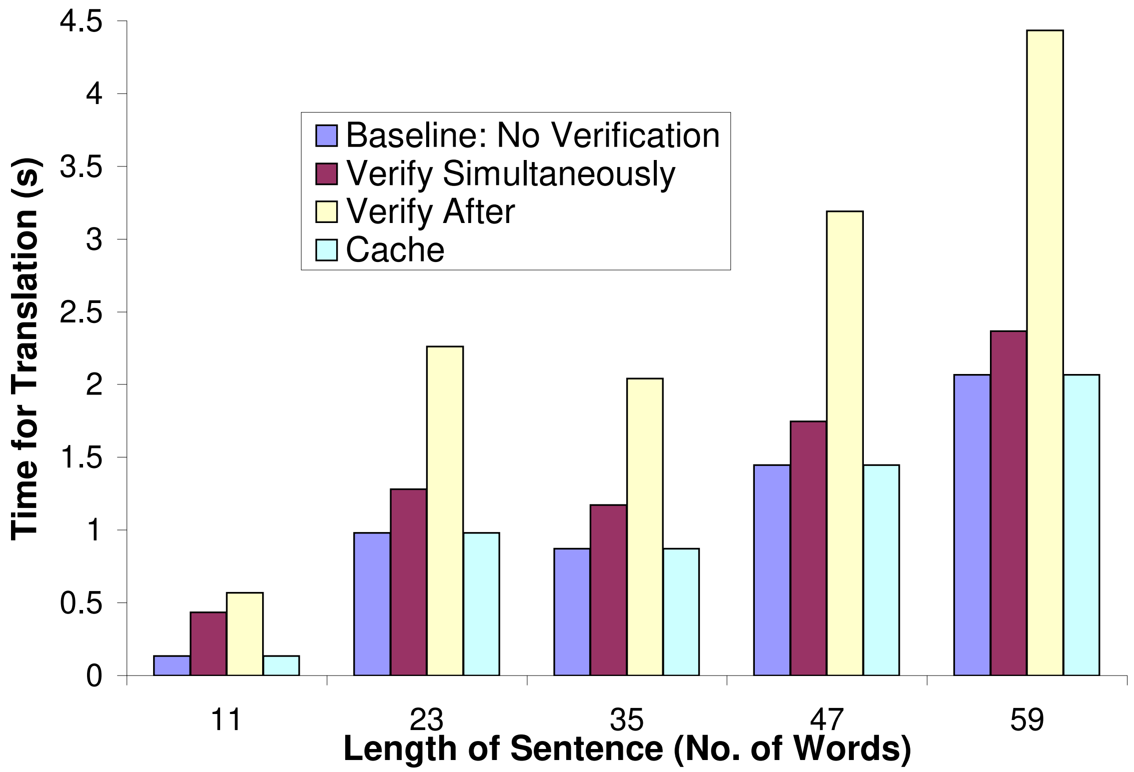
However, *Cache* requires disk space proportional to the number of results that are being cached. This can range from just a few bytes of space (if only 1 or 2 results are cached) to a few gigabytes (if hundreds of results are cached). The amount of disk space needed is also highly application dependent. For example, the highly compressible text output of Panlite requires far less space than the large binary output of Flite (audio file containing the spoken text) or GLVU (bitmap of the screen). This disk space requirement is a main reason why *Cache* may not be suitable for all mobile clients; especially those with limited storage capacity.

5.6.3.3 Latency Impact Analysis and Experimental Results

Analytically, the latency impact of each method is shown in Table 5.12. *Verify After* has the largest expected latency as it has to perform two computations sequentially. *Verify Simultaneously* will have the latency of either the remote server computation or the trusted server computation; whichever is larger. This usually will be the trusted server computation latency as the trusted server is usually located further away than the remote server. *Cache* has no extra latency (other than the miniscule amount of time needed to verify a result

Verify Simultaneously	max(Latency of Computation at Surrogate + Latency of Computation at Trusted Server)
Verify After	Latency of Computation at Surrogate + Latency of Computation at Trusted Server
Cache Results	Latency of Computation at Surrogate

Table 5.12: Latency Impact of Each Implementation



This figure shows the latencies of each of the three methods. The latency of the computation performed without verification is shown as a baseline.

Figure 5.11: Experimentally Determined Latencies of Each Implementation

against a cached result).

To verify the latency impact experimentally, each of the three verification methods was implemented in Chroma. I used the hardware setup described in Section 5.1.1. In particular, I used the slow client as the mobile client and had a single trusted server and a single untrusted remote server. I assumed that the trusted server was located far from the mobile client (otherwise, the client could just use the trusted server all the time). Hence, the round trip latency to the trusted server was set to 300ms, using NistNet [159], and the round trip latency to the untrusted surrogate was set to 10ms. I used Panlite as the test application and each experiment was repeated 5 times. This experiment presents the worst case latency as each result returned by the remote server was always verified. In practice, the average latency will be lower as not all results will be verified.

Figure 5.11 shows the experimentally measured latencies for each method. They agree with the analytical claims and show that the *Verify After* method had the worst latency while the *Cache* method had the best latency.

5.6.4 Choosing the Appropriate Method

After factoring in the pros and cons of each method, *Cache* appears to be the best. It uses the least amount of bandwidth and has the lowest latency. However, it requires disk space which could be a problem for some handheld devices. *Cache* also requires pre-planning as mobile clients will have to cache trusted results for each application in advance. However, its other benefits outweigh these disadvantages. As such, *Cache* should be used except when disk space is at a premium. In such cases, or when a trusted result has not been cached, *Verify Simultaneously* should be used as it has the next lowest latency. *Verify After* should only be used when mobile clients cannot use *Cache* and they suspect that remote servers are actively monitoring the network.

5.6.5 Q5: Summary

In this section, I showed how a simple probabilistic verification mechanism can be used to provide protection against malicious remote servers returning incorrect results. This mechanism can be implemented in various ways; each with its own pros and cons. A comparative analysis of three such ways, *Verify After*, *Verify Simultaneously*, and *Cache*, suggests that *Cache* has the best overall performance.

5.7 Q6: Chroma's Performance in More Realistic Environments

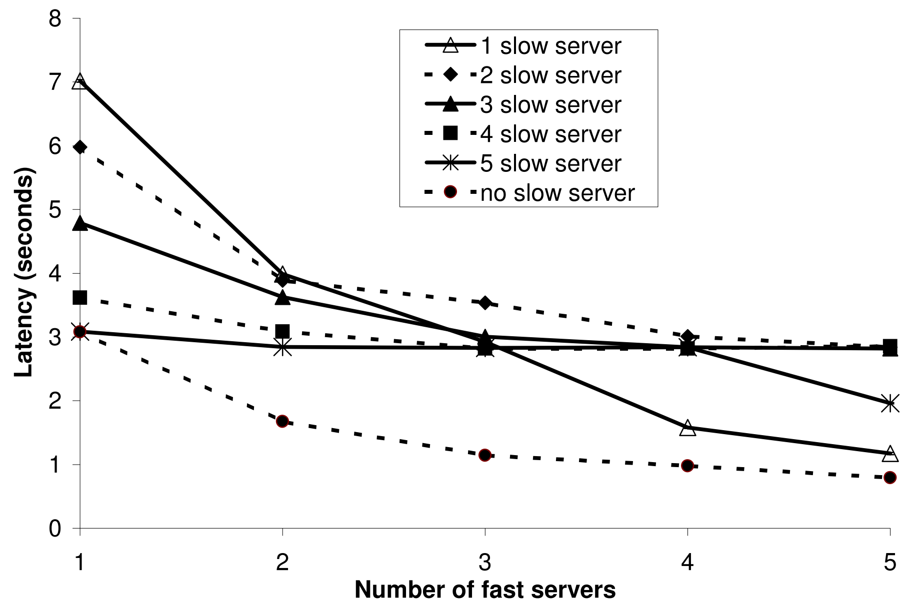
In this section, I investigate how effective Chroma is when operating in more realistic environments. In Section 5.7.1, I present results showing how Chroma behaves in environments that have heterogeneous servers while in Section 5.7.2, I show how multiple Chroma clients can effectively share the same set of server resources.

All the results in this section are taken from the thesis of a Masters student, Jesse Cheng, that I supervised. The detailed results, experimental setup, and discussion appear in his thesis [45]. I merely present a summary of his results here for completeness.

5.7.1 Heterogeneous Environments

The previous validations presented in this chapter have assumed that the servers available to a Chroma client have been homogeneous. In particular, the servers have had the same CPU capability and available memory. In this section, I present results to show how Chroma would behave in more realistic environments that have heterogeneous servers.

For this experiment, five slow servers and five fast servers were used. The slow and fast servers were the same as the slow and fast clients described in Section 5.1.1. The slow client was used for all the experiments. GOCR was used as the application for these experiments and it was provided an input containing 1524 characters. For these experiments, we used data decomposition and provided each server with a different part of the input to process.



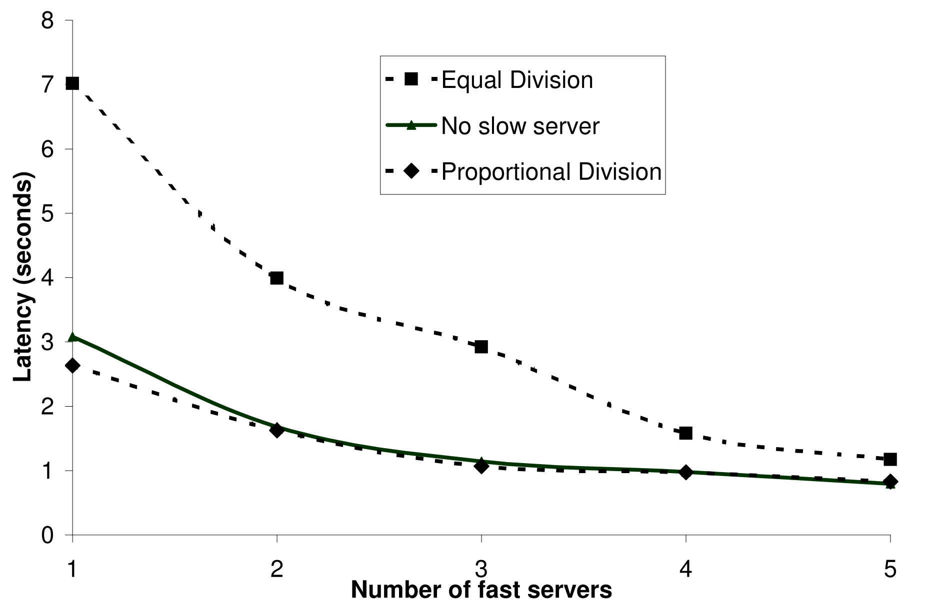
The lines are in the same order as the legend. This figure is reproduced from Cheng's thesis [45].

Figure 5.12: Naive Use of Slower Servers

Figure 5.12 shows the performance when each server (regardless of whether it was fast or slow) was provided the same size input to process. We see that, in general, the overall latency decreases we increase the number of servers in the system. However, it is clear that the overall performance of the system is bottlenecked by the slow servers. For example, when we have 5 fast servers and 5 slow servers in the environment, the average latency is around 3 seconds. But with 5 fast servers alone (the bottom line in the Figure), the latency is less than one second on average. This phenomena arises because the input was proportionally divided between the server without regard to the actual computationally capabilities of the servers. Hence, the slow and fast servers had to do the same amount of work and this resulted in the entire system being slowed down by the slow servers.

Figure 5.13 shows the performance when each server receives inputs that are sized according to the computational capabilities of the server. Hence, a server that is 3x faster will receive and input that is 3x larger. We see that this proportional division of input has a dramatic impact on performance. In particular, using a slow server now has a small improvement over not using any slow servers at all (when the number of fast servers used is small). More importantly, using slow servers doesn't hurt performance compared to using just fast servers.

Hence, in summary, when using heterogeneous servers, Chroma needs to divide its work proportional between the servers (based on their computational capability). Otherwise, the slower servers will limit the performance that Chroma can achieve even when a



One slow server was used for the "Equal Division" and "Proportional division" results. Only fast servers were used for the "No slow server" result. This figure is reproduced from Cheng's thesis [45].

Figure 5.13: Proportional Use of Slower Servers

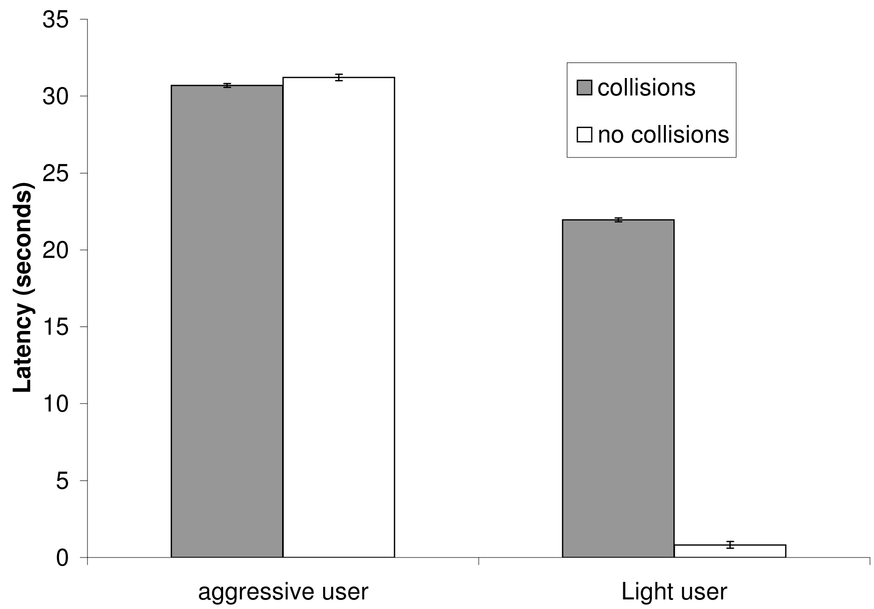
large number of servers are used.

5.7.2 Sharing Server Resources Between Independent Chroma Clients

In the previous validations presented in this chapter, with the exception Section 5.5.1, I assumed that the Chroma client had sole and unrestricted use of all server resources. However, this assumption is not realistic. In practice, multiple Chroma clients could be vying for the same set of server resources. For example, there may be a few Chroma clients at a cafe all trying to use the same set of servers provided by the cafe.

To investigate the effect of competition between Chroma clients, a simple experiment was conducted using two slow Chroma clients. We chose GOCR as the application for this experiment. One client was designated as the "aggressive user" and it continuously ran a large optical recognition involving 9144 characters. This operation takes 120 seconds to complete on a single fast server. The other client was designated as the "light user" and it ran a small optical recognition involving 768 characters every 10 seconds. This operation takes 1.36 second on average to complete on a single fast server.

We provided eight fast servers in the environment and allowed each client to use two-way data decomposition (i.e., they could split the input into two pieces and use two servers simultaneously). We allowed each client to pick any server it wanted, using the normal



This figure is reproduced from Cheng's thesis [45].

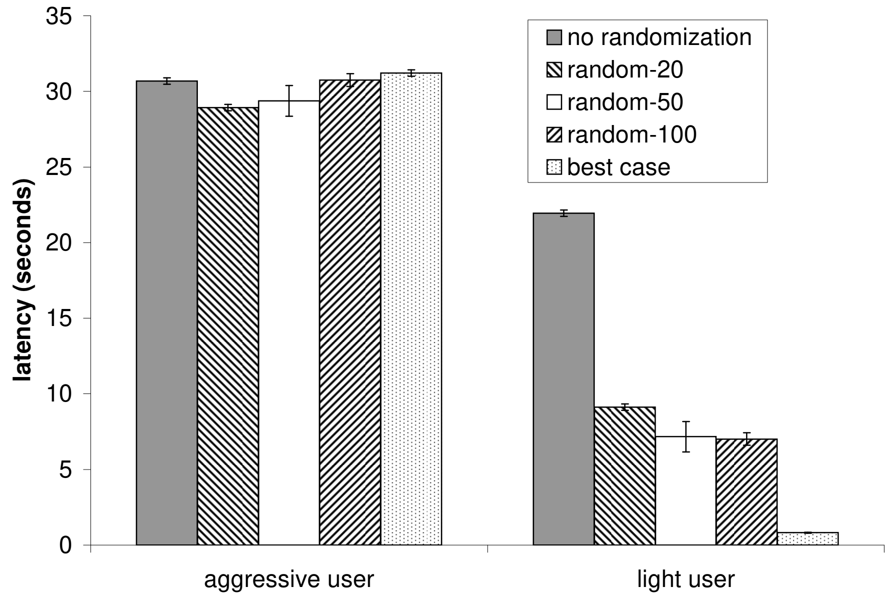
Figure 5.14: Effect of Server Collisions Between Competing Chroma Clients

Chroma server selection routines (that pick the first available server that is suitable), and ran the experiment until the light client had finished 100 recognitions of its 768 character input.

Figure 5.14 shows the effect of server competition (the bars labelled "collision"). Since each client picks servers independently (and currently will pick the first available server they find), the light and aggressive clients frequently chose the same servers. Note that they only pick the same server if the server was unloaded at the time they both were deciding which server to picked. If the server was already being used, Chroma would have picked another server.

The aggressive client greatly benefited from data decomposition as its latency decreased for 120 seconds to about 30 seconds. However, the latency for the light client increased by more than an order of magnitude and went from 1.36 seconds to about 23 seconds. The figure also shows the latency (the bars labelled "no collisions") that would have been obtained if there was some global coordinator, or server-side strict admission control mechanism, that ensured that the two clients used different servers (which is possible as the two clients, together, need only 4 of the 8 available servers). In the best case, the aggressive client's latency would still have been about 30 seconds. However, the light client's latency would improve tremendously to about 0.82 seconds.

Unfortunately, a global coordinator, that can keep track of independent clients and assign them to distinct servers, is not likely to be found in many environments. Hence, what



"Random 20", "Random 50", and "Random 100" are 20, 50, and 100 iterations of the light client respectively. The aggressive client had similar latencies for all the experiments (the apparent variations were due to random fluctuations and were not meaningful). This figure is reproduced from Cheng's thesis [45].

Figure 5.15: Benefits of Randomized Server Selection in Reducing Collisions

measures can individual clients take to improve their performance, even in the presence of competition from other independent clients?

Figure 5.15 shows the effect that a simple randomized server selection scheme has on client performance. In this scheme, each client randomly picks servers from the list of available, and equally capable, servers. I.e., the client first generates a list of servers that have sufficient resources to handle the client's request. The client then randomly picks servers from this list to use. This is different from the current algorithm that always picks the first available server from the list of usable servers.

We see that the longer the client uses the randomized scheme, the better its performance gets. This is not a surprising results as, in the long run, the probability that the client uses a server that is completely free is higher. Unfortunately, even after 100 iterations, the randomized performance is only about 7 seconds. This is still quite far from the best achievable latency of 0.82 seconds. However, it is much better than the latency of 23 seconds reported in Figure 5.14. Developing better algorithms that will allow Chroma clients to compete fairly without requiring independent Chroma clients to communicate with each other or requiring a global coordinator or strict server-side admission control is an ongoing research area.

5.7.3 Q6 Summary

In this section, I showed how Chroma could operate in more realistic environments where server capabilities could be heterogeneous and where multiple clients could be competing for the same server resources. I showed that to make full use of heterogeneous servers, Chroma needs to send each server data proportional to the capabilities of the server. I also showed that a simple randomization algorithm can reduce the effect of competition on clients. However, the randomized scheme's performance is still significantly worse than the best possible performance and more research is needed to develop better algorithms to allow independent clients to share servers.

5.8 Summary

In this chapter, I provided validation that applications can use Chroma to effectively run on resource limited mobile devices under various mobile computing scenarios. This validation was presented in five parts: first I showed that Chroma can choose the optimal runtime setting for an application when the resource environment and user preferences are fixed. I then showed that Chroma can also choose the optimal runtime setting even when the resource environment and user preferences change. Next, I showed that Chroma has reasonable overheads as long as the user is willing to use cached resource measurements. I then showed that Chroma can successfully use extra server resources in the environment to dramatically improve application performance. I then provided evidence that Chroma can use a simple probabilistic verification mechanism to protect users from malicious remote servers. Finally, I presented results that showed that Chroma can work in real environments that have competing Chroma clients and heterogeneous servers. These six smaller validations provide substantial evidence of the viability of Chroma to effectively support the dynamic runtime needs of mobile users. This validates that applications retargeted using RapidRe can achieve excellent performance in cyber foraging environments.

In this chapter, I provided detailed results showing the Chroma could pick the correct runtime setting for Panlite (Sections 5.2.2.2, 5.3, and 5.5), Janus (Sections 5.2.3.2 and 5.5), Face (Sections 5.2.4.2 and 5.5), c2dfft (Section 5.5), and Popup (Section 5.5). In Section 4.6, I showed that GLVU, Flite, Music, and Radiator could also be used with Chroma under different conditions. Finally, Cheng's dissertation [45] shows that GOCR (the final application) can also be successfully adapted by Chroma. It also has additional results for Panlite, and Flite. Overall, these various validations prove that all ten applications described in Chapter 3 can be effectively adapted by Chroma.

In Chapter 4, I showed that eight of the ten applications (Chapter 3) could be quickly and effectively retargeted by developers. Seven of the applications (Face, Flite, GLVU, Janus, Music, Panlite, and Radiator) were retargeted by participants. The last application (GOCR) was used as a training application and was retargeted completely as part of the guided training process. Even though GOCR was not modified completely by a participant, I am confident that it would not have needed more time (or resulted in worse performance)

Applications	Good Runtime Performance?	Easy to Retarget?
Face (Face Recognizer)	Sections 5.2.4.2 and 5.5	Yes
Flite (Text to Speech)	Section 4.6 and in [45]	Yes
c2dfft (Image Filtering)	Section 5.5	Untested
GLVU (3D Visualizer)	Section 4.6	Yes
GOCR (Char. Recognizer)	In [45]	Yes (training app)
Janus (Speech Recognizer)	Sections 5.2.3.2 and 5.5	Yes
Music (Music Finder)	Section 4.6	Yes
Panlite (Lang. Translator)	Sections 5.2.2.2, 5.3, 5.5, and in [45]	Yes
Radiator (3D Lighting)	Section 4.6	Yes
PopUp (3D Scene Generation)	Section 5.5	Untested

For each of the ten applications, the second column (Excellent Runtime Performance?) lists the section (or related work) where the validation for the application, that shows that it can achieve excellent cyber foraging performance is presented. The third column (Easy to Retarget?) lists if the application was used in the user study presented in Chapter 4.

Table 5.13: Overview of the Versatility of Chroma and RapidRe

than the other seven applications. The user study provides strong validation that RapidRe is capable of rapidly retargeting a broad range of useful mobile applications.

I currently have not validated whether c2dfft and PopUp can be retargeted by developers. Both these applications require developers to do a little more work. In particular, developers have to write the split and join functions to perform the data decomposition steps (Section 2.5.2.2) for these applications. However, I believe that these additional steps can be done in a reasonable amount of time and that retargeting these two applications will not take more than 4 to 5 hours. Table 5.13 summarizes these results.

In Section 2.3.3, a list of requirements for RapidRe were stated. Now that the validation section for this dissertation have been presented, I can determine how well I supported those requirements. The requirements, along with how well my dissertation supports them, are:

- Any application that fits the requirements listed in Section 2.2.3. Chapter 3 demonstrates this requirement.
- Any language (C, C++, Ada, Java, Tcl, etc.). Chapter 3 demonstrates this requirement.
- Any developer (novice or expert). The user study presented in Chapter 4 was conducted using novice developers. I am confident that the results apply to expert developers as well.

- Allow quick retargeting of applications. As Section 4.4 shows, the retargeting time has decreased from weeks to just a few hours.

- Retargeted application must be effective (in terms of absolute runtime performance). Chapter 5 shows that Chroma is highly effective in achieving excellent performance for applications running on mobile devices. However, not all the retargeted applications (Section 4.6) achieved the maximum performance possible. This was due to errors made by the novices when retargeting the applications. I am confident that some procedural changes in the retargeting process, explained in Section 4.7.2, will reduce these retargeting errors significantly.

Overall, in summary, this chapter, along with Chapters 3 and 4, collectively provide good validation of this dissertation's thesis.

Chapter 6

Related Work

In this chapter, I present the related work relevant to this dissertation. My work spans mobile computing, software engineering and human-computer interaction (HCI). At their juncture lies the problem of rapid modification of resource-intensive applications for effective cyber foraging. To the best of my knowledge, this is the first work to recognize the importance of this problem and propose a solution. My solution and its validation build upon work in three areas: declarative descriptions of application behavior using little languages, adaptive runtime systems for mobile computing, and a complete software engineering process evaluated using rigorous HCI evaluation methods. I am not the first to research any of these individual areas. However, I am the first to combine these three areas into a complete solution for an important problem.

Section 6.1 presents the work most closely related to the Vivendi little language. I show that even though little languages have been used in many different scenarios before, my use of languages to support mobile applications is fairly new.

In Section 6.2, I present the work relevant to Chroma. Chroma is similar to and even draws inspiration and ideas from a number of previous adaptive systems. However, my use of tactics to reduce the solution search space is novel.

Finally, in Section 6.3, I present the work relevant to RapidRe. In particular, I discuss previous research in facilitating rapid retargeting of applications for a particular domain. I also describe prior research that evaluated the usability of software systems and explain how I used those ideas to develop the user study described in Chapter 4.

6.1 Vivendi: Related Work

The declarative language I use to express an application's tactics addresses some of the same issues as 4GLs [143] and "Little languages" [25]. The latter are task-specific languages that allow developers to express higher level semantics without worrying about low level details.

The power of little languages was first shown by early versions of the Unix programming environment. *Make* [71] is perhaps the best-known example of a little language. As

Bentley explains [25], the power of a little language comes from the fact that its abstractional power is closely matched to a task domain. My Vivendi language exploits this power as it allows application developers to specify the adaptive capabilities of their applications at a higher level without needing to worry about low level system integration details.

There have been numerous other systems that used a little language to express application capabilities. I provide just a small sampling here. In the area of software compilation tools, in addition to Make, the GNU configuration tools, comprising of Autoconf [88], Automake [89], M4 [91], and libtool [90], provide a language for developers to state the requirements (software tools, source files, dependencies, etc.) for compiling their software. These tools then read these descriptions and generate the appropriate Makefiles needed to compile the software. The Ant build tool [13] uses an XML syntax (instead of Makefiles) to describe the software's build dependencies. Finally, both the Debian [56] and RedHat [176] Linux distributions provide customized little languages for creating software packages. Debian uses the dpkg [57] language and toolset while RedHat uses the RPM [18] language and toolset. Vivendi differs from all these languages in that it is focused towards expressing the adaptive capabilities of useful mobile applications.

There have also been several little languages created to assist the development of applications that have specific Quality-of-Service (QoS) requirements. Leue describes the use of formal specification languages, specifically Specification and Description Language (SDL), Message Sequence Charts (MSCs), and temporal logic, to specify QoS requirements for different environments [139]. The QUASAR project used the Z specification language to specify various application-level QoS requirements [200]. These were then translated into specific resource reservation requests.

Frolund and Koistinen describe a specification language, QoS Modeling Language (QML), for specifying QoS requirements [82]. QML is an extension of the Unified Modeling Language (UML) [163] and allows developers to specify the QoS requirements of applications along multiple dimensions. BeeHive provided a set of service-specific application programming interfaces (APIs), through which objects can make QoS requests from an underlying resource manager [201]. The resource manager translates each request into low-level resource requests. Monteiro et al. describe a language (and mechanisms to enforce the language specifications) for specifying QoS requirements for communication networks [147].

Finally (in the area of QoS specification languages), the Condor scripting language [51] allows developers to specify how, to maximize application throughput, existing applications can be distributed across a computing cluster. These languages all assume a mostly static environment where resources do not fluctuate dramatically due to environmental factors. Resource availability changes only because of competition from other applications and processes.

The languages most closely related to Vivendi are those that specifically assist the development of mobile computing applications. The best example of this are the Quality Description Languages (QDL) [142] developed by the QuO project. The QuO framework uses object-based remote execution to create distributed CORBA [162, 221] applications that have specific QoS requirements. QDL comprises of two main languages: a Contract Description Language (CDL) and a Structure Description Language (SDL). These languages use aspect-oriented language principles first proposed by Kiczales et al. [132]. CDL is used to describe the QoS requirements of a client while SDL is used to describe the structure of a QuO application. This structure includes specifying the mechanisms that an application should use when performing remote object execution. Pal et al. [167] extended this work and developed the Configuration Setup Language (CSL). CSL allows developers to specifying exactly how QuO applications should be initialized and shut down. This makes it possible to dynamically start server components on remote servers without any manual setup steps. Vivendi differs from QDL as it doesn't assume that applications will use the CORBA Object Request Broker (ORB) model of remote execution. Vivendi's goal is to support a far wider range of applications than QDL. The tradeoff is that, by constraining the execution environment, QDL allows developers to specify far more precise QoS and runtime settings.

Finally, looking a little broader, there have been numerous programming languages that were built to support the development of distributed and parallel applications. These languages include Durra [21, 22], Emerald [124], Facile [85], Jade [180], Java RMI [204], Mentat [95], MPI [149, 215], NESL [35, 34], Obliq [39, 40], Orca [19], PVM [50, 84, 205], and TDL [197]. Using these languages, programmers can explicitly specify how applications (or processes) should be parallelized and partitioned either across a multi-CPU or cluster computing environment. To obtain the benefits of these languages, applications had to be programmed using the language. The mobile computing domain doesn't allow this. A large number of useful applications already exist, in a number of different languages, and reprogramming them using a specific language is highly impractical. Additionally, with the exception of Java RMI, none of these languages were designed for mobile environments.

6.2 Chroma: Related Work

Building an adaptive system is definitely not new. There is a long and rich history of research in building adaptive systems. In the rest of this section, I present the systems that are most related to Chroma. I do not describe the work related to the service discovery and security components of Chroma as that work is discussed in Appendix A.7.1.

Broadly speaking, there are two types of adaptive systems: ones that perform purely local adaptations, such as the original Odyssey system [160], and ones, such as Chroma, that also use application partitioning and remote execution as part of the adaptation. In this section, I consider mainly the systems that use remote execution.

Within those systems, there are two broad methods of adaptation. The first type is application-transparent adaptation where the adaptive system uses well-defined APIs to adapt the application without needing any application modifications. The Coign [115] and Puppeteer [54] systems use this method of adaptation. Both of them adapt applications by changing the inputs of the application using the well defined Windows COM interface. Application-transparent adaptation makes it easy to add new applications into the system.

However, the adaptation is limited by the scope of the well-defined APIs. For example, it is impossible to change fidelity settings within the application that are not exposed via the well-defined API. Other examples of application-transparent adaptation are the TCP congestion control protocol [116] which transparently adapts the network usage of applications using the standard socket interface and the Coda filesystem [133, 190] which provides a standard filesystem interface, but hides network disconnections transparently from applications. Application transparency can also be achieved through the use of proxies as demonstrated by the Active Streams research [38], the Conductor system [237, 238], and Fox et al. [80].

The second is called application-aware adaptation and it requires applications to be modified to use the system. Chroma uses this method of adaptation. This results in better adaptation results at the expense of longer application modification times. I concentrate on application-aware adaptation in the rest of this section.

Chroma builds on ideas first proposed in Odyssey [160]. Odyssey demonstrated the importance of adapting to network bandwidth in mobile scenarios, and proposed the first generic API for application-aware adaptation. Narayanan extended this API to support multi-fidelity adaptation [151] where application quality can be changed at runtime (to conserve resources) by adjusting specific application variable settings. Chroma uses multi-fidelity adaptation and allows developers to specify the variables, using the fidelity variable aspect of Vivendi, that control the application's fidelity. Chroma also uses the history-based prediction mechanisms first proposed by Narayanan et al. [152]. These mechanisms have since been extended by Gurun et al [98]. Chroma builds on the remote execution framework developed by Flinn et al. [75]. It extends this framework to support the notion of tactics.

Systems such as Chroma and Odyssey were designed to support the adaptive requirements of individual mobile applications. There are also systems, such as Prism [199], that track the mobile user and perform adaptation at the user task level. A user task is a high level representation of a computing activity that the user wishes to perform. For example, one task may be "editing my thesis", while another may be "finish my programming assignment". Each task may require multiple applications working in sequence before it can be satisfied. For example, the "editing my thesis" task may require the use of an editor to create the Latex source and Make to create the final compiled document. Prism uses a combination of user input, provided via a GUI, and smart measurement and prediction mechanisms to accurately create task descriptions and utility functions that match the current user's needs. Prism also has mechanisms to discover the applications and resources that are available in the local environment. Prism then solves a resource optimization problem, using a component called the Environment Manager [173], to pick the appropriate applications and QoS levels that will satisfy all the user's tasks. It leaves the runtime optimization of these applications to lower level systems like Chroma.

Chroma and Prism complement each other very well and have been integrated with each other. Prism provides Chroma with utility functions that match the user's preferences. It also tells Chroma which applications need to be executed to satisfy a given user's task. Chroma, on the other hand, uses these utility functions to provide the best possible runtime performance for every chosen application. Prism is also able to query Chroma's fine-grained resource measurers and application resource predictors to obtain a better understanding of the current resource environment and application resource requirements. Appendix A.5 describes exactly how Prism interacts with Chroma.

Looking at related work from a broader perspective, there have been numerous other adaptive distributed systems. Rudenko et al. [182] and Flinn and Satyanarayanan [76, 77] have demonstrated that remote execution can dramatically reduce the energy consumption of mobile devices. Bayou [211, 172] and Rover [121] were application-aware solutions (unlike Coda) that allowed mobile users to access their data anywhere. In the domain of cluster and grid computing, Abacus [11], Condor [23, 213, 214], Globus [78], Legion [114, 154], OGSI [79, 122, 218], and OGSI.NET [225] have used remote execution to improve overall system throughput. There have also been grid computing systems, such as Mobile Legion [48] and Mobile OGSI.NET [46] that have developed client support for mobile devices. Finally, the Virtual Microscope project [7, 42, 73] uses remote execution and data adaptation to allow users to visualize high-fidelity medical images. These systems were either developed for very specific adaptation goals (energy, mobile data access, and data visualization) or were designed to work in mostly static resource environments (cluster and grid computing). Chroma, on the other hand, was designed to support multiple user-centric adaptation goals, such as accuracy, latency, and energy conservation, in a dynamic resource environment.

The adaptive systems most similar to Chroma are the various component-based adaptive QoS systems. These include CORBA-based adaptive middleware frameworks [63, 87, 136, 194], CORBA-based adaptive systems that use QuO [127], CORBA-based mobile application frameworks [102], and Java-based remote execution systems [27, 28]. These systems all provide functionality to measure the available resources in the environment and use these resource measurements to remotely partition applications according to a provided metric. Chroma differs from these systems in that it doesn't assume that applications are written using CORBA or Java.

Chroma's use of data decomposition is similar to methods first proposed in River [15]. These ideas were then expanded into a full production system by Google [55]. However, to the best of my knowledge, this is the first system to apply these ideas to the domain of mobile computing. In particular, as a viable mechanism for exploiting extra server resources in mobile environments.

Finally, Chroma's effectiveness depends on the correctness of the utility functions used by the solver. In this dissertation, I used Prism to obtain accurate utility functions that capture the user's preferences. There are other systems, such as Horvitz's mixed-initiative approach [109] and the Lumière Project [110], that can also be used. In addition, there has been a large amount of research in modeling user tasks, such as Albrecht et al. [10], Shearin and Lieberman [195], Terveen and Murray [212], and Weld [230], that can be used to determine the appropriate user preferences.

6.3 RapidRe: Related Work

RapidRe consists of a little language, a runtime system, and a stub generator. Individually, these components are not particularly novel. However, RapidRe combines them together to achieve the first solution for rapidly retargeting applications, written in a variety of languages, for cyber foraging. The related work for the little language and the runtime (Chroma) have been previously presented. In this section, I present the work related to a) the stub generator, b) the evaluation of RapidRe, and c) the problem of retargeting applications.

The use of a stub generator to greatly reduce the burden of developing large amounts of well-defined code is a well understood and used technique. For example, Birrell and Nelson [30] used a stub generator, called Lupine, to generate the client and server code stubs for their seminal RPC work. Their work inspired numerous other stub generators [86, 104, 170, 177, 219, 188, 203, 229]. RapidRe reuses these well-known and understood stub generation techniques.

To evaluate RapidRe, I used well-established techniques from HCI to conduct the user-centric evaluation. Nielsen [158] gives a good overview of these techniques. These techniques have traditionally not been used to investigate the effectiveness of programming tools. Recently, there has been some work in conducting user-centric evaluations of programming tools. Ko et al. conducted a study of Java programmers using Eclipse [66] to identify the tools most likely to help these programmers perform routine code maintenance. Klemmer et al. investigated the effectiveness of a toolkit at reducing the difficulty of developing tangible user interfaces. I used these evaluations as a reference when designing the user study presented in Chapter 4.

From a broader perspective, my work overlaps closest with automatic re-targeting systems such as IBM's WebSphere [37] and Microsoft's Visual Studio [235]. For example, IBM's WebSphere allows developers to retarget Java applications for use as mobile Web Portals while Visual Studio allows C# applications to be retargeted for mobile devices running the WinCE operating system. These systems allow developers to quickly port applications to new target systems. Unfortunately, they use a language-specific approach, which runs counter to my design considerations.

Looking further afield, RapidRe shares many similarities to approaches used to retarget programming language compilers for different hardware architectures [14, 43, 68]. RapidRe is also similar to various approaches used to retarget hardware specifications for new hardware devices [4, 36, 106, 108].

Chapter 7

Conclusion

For mobile computing to become a reality, we need small lightweight devices, pervasive wireless connectivity, and useful mobile applications. In the last decade we have seen rapid advances along the first two fronts. Devices are becoming smaller everyday and high speed wireless network connectivity will soon be available almost everywhere. However, it is still difficult to execute resource-intensive applications, such as language translators and speech recognizers, on these devices. Alas, these applications also tend to be of high value to mobile users.

This dissertation provided a solution to this problem. it presents a process called RapidRe, that makes it easy to retarget existing applications to work on mobile devices. RapidRe uses a little language called Vivendi and a custom adaptive runtime system called Chroma. Chroma uses remote servers and fidelity adaptation to allow resource-intensive applications to effectively run on small lightweight devices.

In this chapter, I summarize, in Section 7.1, the research contributions made by this dissertation. I then describe possible future work in Section 7.2. Finally, Section 7.3 places this dissertation in perspective with the current state of mobile computing research.

7.1 Contributions

This dissertation makes contributions in three major areas. The first area is conceptual—this consists of the novel ideas generated by this work. The second area is a set of artifacts: Chroma, the stub generator, and the other software systems that were developed to validate this dissertation. The final area of contribution is the experimental evaluation and the user study which validate the ideas presented in the dissertation.

7.1.1 Conceptual contributions

The primary conceptual contribution is the recognition of a deep architectural similarity in computationally-intensive interactive applications. In particular, a large number of these applications are already structured in a way that makes them very amenable to the retargeting process and remote execution and adaptation mechanisms developed in this dissertation. For example, applications are already developed such that operations are cleanly separated as distinct modules in the application. This makes it easy to identify the module that performs an operation and to remotely execute it. Additionally, any built-in application functionality for fidelity adaptation can be easily exploited. This thesis does not, however, provide any assistance for developers to add fidelity adaptation support to applications. This architectural similarity is thus a crucial factor that a) allows tactics to be easily discovered and specified, and b) enables RapidRe to be successful at dramatically reducing the time needed to retarget applications.

The second conceptual contribution is the recognition that for each application *operation*, the *parameters*, *fidelity variables* and *tactics* of that operation can be concisely described using a little language. Knowledge of these three attributes is sufficient to efficiently and effectively adapt the application in a mobile environment. This is an important contribution as it greatly reduces the application knowledge developers need to have before they can retarget an application for cyber foraging. It also allows the development of a customized little language, called *Vivendi*, that allows developers to quickly specify just these application aspects.

The third conceptual contribution of this dissertation is *tactics*. Tactics provide an excellent tradeoff between full dynamic application partitioning and rigid pre-determined application partitioning. Full dynamic partitioning occurs when applications can be partitioned, at runtime, in any arbitrary fashion. This form of partitioning can result in the theoretically optimal application performance. However, the high search space of this form of partitioning makes it intractable to use in practice. On the flip side, rigid pre-determined application partitioning is simple to use but will result in inefficient application performance in dynamic environments (as shown in Section 5.3). Tactics allows us to use the best features of both. Tactics list the small set (out of all possible partitions) of useful modular-level application partitions. At runtime, the tactic that will achieve the best application performance is picked. Tactics thus achieve the performance of dynamic partitioning while retaining the tractability of static partitioning.

The fourth conceptual contribution is the *RapidRe* process for rapidly retargeting applications for cyber foraging. RapidRe uses three components: the Vivendi little language, the Chroma runtime system, and a smart stub generator. It uses these components as part of a four stage process that developers can use to rapidly retarget applications for cyber foraging (as shown in Figure 2.3). Two parts of the process require significant developer attention: step 1 where the adaptive characteristics of the application are described using Vivendi and step 3 where application-specific APIs are inserted to create the modified client and server components. To make these two steps more manageable, they have been further broken down into multiple subtasks (as shown in Table 4.3). Each of these subtasks has been well documented and made as easy as possible for developers to understand and complete correctly.

The fifth conceptual contribution is the idea that mobile devices should be able to benefit from *overprovisioned environments*. Traditionally, mobile adaptive systems have been built to use only as many servers as normally required by the applications. However, with computers steadily becoming cheaper, it is quite likely that mobile devices will encounter environments that have an abundance of server resources – possibly because the environment is currently underutilized. This dissertation is one of the first to recognize this and present a solution that allows mobile devices to seamlessly exploit extra resources to improve application performance.

The sixth conceptual contribution is the development of a *validation approach* for investigating the usefulness of software tools. Historically, systems evaluations have concentrated on measurable performance metrics such as latency, cache hit rate, throughput, and overhead. Unfortunately, usually very little validation is performed, beyond a description of personal usage experience, to quantify how easy it is for the target audience to use their tools or system. This validation is made harder as traditional HCI evaluations cannot be easily applied as they concentrate mostly on evaluating user interfaces and not low-level programming tools and system interfaces. This dissertation aims to remedy this situation. It presents one of the first rigorous evaluations of the usability of low-level programming tools and shows that such an evaluation can a) be attempted with confidence, and b) produce excellent results about the usability of the system. This evaluation combines both traditional user-centric evaluation methods with system-centric evaluation methods to provide a complete understanding of the effectiveness of RapidRe. Additionally, this dissertation contains enough information about how to setup and conduct such an evaluation that I am confident it will be a useful template for other systems researchers hoping to evaluate the usability of their systems.

7.1.2 Artifacts

This dissertation has produced a number of tangible software artifacts. The first artifact is Chroma. Chroma is a fully functional dynamic adaptive runtime system that can achieve good application performance in cyber foraging environments. In particular, it can determine the current resource availability in the environment and pick the optimal operation setting for an application that both matches the resource availability and satisfies the user preferences. Chroma uses many components that were developed for Odyssey, such as the resource measurers and the resource demand predictors. The main differences from Odyssey is the tactic and user preference support. This is explained in more detail in Appendix A.

The second artifact is a complete specification of Vivendi. Vivendi allows developers to completely specify the necessary information about an application that needs to be known before the application can be adapted effectively in a mobile environment.

The third artifact is a sophisticated stub generator that can accept Vivendi descriptions and generate most of the code to interface the application with Chroma. The stub generator will generate application-specific APIs that need to be manually inserted by a developer to complete the retargeting process. The stub is sophisticated enough to generate correct code even if the application has multiple parallel stages or uses data decomposition (the split and join functions must be written by the developer).

The fourth and final artifact is a large set of useful mobile applications (described in Chapter 3). These applications were obtained from other developers. However, all of them have been successfully modified, using RapidRe, for use on small mobile devices.

7.1.3 Evaluation results

The evaluation results presented in this dissertation have contributed several important findings. The evaluation has two distinct flavors: the first part of the evaluation shows that RapidRe can be used by developers to quickly and effectively retarget large unfamiliar applications for cyber foraging while the second part shows that Chroma can effectively and efficiently adapt applications in a cyber foraging environment.

The evaluation of RapidRe presented several important results. First, it is one of the few rigorous evaluations of the usability of low-level software tools. The details of the two-phase (user-centric followed by system-centric) evaluation can serve as a template for other researchers who want to evaluate the usability of their systems. The user study component of the evaluation showed that novice developers could be trained to use RapidRe in under one hour. Even with this short training, novices could still retarget unfamiliar large applications in under four hours each. In the system-centric portion of the evaluation, I showed that even with this short retargeting times, the quality of the retargeted applications was still excellent – there were very few errors in the retargeted applications. More impressively, most of the retargeted applications had comparable performance to applications that were retargeted by experts who understood both Chroma and the applications intimately. Analysis of the retargeted applications that did not have comparable performance revealed just two types of errors that led to the performance difference. I am confident that a simple procedural change to RapidRe will, in the future, make it harder for developers to commit these two types of errors.

The evaluation of Chroma also presented several important results. First, it showed that tactics can be used to achieve excellent application performance in mobile environments. It then showed that dynamic user preferences are vital for optimal application performance in a dynamic environment. In particular, a static utility function will result in inefficient results. Next, it showed that most of Chroma's overhead arises from measuring the resource availability on remote servers. This overhead can be reduced with clever caching techniques. The evaluation then showed how using extra server resources can significantly improve application performance; up to a 4x improvement when using data decomposition. Finally, the evaluation showed that a simple client-side verification mechanism had the potential to protect clients against malicious servers.

7.2 Future work

In this section, I present avenues for future work on RapidRe and Chroma.

7.2.1 RapidRe

In Section 4.7.2, it was discovered that RapidRe could be improved by improving the "Find Fidelities" subtask by making developers aware of common pitfalls when describing parameters and using fidelity variables. It was also discovered that developers would benefit from solutions that allow them to better keep track of the minutiae of each subtask. In addition to the areas of improvement, there are other avenues for future RapidRe research.

7.2.1.1 Evaluate Data Decomposition Applications

The user study in Chapter 4 provided reasonable proof that RapidRe can be used for a broad range of applications. However, none of the applications used in the user study were retargeted explicitly to use data decomposition. In particular, participants did not have to create split and join functions for any application. In the future, I plan to conduct further user studies to test if developers can retarget applications, such as c2dfft and PopUp, explicitly for data decomposition in a reasonable amount of time.

7.2.1.2 Extend RapidRe to Other Adaptive Domains

A very interesting area of future work lies in investigating the applicability of RapidRe to other adaptive domains. These domains include grid computing, self-configuring systems, and web services. Grid computing, in particular, can already benefit from Chroma and RapidRe. The only differences are that a) most grid applications will use data decomposition, and b) the utility function that Chroma's solver uses will optimize for overall system throughput instead of user-centric goals.

To support the other domains, I will first have to carefully identify the adaptive requirements of applications for that domain. For example, how exactly does a web service adapt? This knowledge can be used to separate the application-specific information from the general mechanisms needed to perform the domain-specific adaptation. For example, for mobile computing, the general adaptation mechanisms are resource monitoring, resource prediction, and solving for an optimal solution. The application-specific information, that is used to guide the solver, is the list of tactics, parameters, and fidelity variables. The general adaptation mechanisms can then be built into a common domain-specific runtime; if such a runtime doesn't already exist.

With this knowledge separation, I can develop a little language, similar to Vivendi, that will allow developers to easily express the application-specific portions of information. The generic mechanisms can then be built into a common runtime. This syntax allows the use of a stub generator that can create the code required to interface an application with the domain-specific runtime.

7.2.1.3 Case Study: Self Configuring Systems

In this section, I explain how RapidRe could potentially be used to rapidly retarget applications for use with a self-configuring system. I use *Rainbow* [83] as the example

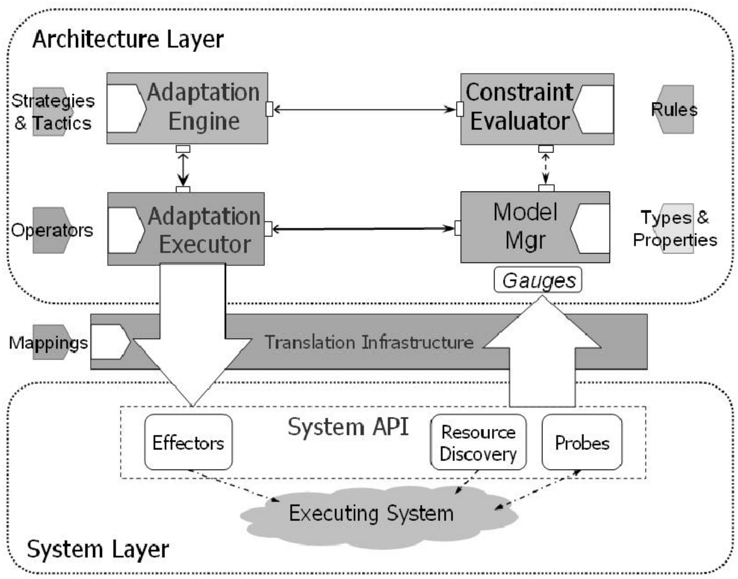


Figure 7.1: The Rainbow Architecture

self-configuring system. The goal of Rainbow is to reduce the cost and improve the reliability of making changes to complex systems. The idea is to develop a smart runtime monitoring system that uses architectural models, explicit representation of user tasks, and performance-oriented run-time gauges to determine if the overall system is functioning optimally. If inefficiencies are detected, Rainbow will reorganize the applications used by the system to fix the inefficiencies. For example, Rainbow is monitoring a Voice-over-IP system that uses five different applications and ten machines. If Rainbow detects that either a machine or application is not functioning properly, it might decide to either use a different application or route around the machine to fix the problem.

Figure 7.1 shows the Rainbow architecture. It consists of two layers. The architecture layer contains the Rainbow runtime adaptation engine. This layer determines if the current system is efficiently meeting the requirements of the architectural model and constraints. The system layer is the actual working system (which can comprise of multiple applications and machines). Each application and machine in the system needs to be augmented with a Rainbow API that a) facilitates communication with Rainbow, and b) exchanges QoS metrics with Rainbow (Rainbow uses these metrics to detect inefficiencies).

Currently, every application needs to be hand-modified to support Rainbow. These modifications can take up to two weeks per applications as each application needs to a) establish a connection with Rainbow, b) understand the messages that Rainbow sends to it, and c) provide appropriate responses to these messages.

Fortunately, most of this work can be automated making Rainbow a prime candidate for RapidRe. In particular, Rainbow already separates the application-specific information (the system layer) from the general mechanisms (the architectural layer) needed to perform the adaptation. As mentioned in Section 4.7.1, this separation is a key reason for RapidRe's success.

The three key steps that need to be done before Rainbow can use RapidRe are a) identifying the exact application-specific bits of information that Rainbow requires, b) developing a syntax that allows those bits of information to be easily specified, and c) developing a stub generator that can generate the necessary interface code. The application-specific information needed by Rainbow is very similar to tactics, parameters, and fidelity variables as Rainbow also needs to understand the resource usage and the adaptation possibilities of each application to achieve optimal application performance when resources or user preferences change. In addition, Rainbow also needs to understand the exact services offered by each application. For example, Panlite is a language translator that can translate Spanish to English and English to Spanish. The stub generator will require the most amount of time to create – $\approx$ 1-2 weeks. However, once this is done, subsequent applications can be retargeted in just a few hours. I am currently working with Rainbow developers and I hope to build and evaluate a complete RapidRe solution for Rainbow in the near future.

7.2.2 Chroma

In addition to adding proper service discovery, server instantiation, and security to Chroma (discussed in Appendix A.7.1), there are other improvements that can be made to Chroma.

- **Better Solver Mechanisms**: As stated in Appendix A.6, Chroma has two different solvers. There is a general solver that solves for the optimal operation setting. There is also a data decomposition solver that determines a set of servers that can be used for the decomposition.

  The normal solver uses a very simple computationally intensive algorithm (shown in Figure A.7). Even though this algorithm has a reasonably low overhead (shown in Figure 5.4), in the future, I plan to replace the solver with an even less computationally demanding solver. More importantly, I hope to replace the current solver with a provably correct solver. Examples of such a provably correct solver includes the multi-resource solver developed by Lee et al. [138].

  The data decomposition solver is also in a fairly preliminary stage. It currently uses static thresholds to determine if a remote server is good enough to be used. In the future, the solver should use thresholds obtained from the user. The decomposition solver also currently doesn't handle multiple tactics and fidelity variables.

- **Lower Overhead Chroma Runtime**: As shown in Figure 5.6, Chroma has fairly high overheads when measuring the resource availability on remote servers. In the next version of Chroma, I hope to reduce these overheads through the use of better mechanisms and intelligent caching.

- **Deploying Chroma on Real Mobile Devices**: Finally, the ultimate goal for the next version of Chroma is a field study on actual small mobile devices. This would allow

Chroma to be tuned for the needs of real users.

7.2.3 Longer Term Research

The previous sections discussed shorter term research relevant to RapidRe and Chroma. In this section, I discuss longer term research in the general area of mobile and pervasive computing.

Designing the next generation mobile operating system : My thesis research has focused on legacy applications. However, in the near future, it is likely that legacy applications will become less important and that new applications will be built from scratch to support mobility. With this in mind, an interesting research question is the following: "What extensions do future mobile operating systems need to have to sufficiently support the next generation of adaptive mobile applications?". I personally believe that the OS will need to provide some sort of application-level resource prediction querying mechanism. This would allow higher level management systems to query the OS and figure out what performance any particular application would achieve if it ran with a particular set of fidelities under a particular resource environment. This knowledge would allow those management systems to accurately decide which applications (and with what particular fidelities) should be run to achieve the user's current goals.

However, building this resource prediction querying mechanism will require more research to understand exactly what types of predictive queries applications will want to ask and develop mechanisms to support those queries. For example, supporting a query of "If I ran this application with these settings right now, what latency would I achieve" requires less work than supporting a query of "I need this application to have a latency of less than 0.8 seconds. Tell me when and how I should run this application.". The example also shows that research in providing a query syntax that allows applications to accurately specify their requirements while keeping the syntax simple enough to be tractable and usable will also be necessary.

Managing multiple mobile devices : A popular vision of pervasive computing is that eventually, almost every device in the environment will have some sort of processing power embedded in it. These devices would be able to exchange information among themselves and use that information to better support users. For example, my fridge may realize that it is out of milk. It sends that information to my PDA which automatically adds it to my shopping list.

However, a major concern with this vision is dealing with the complexity of managing multiple smart devices. It is already the case that most users cannot effectively manage their personal computers – the computer is either not up-to-date with security patches etc. or the user is unable to fix problems that arise with various applications running on the computer. This management problem is only going to get worse when the number of smart devices in the environment increases. For example, what happens if my fridge tells my PDA to buy beef instead of milk? How easy is it for me to debug the problem and figure out exactly which device or software component needs to be fixed? In a similar vein, is it easy for me to tell my fridge to send a short message service (SMS) notice to my cellphone instead of changing the shopping list on my PDA?

I claim that normal users will not want to use pervasive technology if they are unable to fix common problems or change their requirements without calling in an expert. However, providing this capability will require building management tools and services that will quickly and easily allow users to update their personal requirements and to locate, identify, and fix problems. This is a very challenging area of research that is both very interesting and extremely important and timely.

Making Pervasive Computing a reality : My ultimate research goal is to understand why, even after 10-15 years of research, pervasive computing is not as pervasive as many people predicted. At best, it has achieved a small foothold, in the form of context software on cellphones, in some Asian and European countries. However, overall, pervasive computing still remains confined to academic research and has not been widely deployed. I speculate that understanding the deep reasons for this lack of deployment will require finding answers for the following three sub-questions.

- *Is there no killer pervasive application?* : The most pervasive mobile application seems to be the SMS on cellphones. Indeed, it is a highly lucrative revenue source for many Asian and European cellphone providers. Unfortunately, SMS is similar to the functionality of a cellphone in that it is fundamentally a person-to-person communication technology. SMS became incredibly popular as it allowed people to communicate with each other in situations where placing a phone call would not be as convenient. Does this then mean that communication technology is the only kind of application that mobile users desire? Is there no other killer application out there that will convince the masses to jump onto the pervasive bandwagon? I personally think that mobile gaming has the potential to be such a killer application. It is addictive, simple to use, massively multiplayer-friendly, amenable to context-sensitive technology, and deployable on existing mobile devices such as cellphones. However, a large amount of research still needs to be done before a usable and highly addictive mobile gaming service can be deployed.

  I also believe that another potentially "killer" pervasive application would be a personal butler service. This service would store all your credit card information, phone book, context information, and eventually even contain electronic cash. The vision is that this electronic butler would effectively replace your wallet. Such an application would mean that people would not need to manually carry a huge stack of cards and cash with them wherever they go. However, before this application can become reality, a large amount of research in user interfaces, management toold, authentication, security, storage, backup, and revocation needs to be done.

- *Do we need to develop better pervasive computing infrastructure?* : Before pervasive computing can become literally pervasive, the appropriate infrastructure needs to be deployed. This infrastructure can be in the form of the hardware needed to support pervasive computing, such as sensors, power lines, wireless access points etc. Deploying and maintaining this hardware is a tricky endeavour. However, I believe that this is the simpler problem to solve. The more difficult problem is to develop the software that would allow providers and end-users of pervasive technology to quickly, easily, effectively, and securely manage and use their pervasive environments. As mentioned above, I believe that there is still a large amount of research that needs to be done in this area of mobile device management. This research will need to span multiple areas of research. For example, it will be necessary to develop systems protocols and mechanisms for managing a large number of devices. It will also be necessary to develop AI techniques to automatically predict user behaviour and correctly manage the system as much as possible without user interventation. In addition, software architectures that allow the development of clean, modular, and easy to manage pervasive systems will need to be created. Finally, HCI expertise will need to be leveraged to develop excellent GUIs that allow users to easily manage and operate various pervasive applications and/or devices.

- *Do we need a better business model?* : A possible problem with deploying pervasive technology is a lack of a good business model. For example, what is the economic incentive for technology companies to develop and deploy new pervasive solutions? I am very interested in collaborating with business experts to understand how to evaluate and understand the economic and business implications of deploying pervasive technology. I believe that this understanding will help focus subsequent research towards avenues that are more likely to be economically and practically viable.

7.3 Closing Remarks

Mobile computing is at a crossroads today. A decade of sustained research effort has developed the core concepts, techniques, and mechanisms to provide a solid foundation for progress in this area. Yet, mass-market mobile computing lags far behind the frontiers explored by researchers. Smart cell phones and PDAs define the extent of mobile computing experience for most users. Laptops, though widely used, are best viewed as portable desktops rather than true mobile devices that are always with or on a user. Wearable computers have proven effective in industrial and military settings [198, 240], but their impact has been negligible outside niche markets.

An entirely different world, that this thesis addresses, awaits discovery. In that world, mobile computing augments the cognitive abilities of users by exploiting advances in areas such as speech recognition, natural language processing, image processing, augmented reality, planning and decision-making. This can transform business practices and user experience in many segments such as travel, health care, and engineering. Will we find this world, or will it remain a shimmering mirage forever?

We face two obstacles in reaching this world. The first is a technical obstacle: running resource-intensive applications on resource-poor mobile hardware. Remote execution can remove this obstacle, provided one can count on access to a compute server via a wireless network. The second obstacle is an economic one. The effort involved in creating applications of this new genre from scratch is enormous, requiring expertise in both the application domain and in mobile computing. Further, there is no incentive to publicly deploy compute servers if such applications are not in widespread use. We thus have a classic deadlock, in which applications and infrastructure each await the other.

This dissertation aims to break this deadlock by providing solutions to both obstacles. It uses the concept of tactics to develop a runtime system called Chroma. Chroma allows resource-poor mobile hardware to exploit fidelity adaptation and remote execution to achieve good performance even when running resource-intensive applications. This dissertation also presents a solution for lowering the cost of creating resource-intensive mobile applications by reusing existing software that was created for desktop environments. Using the RapidRe approach, relatively unskilled programmers can do an excellent job of rapidly porting such software to new mobile devices. The results are close to optimal in terms of performance in most cases. RapidRe also makes it easier to fix incorrectly retargeted applications as only a small number of changes need to be made to the application source. RapidRe and Chroma have been extensively validated and shown to be effective solutions. Even so, this dissertation has just scratched the tip of the iceberg in addressing the issues necessary for making true mobile computing a reality. Necessary components such as ubiquitous service discovery and security still remain to be resolved. Nonetheless, I am confident that this dissertation is a major step in the right direction and that it can help stimulate the transformation of mobile computing.

Bibliography

[1] Secure shell protocol, Sept. 2005. `http://www.openssh.org/`, Version 4.2.

[2] Transistor counts of AMD processors. `http://sandpile.org/`, May 2006.

[3] Transistor counts of Intel processors. `http://www.intel.com/pressroom/kits/events/moores_law_40th/`, May 2006.

[4] M. Abbaspour and J. Zhu. Retargetable binary utilities. In *Proceedings of the 39th Design Automation Conference (DAC)*, New Orleans, LA, June 2002.

[5] G. B. Adams, III, D. P. Agrawal, and H. J. Siegel. A survey and comparison of fault-tolerant multistage interconnection networks. *IEEE Computer*, 20(6):14–29, June 1987.

[6] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, December 1999.

[7] A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital dynamic telepathology - the virtual microscope. In *Proceedings of the American Medical Informatics Association (AMIA) Annual Fall Symposium*, Lake Buena Vista, FL, Nov. 1998. Also available as University of Maryland Technical Report CS-TR-3892 and UMIACS-TR-98-23.

[8] D. Aguayo, J. C. Bicket, S. Biswas, G. Judd, and R. Morris. Link-level measurements from an 802.11b mesh network. In *Proceedings of the ACM SIGCOMM Conference*, Portland, OR, Aug. 2004.

[9] A. Akella, S. Seshan, and A. Shaikh. An empirical evaluation of wide-area internet bottlenecks. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement (IMC)*, Oct. 2003.

[10] D. W. Albrecht, I. Zukerman, A. E. Nicholson, and A. Bud. Towards a bayesian model for keyhole plan recognition in large domains. In *Proceedings of the Sixth International Conference on User Modeling (UM)*, Sardinia, Italy, June 1997.

[11] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[12] P. G. And. Secure distributed computing in a commercial environment. In *Proceedings of the 5th International Conference on Financial Cryptography*, Grand Cayman, British West Indies, Feb. 2002.

[13] The Apache Software Foundation. *Apache Ant*, June 2005. `http://ant.apache.org/`, Version 1.6.5.

[14] A. Ardo. Experience acquiring and retargeting a portable ada compiler. *Software - Practice and Experience (SPE)*, 17(4):291–307, Apr. 1987.

[15] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster i/o with river: Making the fast case common. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems (IOPADS)*, Atlanta, GA, May 1999.

[16] Y. Artsy and R. A. Finkel. Designing a process migration facility: The charlotte experience. *IEEE Computer*, 22(9):47–56, 1989.

[17] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre. Recent advances in augmented reality. *IEEE Computer Graphics and Applications*, 21(6):34–47, Nov. 2001.

[18] E. C. Bailey, P. Nasrat, M. Saou, and V. Skyttä. Maximum rpm. `http://www.rpm.org/max-rpm-snapshot/`, July 2005.

[19] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, Mar. 1992.

[20] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 273–286, San Francisco, CA, May 2003.

[21] M. R. Barbacci and J. M. Wing. Durra: A task-level description language. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, University Park, PA, Aug. 1987.

[22] M. R. Barbacci and J. M. Wing. Durra: A task-level description language reference manual (version 2). Technical Report CMU/SEI-89-TR-34, Carnegie Mellon University, Pittsburgh, PA, Sept. 1989.

[23] J. Basney and M. Livny. Improving goodput by co-scheduling cpu and network capacity. *International Journal of High Performance Computing Applications*, 13(3), Fall 1999.

[24] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS)*, Fairfax, VA, Nov. 1993.

[25] J. Bentley. Programming pearls: Little languages. *Communications of the ACM (CACM)*, 29(8):711–721, 1986.

[26] T. Berson, D. Dean, M. Franklin, D. Smetters, and M. Spreitzer. Cryptography as a network service. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2001.

[27] G. Biegel and V. Cahill. A framework for developing mobile, context-aware applications. In *Proceedings of the Second IEEE Conference on Pervasive Computing and Communications (Percom)*, Orlando, FL, Mar. 2004.

[28] G. Biegel, V. Cahill, and M. Haahr. A dynamic proxy-based architecture to support distributed java objects in mobile environments. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, Irvine, CA, Oct. 2002.

[29] M. Billinghurst, S. Weghorst, and T. A. Furness. Wearable computers for three dimensional CSCW. In *Proceedings of the International Symposium on Wearable Computers*, pages 39–46, Cambridge, MA, Oct. 1997.

[30] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, Bretton Woods, NH, Oct. 1983.

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

[32] A. W. Black and K. A. Lenzo. Flite: a small fast run-time synthesis engine. In *4th ISCA Tutorial and Research Workshop on Speech Synthesis*, Perthshire, Scotland, Aug. 2001.

[33] A. W. Black and K. A. Lenzo. FLITE: a small fast run-time synthesis engine. `http://www.speech.cs.cmu.edu/flite/`, Feb. 2003. (Version 1.2).

[34] G. E. Blelloch. Nesl: A nested data-parallel language (3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University, Pittsburgh, PA, Sept. 1995.

[35] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the fourth*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), San Diego, CA, May 1993.

[36] G. Brown, W. Luk, and J. O'Leary. Retargeting a hardware compiler proof using protocol converters. In *Proceedings of the First International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Utah, USA, Nov. 1994.

[37] F. Budinsky, G. P. DeCandio, R. Earle, T. Francis, J. Jones, J. Li, M. Nally, C. Nelin, V. Popescu, S. Rich, A. G. Ryman, and T. W. Wilson. Websphere studio overview. *IBM Systems Journal*, 43(2):384–419, May 2004.

[38] F. E. Bustamante, G. Eisenhauer, P. Widener, K. Schwan, and C. Pu. Active streams: An approach to adaptive distributed systems. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS)*, Elmau/Oberbayern, Germany, May 2001.

[39] L. Cardelli. A language with distributed scope. In *In Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, San Francisco, CA, Jan. 1995.

[40] L. Cardelli. Obliq. `http://www.luca.demon.co.uk/Obliq/Obliq.html`, 2005.

[41] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186, New Orleans, LA, Feb. 1999.

[42] U. Catalyurek, M. D. Beynon, C. Chang, T. Kurc, A. Sussman, and J. Saltz. The virtual microscope. *IEEE Transactions on Information Technology in Biomedicine*, 7(4):230–248, Dec. 2003.

[43] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. C compiler retargeting based on instruction semantics models. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Munich, Germany, Mar. 2005.

[44] B. Chen and R. Morris. Certifying program execution with secure processors. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HOTOS)*, Lihue, HI, May 2003.

[45] J. Cheng. Exploiting excess computational resources in remote execution. Master's thesis, Information Networking Institute, Carnegie Mellon University, Nov. 2005.

[46] D. C. Chu and M. A. Humphrey. Mobile ogsi.net: Grid computing on mobile devices. In *Proceedings of the Grid Computing Workshop*, Pittsburgh, PA, Nov. 2004.

[47] Citrix Online, LLC. *GoToMyPC: Remote Access to Your PC From Anywhere*, June 2005. `http://www.gotomypc.com/`.

[48] B. Clarke and M. Humphrey. Beyond the device as portal: Meeting the requirements of wireless and mobile devices in the legion grid computing system. In *Proceedings of the Second International Workshop on Parallel and Distributed Computing: Issues in Wireless Networks and Mobile Computing*, Ft. Lauderdale, FL, Apr. 2002.

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

[50] Computer Science and Mathematics Division, Oak Ridge National Laboratory. *The Parallel Virtual Machine*, Sept. 2004. `http://www.csm.ornl.gov/pvm/pvm_home.html`, Version 3.4.5.

[51] The Condor Project. *Submitting a Job to Condor*, Aug. 2005. `http://www.cs.wisc.edu/condor/manual/v6.7/2_5Submitting_Job.html`, Version 6.7.10.

[52] R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext. In *Proceedings of the 18th Annual IACR Crypto Conference (CRYPTO)*, Santa Barbara, CA, Aug. 1998.

[53] S. E. Czerwinski, B. Y. Zhao, T. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *Proceedings of the Fifth Annual International Conference on Mobile Computing and Networks (MobiCOM)*, Seattle, WA, Aug. 1999.

[54] E. de Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, Berkeley, CA, March 2001.

[55] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[56] The Debian Project. Aug. 1993. `http://www.debian.org/`.

[57] The Debian Project. *Debian New Maintainers' Guide*, Jan. 2005. `http://www.debian.org/doc/maint-guide/`, Version 1.2.3.

[58] H. M. Deitel and P. J. Deitel. *C++ How to Program (4th Edition)*. Prentice-Hall, Englewood Cliffs, New Jersey, Aug. 2002.

[59] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, Nov. 1976.

[60] F. Douglis. Transparent process migration in the sprite operating system. Technical Report CSD-90-598, University of California, Berkeley, 1990.

[61] F. Douglis and J. K. Ousterhout. Transparent process migration: Design alternatives and the sprite approach. *Software - Practice and Experience (SPE)*, 21(7):1–27, July 1991.

[62] A. H. Dutoit, O. Creighton, G. Klinker, R. Kobylinski, C. Vilsmeier, and B. Bruegge. Architectural issues in mobile augmented reality systems: a prototyping case study. In *Proceedings of the Eighth Asian Pacific Conference on Software Engineering (APSEC'2001)*, pages 341–344, Macau, China, Dec. 2001.

[63] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, and J. Zinky. Building adaptive distributed applications with middleware and aspects. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, Mar. 2004.

[64] K. Echtle. Fault-masking with reduced redundant communication. In *Proceedings of the 16th International Symposium on Fault-Tolerance Computing (FTCS)*, pages 178–183, Vienna, Austria, June 1986.

[65] D. Eckhardt and P. Steenkiste. Measurement and analysis of the error characteristics of an in-building wireless network. In *Proceeding of ACM SIGCOMM Conference*, pages 243–254, Stanford, CA, October 1996.

[66] The Eclipse Foundation. *Eclipse Platform*, June 2005. `http://www.eclipse.org/`, Version 3.1.

[67] J. Eisenstein, J. Vanderdonckt, and A. Puerta. Applying model-based techniques to the development of uis for mobile computers. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*, Santa Fe, NM, January 2001.

[68] M. A. Ertl and D. Gregg. Retargeting jit compilers by using c-compiler generated executable code. In *Proceedings of the Thirteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Antibes Juan-les-Pins, France, Sept. 2004.

[69] Federal Communications Commision. *License Database*, March 2003. `https://gullfoss2.fcc.gov/prod/oet/cf/eas/reports/GenericSearch.cfm`.

[70] S. Feiner, B. MacIntyre, T. Hollerer, and A. Webster. A touring machine: Prototyping 3d mobile augmented reality systems for exploring the urban environment. In *Proceedings of the International Symposium on Wearable Computers*, pages 74–81, Cambridge, MA, Oct. 1997.

[71] S. I. Feldman. Make-a program for maintaining computer programs. *Software - Practice and Experience (SPE)*, 9(4):255–265, Apr. 1979.

[72] N. Ferguson and B. Schneier. *Practical Cryptography*. John Wiley and Sons, Inc., Apr. 2003.

[73] R. Ferreira, B. Moon, J. Humphries, A. Sussman, J. Saltz, R. Miller, and A. Demarzo. The virtual microscope. In *Proceedings of the American Medical Informatics Association (AMIA) Annual Fall Symposium*, Nashville, TN, Nov. 1997. Also available as University of Maryland Technical Report CS-TR-3777 and UMIACS-TR-97-35.

[74] J. Flinn. *Extending Mobile Computer Battery Life through Energy-Aware Adaptation*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2001.

[75] J. Flinn, S. Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

[76] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 48–63, Kiawah Island, SC, December 1999.

[77] J. Flinn and M. Satyanarayanan. Managing battery lifetime with energy-aware adaptation. *ACM Transactions on Computer Systems (TOCS)*, 22(2):137–179, May 2004.

[78] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[79] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. `http://www.globus.org/alliance/publications/papers/ogsa.pdf`, June 2002.

[80] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the Seventh International ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, October 1996.

[81] R. Frederking and R. D. Brown. The pangloss-lite machine translation system. In *Expanding MT Horizons: Proceedings of the Second Conference of the Association for Machine Translation in the Americas*, pages 268–272, Montreal, Canada, 1996.

[82] S. Frolund and J. Koistinen. Quality-of-service specification in distributed object systems. *Distributed Systems Engineering Journal*, 5(4):179–202, Dec. 1998.

[83] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, Oct. 2004.

[84] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.

[85] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, Apr. 1989.

[86] P. B. Gibbons. A stub generator for multi-language rpc in heterogeneous environments. *IEEE Transactions on Software Engineering*, 13(1):77–87, Jan. 1987.

[87] C. Gill, J. M. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, and D. C. Schmidt. Integrated adaptive qos management in middleware: An empirical case study. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Toronto, Canada, May 2004.

[88] GNU Project. *GNU Autoconf*, Dec. 2003. `http://www.gnu.org/software/autoconf/`, Version 2.59.

[89] GNU Project. *GNU Automake*, July 2005. `http://www.gnu.org/software/automake/`, Version 1.9.6.

[90] GNU Project. *GNU Libtool*, Aug. 2005. `http://www.gnu.org/software/libtool/`, Version 1.5.20.

[91] GNU Project. *GNU m4*, Mar. 2005. `http://www.gnu.org/software/m4/`, Version 1.4.3.

[92] P. Golle and I. Mironov. Uncheatable distributed computations. In *Proceedings of RSA Conference – Topics in Cryptography*, pages 425–440, San Francisco, CA, Apr. 2001.

[93] S. Goyal and J. Carter. A lightweight secure cyber foraging infrastructure for resource-constrained devices. In *Proceedings of the Sixth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, English Lake District, United Kingdom, Dec. 2004.

[94] S. D. Gribble, M. Welsh, J. R. von Behren, E. A. Brewer, D. E. Culler, N. Borisov, S. E. Czerwinski, R. Gummadi, J. R. Hill, A. D. Joseph, R. H. Katz, Z. M. M. Mao, S. Ross, and B. Y. Zhao. The NINJA architecture for robust internet-scale systems and services. *Journal of Computer Networks*, 35(4):473–497, Mar. 2001. Special Issue on Pervasive Computing.

[95] A. S. Grimshaw and J. W. S. Liu. Mentat: An object-oriented data-flow system. In *Proceedings of the second ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Orlando, FL, October 1987.

[96] D. Gross. *Buy Cell: How many mobile phones does the world need?* Slate, June 2004. `http://slate.msn.com/id/2101625/`.

[97] J. Gross and A. Willig. Measurements of a wireless link in different RF-isolated environments. In *Proceedings of the European Wireless Conference*, Florence, Italy, Feb. 2002.

[98] S. Gurun, C. Krintz, and R. Wolski. NWSLite: A Light-Weight Prediction Utility for Mobile Devices. In *Proceedings of the Second International Conference on Mobile Computing Systems, Applications and Services (MobiSys)*, Boston, MA, June 2004.

[99] E. Guttman. Service location protocol: Automatic discovery of ip network services. *IEEE Internet Computing*, 3(4):71–80, July 1999.

[100] E. Guttman, C. E. Perkins, and M. Day. Service location protocol, version 2. *Internet RFC 2608*, June 1999.

[101] E. Guttman, C. E. Perkins, and J. Kempf. Service templates and service: Schemes. *Internet RFC 2609*, June 1999.

[102] M. Haahr, R. Cunningham, and V. Cahill. Supporting corba applications in a mobile environment. In *Proceedings of the Fifth International Conference on Mobile Computing and Networking (MobiCom)*, Seattle, WA, Aug. 1999.

[103] J. Haartsen. The bluetooth radio system. *IEEE Personal Communications*, 7(1):28–36, Feb. 2000.

[104] A. Haeberlen, J. Liedtke, Y. Park, L. Reuther, and V. Uhlig. Stub-code performance is becoming important. In *First Workshop on Industrial Experiences with Systems Software (WIESS)*, Berkeley, CA, Oct. 2000.

[105] K. Harfoush, A. Bestavros, and J. Byers. Measuring bottleneck bandwidth of targeted path segments. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOMM)*, Apr. 2003.

[106] R. Hartono, N. Jangkrajarng, S. Bhattacharya, and C.-J. R. Shi. Automatic device layout generation for analog layout retargeting. In *Proceedings of the Eighteenth International Conference on VLSI Design*, Kolkata, India, Jan. 2005.

[107] D. Hoiem and T. Riker. POPUP source code and online documentation. `http://www.cs.cmu.edu/~dhoiem/projects/popup/`, May 2005.

[108] A. Homann, H. Meyr, and S. Pees. Retargeting of compiled simulators for digital signal processors using a machine description language. In *Proceedings of the conference on Design, automation and test in Europe*, Paris, France, Mar. 2000.

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

[110] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. The lumière project: Bayesian user modeling for inferring the goals and needs of software users. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, Madison, Wisconsin, July 1998.

[111] N. Hu, L. E. Li, Z. M. Mao, P. Steenkiste, and J. Wang. Locating internet bottlenecks: Algorithms, measurements, and implications. In *Proceedings of the ACM SIGCOMM Conference*, Sept. 2004.

[112] A.-C. Huang and P. Steenkiste. Building self-configuring services using service-specific knowledge. In *Proceedings of the Thirteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, Honolulu, HI, June 2004.

[113] A.-C. Huang and P. Steenkiste. Building self-adapting services using service-specific knowledge. In *Proceedings of the Fourteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, Research Triangle Park, NC, July 2005.

[114] M. A. Humphrey. From legion to legion-g to ogsi.net: Object-based computing for grids. In *Proceedings of the IPDPS NSF Next Generation Software Workshop*, Nice, France, Apr. 2003.

[115] G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *Proceedings of the 3rd Symposium on Operating System Design and Implemetation (OSDI)*, pages 187–200, New Orleans, LA, Feb. 1999.

[116] V. Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 1988. Stanford, CA.

[117] V. Jacobson. pathchar - a tool to infer characteristics of internet paths. `ftp://ftp.ee.lbl.gov/pathchar`, Apr. 1997.

[118] M. Jakobsson and A. Juels. Proofs of work and bread pudding protocols. In *Proceedings of the IFIP TC6 and TC11 Joint Working Conference on Communications and Multimedia Security (CMS)*, Leuven, Belgium, Sept. 1999.

[119] M. Jeronimo and J. Weast. *UPnP Design by Example*. Intel Press, 2003.

[120] A. Johnsson, B. Melander, and M. Bjo:rkman. Bandwidth measurement in wireless networks. In *Proceedings of the 4th Mediterranean Ad Hoc Networking Workshop*, Porquerolles, France, June 2005.

[121] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and F. M. Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 156–171, Copper Mountain, Co., Dec. 1995.

[122] J. Joseph. A developer's overview of ogsi and ogsi-based grid computing. `http://www.ibm.com/developerworks/grid/library/gr-ogsi/`, Apr. 2003.

[123] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, Austin, TX, Nov. 1987.

[124] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):109–133, Feb. 1988.

[125] Z. Kalbarczyk and J. Christmansson. Design principles for software fault tolerance - a survey. Technical Report 149, CTH, Dept. of Computer Engineering, Laboratory for Dependable Computing (LDC), 1993.

[126] M. Kanellos. *Nation: Techno-revolution in the making*. CNET news.com, June 2004. `http://news.com.com/Nation+Techno-revolution+in+the+making+-+Part+1+of+%South+Koreas+Digital+Dynasty/2009-1040_3-5239544.html`.

[127] D. A. Karr, C. Rodrigues, Y. Krishnamurthy, I. Pyarali, and D. C. Schmidt. Application of the quo quality-of-service framework to a distributed video application. In *Proceedings of the Third International Symposium on Distributed Objects and Applications (DOA)*, Rome, Italy, Sept. 2001.

[128] R. H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1):6–17, First Quarter 1994.

[129] Y. Ke, D. Hoiem, and R. Sukthankar. MUSIC source code and online documentation. `http://www.cs.cmu.edu/~yke/musicretrieval/`, Aug. 2004.

[130] S. Kent and R. Atkinson. Security architecture for the internet protocol. *Internet RFC 2401*, Nov. 1998.

[131] B. W. Kernighan and D. M. Richie. *The C Programming Language (2nd Edition)*. Prentice-Hall, Englewood Cliffs, New Jersey, Mar. 1988.

[132] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, Jyväskylä, Finland, June 1997.

[133] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1), Feb. 1992.

[134] S. R. Klemmer, J. Li, J. Lin, and J. A. Landay. Papier-mache: Toolkit support for tangible input. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 399–406, Vienna, Austria, Apr. 2004.

[135] A. J. Ko, H. H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented ides: A detailed study of corrective and perfective maintenance tasks. In *Proceeding of the 27th International Conference on Software Engineering (ICSE)*, St. Louis, MO, May 2005.

[136] Y. Krishnamurthy, V. Kachroo, D. A. Karr, C. Rodrigues, J. P. Loyall, R. Schantz, and D. C. Schmidt. Integration of qos-enabled distributed object computing middleware for developing next-generation distributed applications. In *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM)*, Snowbird, UT, June 2001.

[137] K. Lakshminarayanan, V. N. Padmanabhan, and J. Padhye. Bandwidth estimation in broadband access networks. In *Proceedings of the 4th ACM/USENIX Internet Measurement Conference (IMC)*, Taormina, Sicily, Italy, Oct. 2004.

[138] C. Lee, J. P. Lehoczky, D. P. Siewiorek, R. Rajkumar, and J. P. Hansen. A scalable solution to the multi-resource qos problem. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*, pages 315–326, Phoenix, AZ, Dec. 1999.

[139] S. Leue. Qos specification based on sdl/msc and temporal logic. In *Proceedings of the Montreal Workshop on Multimedia Applications and Quality of Service Verification*, Montreal, Canada, May 1994.

[140] C. Lewis and J. Rieman. Task-centered user interface design: A practical introduction. `ftp://ftp.cs.colorado.edu/pub/cs/distribs/clewis/HCI-Design-Book/`, Chapter 5, Sept. 1994.

[141] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 140:1–55, June 1932.

[142] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, and K. Anderson. Qos aspect languages and their runtime integration. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR)*, Pittsburgh, PA, May 1998.

[143] J. Martin. *Fourth-Generation Languages*, volume 1: Principles. Prentice-Hall, 1985.

[144] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proceedings of the ACM SIGCOMM Conference*, Stanford, CA, Aug. 1999.

[145] Microsoft Corporation. *Visual Studio*, June 2000. `http://msdn.microsoft.com/vstudio/`.

[146] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[147] E. Monteiro, F. Boavida, G. Quadros, P. Marrocos, P. Ii, and V. Freitas. Specification, quantification and provision of quality of service and congestion control for new communication services. In *Proceedings of the 16th Association for Communications, Electronics, Intelligence and Information Systems Professionals (AFCEA) Europe Symposium*, Brussels, Belgium, Oct. 1995.

[148] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, Apr. 1965. Available from `http://www.intel.com/technology/mooreslaw/index.htm`. Checked on May 28th 2006.

[149] The MPI Forum. *The Message Passing Interface*, May 1998. `http://www.mpi-forum.org/`, Version 2.0.

[150] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[151] D. Narayanan. *Operating System Support for Mobile Interactive Applications*. PhD thesis, School of Computer Science, Carnegie Mellon University, Aug. 2002.

[152] D. Narayanan and M. Satyanarayanan. Predictive resource management for wearable computing. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, CA, May 2003.

[153] C. Narayanaswami, N. Kamijoh, M. Raghunath, T. Inoue, T. Cipolla, J. Sanford, E. Schlig, S. Venkiteswaran, D. Guniguntala, V. Kulkarni, and K. Yamazaki. IBM's linux watch: The challenge of miniaturization. *IEEE Computer*, 35(1):33–41, Jan. 2002.

[154] A. Natrajan, A. Nguyen-Tuong, M. A. Humphrey, M. Herrick, B. P. Clarke, and A. S. Grimshaw. The legion grid portal. *Concurrency and Computation: Practice and Experience*, 14(13–15):1365–1394, Sept. 2002.

[155] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, Paris, France, Jan. 1997.

[156] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM (CACM)*, 21(12):993–999, 1978.

[157] J. Nichols, B. A. Myers, M. Higgins, J. Hughes, T. K. Harris, R. Rosenfeld, and M. Pignol. Generating remote control interfaces for complex appliances. In *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology (UIST)*, Oct. 2002.

[158] J. Nielsen. *Usability Engineering*. Academic Press, San Diego, CA, 1993.

[159] NIST/ITL Advanced Network Technologies Division. *Nist Net*, Jan. 1998. `http://snad.ncsl.nist.gov/nistnet/`.

[160] B. D. Noble, M. Satyanarayanan, D. Narayanan, E. J. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 276–287, Saint-Malo, France, October 1997.

[161] M. Nuttall. A brief survey of systems providing process or object migration facilities. *Operating Systems Review (OSR)*, 28(4), Oct. 1994.

[162] Object Management Group. *CORBA Basics*, May 2005.

[163] Object Management Group. *Unified Modeling Language*, May 2005.

[164] D. R. O'Hallaron. C2DFFT source code. `http://www.cs.cmu.edu/~droh/c2dfft.tar`, Feb. 1996. (Accessed on May 11 2005).

[165] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[166] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The sprite network operating system. *IEEE Computer*, 21(2):23–36, Feb. 1988.

[167] P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier. Using qdl to specify qos aware distributed (quo) application configuration. In *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, Newport Beach, CA, Mar. 2000.

[168] J. F. Pane and B. A. Myers. Usability issues in the design of novice programming systems. Technical Report CMU-HCII-96-101, Carnegie Mellon University, Pittsburgh, Pennsylvania, Aug. 1996.

[169] J. A. Paradiso and T. Starner. Energy scavenging for mobile and wireles electronics. *IEEE Pervasive Computing Magazine*, 4(1):18–27, January - March 2005.

[170] G. D. Parrington. A stub generation system for c++. *Computing Systems*, 8(2):135–169, Spring 1995.

[171] C. E. Perkins and E. Guttman. Dhcp options for service location protocol. *Internet RFC 2610*, June 1999.

[172] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint Malo, France, Oct. 1997.

[173] V. Poladian, J. P. Sousa, D. Garlan, and M. Shaw. Dynamic configuration of resource-aware services. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 604–613. IEEE Computer Society Press, 2004.

[174] M. L. Powell and B. P. Miller. Process migration in demos/mp. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 1983.

[175] B. Raman and R. Katz. An architecture for highly available wide-area service composition. *Computer Communications Journal, Special Issue on 'Recent Advances in Communication Networking'*, 26(15):1727–1740, Sept. 2003.

[176] Red Hat Incorporated. June 1993. `http://http://www.redhat.com/`.

[177] I. Reid. Rpcc - a stub compiler for sun rpc. In *Proceedings of the USENIX Summer Conference*, Phoenix, Arizona, June 1987.

[178] N. Reynolds and D. Duchamp. Measured performance of a wireless lan. In *Proceedings of the 17th IEEE Conference on Local Computer Networks (LCN)*, pages 494–499, Minneapolis, MN, Sept. 1992.

[179] Richardson, T., Stafford-Fraser, Q., Wood, K. R., and Hopper, A. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, Jan/Feb 1998.

[180] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of jade. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):483–545, May 1998.

[181] R. L. Rivest, A. Shamir, , and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM (CACM)*, 21(2):120–126, Feb. 1978.

[182] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning. Saving portable computer battery power through remote process execution. *Mobile Computing and Communications Review (MC2R)*, 2(1):19–26, Jan. 1998.

[183] D. Saha, S. Sahu, and A. Shaikh. A service platform for on-line games. In *Proceedings of the 2nd workshop on Network and System Support for Games (Netgames)*, pages 180–184, Redwood City, California, May 2003.

[184] R. Sailer, L. van Doorn, and J. P. Ward. The role of TPM in enterprise security. Technical Report RC23363(W0410-029), IBM Research, October 2004.

[185] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.

[186] T. Salonidis, P. Bhagwat, and L. Tassiulas. Proximity awareness and fast connection establishment in bluetooth. In *Proceedings of the 1st ACM International Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc)*, pages 141–142, Boston, MA, 2000.

[187] Salutation architecture: Overview. `http://www.salutation.org/whitepaper/originalwp.pdf`, 1998.

[188] M. Satyanarayanan. *RPC2 User Guide and Reference Manual*. School of Computer Science, Carnegie Mellon University, Oct. 1991.

[189] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing (PODC)*, Philadelphia, PA, May 1996.

[190] M. Satyanarayanan. The evolution of coda. *ACM Transactions on Computer Systems (TOCS)*, 20(2):85–124, May 2002.

[191] H. Schneiderman and T. Kanade. A statistical approach to 3d object detection applied to faces and cars. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 746–751, Hilton Head Island, South Carolina, June 2000.

[192] J. Schulenburg. GOCR source code and online documentation. `http://jocr.sourceforge.net/`, Feb. 2004. (Version 0.39).

[193] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, United Kingdom, Oct. 2005.

[194] P. K. Sharma, J. P. Loyall, G. T. Heineman, R. E. Schantz, R. Shapiro, and G. Duzan. Component-based dynamic qos adaptations in distributed real-time and embedded systems. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, Agia Napa, Cyprus, Oct. 2004.

[195] S. Shearin and H. Lieberman. Intelligent profiling by example. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*, Santa Fe, NM, Jan. 2001.

[196] B. Shneiderman. Empirical studies of programmers: The territory, paths, and destinations. In *Proceedings of First Workshop on Empirical Studies of Programmers*, Alexandria, VA, Jan 1996.

[197] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings of the IEEE/RSJ Conference on Intelligent Robotics and Systems (IROS)*, Victoria, Canada, October 1998.

[198] A. Smailagic and D. P. Siewiorek. Application design for wearable and context-aware computers. *IEEE Pervasive Computing*, 1(4):20–29, October-December 2002.

[199] J. P. Sousa. *Scaling Task Management in Space and Time: Reducing User Overhead in Ubiquitous-Computing Environments*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 2005.

[200] R. Staehli, J. Walpole, and D. Maier. Quality of service specifications for multimedia presentations. *Multimedia Systems*, 3(5–6):251–263, 1995.

[201] J. A. Stankovic, S. H. Son, and J. Liebeherr. Beehive: Global multimedia database support for dependable, real-time applications. In *Proceedings of the Second International Workshop on Real-Time Databases (RTDB)*, Burlington, VT, Sept. 1997.

[202] Y.-Y. Su and J. Flinn. Slingshot: Deploying stateful services in wireless hotspots. In *Proceedings of the 3rd Annual Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, CA, Mar. 2005.

[203] Sun Microsystems, Mountain View, CA. *rpcgen Programming Guide*, 1987.

[204] Sun Microsystems Inc. *Remote Method Invocation Specification*, 1996.

[205] V. Sunderam. The pvm system: Status, trends, and directions. In *Proceedings of the Third European PVM Conference on Parallel Virtual Machine (EuroPVM)*, Munich, Germany, Oct. 1996.

[206] SWIG. *Simplified Wrapper and Interface Generator*, Mar. 2006. `http://www.swig.org/`, Version 1.3.29.

[207] A. S. Tanenbaum and S. J. Mullender. An overview of the amoeba distributed operating system. *SIGOPS Operating Systems Review (OSR)*, 15(3):51–64, July 1981.

[208] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, and S. J. Mullender. Experiences with the amoeba distributed operating system. *Communications of the ACM (CACM)*, 33(12):46–63, Dec. 1990.

[209] D. J. Taylor, D. E. Morgan, and J. P. Black. Redundancy in data structures: Improving software fault tolerance. *IEEE Transactions on Software Engineering*, 6(6):585–594, Nov. 1980.

[210] TechSmith Corporation. *Camtasia Studio*, June 2004. http://www.techsmith.com/.

[211] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, CO, Dec. 1995.

[212] L. G. Terveen and L. T. Murray. Helping users program their personal agents. In *Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*, Vancouver, Canada, Apr. 1996.

[213] D. Thain and M. Livny. Parrot: Transparent user-level middleware for data-intensive computing. In *Workshop on Adaptive Grid Middleware*, New Orleans, LA, Sept. 2003.

[214] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 17:323–356, February-April 2005.

[215] The MPI Forum. Mpi: A message passing interface. In *Proceedings of the Supercomputing Conference*, Portland, OR, Nov. 1993.

[216] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the v-system. In *Proceedings of the 10th Symposium on Operating System Principles (SOSP)*, Orcas Island, WA, Dec. 1985.

[217] Trusted Computing Group. *Trusted Platform Module Main Specification, Version 1.2, Part 1: Design Principles, Part 2: TPM Structures, Part 3: Commands*, Oct. 2003. https://www.trustedcomputinggroup.org.

[218] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling. Open grid services infrastructure (ogsi) version 1.0. http://www.globus.org/alliance/publications/papers/Final_OGSI_Specifica%tion_V1.0.pdf, June 2003.

[219] G. van Rossum. Ail - a class-oriented rpc stub generator for amoeba. *Lecture Notes In Computer Science (LNCS)*, 433:13–21, 1990.

[220] B. Vandalore, W. chi Feng, R. Jain, and S. Fahmy. A survey of application layer techniques for adaptive streaming of multimedia. *Real-Time Imaging*, 7(3):221–235, June 2001.

[221] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications*, 35(2):46–55, Feb. 1997.

[222] J. Waldo. The jini architecture for network-centric computing. *Communications of the ACM (CACM)*, 42(7):76–82, July 1999.

[223] J. Walko. *Handset sales reach new high in 2004*. EE Times, Jan. 2005. `http://www.eetimes.com/showArticle.jhtml;?articleID=59100009`.

[224] The Walkthru Project. GLVU *source code and online documentation*, Feb. 2002. `http://www.cs.unc.edu/~walk/software/glvu/` (Accessed on July 23 2002).

[225] G. Wasson, N. Beekwilder, M. Morgan, and M. Humphrey. Ogsi.net: Ogsi-compliance on the .net framework. In *Proceedings of the Fourth IEEE/ACM International Symposium on Cluster Computing and the Grid (ccGrid)*, Chicago, IL, Apr. 2004.

[226] G. Wearden. *Nokia: 2 billion cell phone users by 2006*. CNET news.com, Dec. 2004. `http://news.com.com/Nokia+2+billion+cell+phone+users+by+2006/2100-1039_%3-5485543.html`.

[227] The Web3D Consortium. *VRML Functional Specification*, Dec. 2003. `http://http://www.web3d.org/x3d/vrml/`.

[228] A. Webster, S. Feiner, B. MacIntyre, W. Massie, and T. Krueger. Augmented reality in architectural construction, inspection and renovation. In *Proceedings of the ASCE Third Congress on Computing in Civil Engineering*, pages 913–919, Anaheim, CA, June 1996.

[229] Y.-H. Wei, A. D. Stoyenko, and G. S. Goldszmidt. The design of a stub generator for heterogeneous rpc systems. *Journal of Parallel and Distributed Computing*, 11(3):188–197, 1991.

[230] D. S. Weld. Recent advances in ai planning. *AI Magazine*, 20(2):93–123, 1999.

[231] D. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, Kiawah Island, SC, Dec. 1999.

[232] A. Willig, M. Kubisch, C. Hoene, and A. Wolisz. Measurements of a wireless link in an industrial environment using an ieee 802.11-compliant physical layer. *IEEE Transactions on Industrial Electronics*, 43(6):1265–1282, Dec. 2002.

[233] A. J. Willmott. RADIATOR source code and online documentation. `http://www.cs.cmu.edu/~ajw/software/`, Oct. 1999. (Accessed on July 23 2002).

[234] The World Wide Web Consortium (W3C). *Mobile Code*, Oct. 2003. `http://www.w3.org/MobileCode/`.

[235] P. Yao and D. Durant. Microsoft mobile internet toolkit lets your web application target any device anywhere. *MSDN Magazine*, 17(11), Nov. 2001.

[236] M. Yarvis, K. Papagiannaki, and W. S. Conner. Characterization of 802.11 wireless networks in the home. In *Proceedings of the First Workshop on Wireless Network Measurements (WiNMee)*, Trentino, Italy, Apr. 2005.

[237] M. Yarvis, P. Reiher, K. Eustice, and G. J. Popek. Conductor: Enabling distributed adaptation. Technical Report CSD-TR-010025, University of California, Los Angeles, Los Angeles, CA, June 2001.

[238] M. Yarvis, P. Reiher, and G. J. Popek. Conductor: A framework for distributed adaptation. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS)*, Rio Rico, AZ, Mar. 1999.

[239] E. R. Zayas. Attacking the process migration bottleneck. In *Proceedings of the 11th ACM Symposium on Operating System Principles (SOSP)*, Austin, TX, Nov. 1987.

[240] M. J. Zieniewicz, D. C. Johnson, D. C. Wong, and J. D. Flatt. The evolution of army wearable computers. *IEEE Pervasive Computing*, 1(4):30–40, October-December 2002.

Appendix A

Chroma: Runtime Support for Cyber Foraging

In this Appendix, I describe the goals and implementation of Chroma. Chroma is written in C as a Linux user space process. It consists of many different components as shown in Figure A.1. These components provide resource measurement, prediction, fidelity, tactic selection, and remote execution functionality. This detailed discussion of Chroma is in an Appendix as only the tactic selection solver component was specifically developed for this thesis. The other components were originally developed by other researchers and reused. However, for completeness, I still describe each component in detail in the subsequent sections. Where appropriate, I make clear which is my work and which is work that I am reusing.

A.1 Design Goals

Chroma was designed to achieve three major goals. These are:

1. **Seamless from user perspective**: The user should be oblivious to the decisions being made by Chroma and the actual execution of those decisions.

2. **Effectiveness** : Chroma should employ close to optimal strategies for remote execution under all resource conditions. An application developer should not be tempted to hand tune.

3. **Minimal burden on application writers**: I want Chroma to be an easy system for application writers to use.

I achieved these goals as follows:

Seamless from user perspective : I achieved this goal by making Chroma completely automatic from the perspective of the application user. Chroma was designed to support interactive applications which already demand user attention due to their interactive nature.

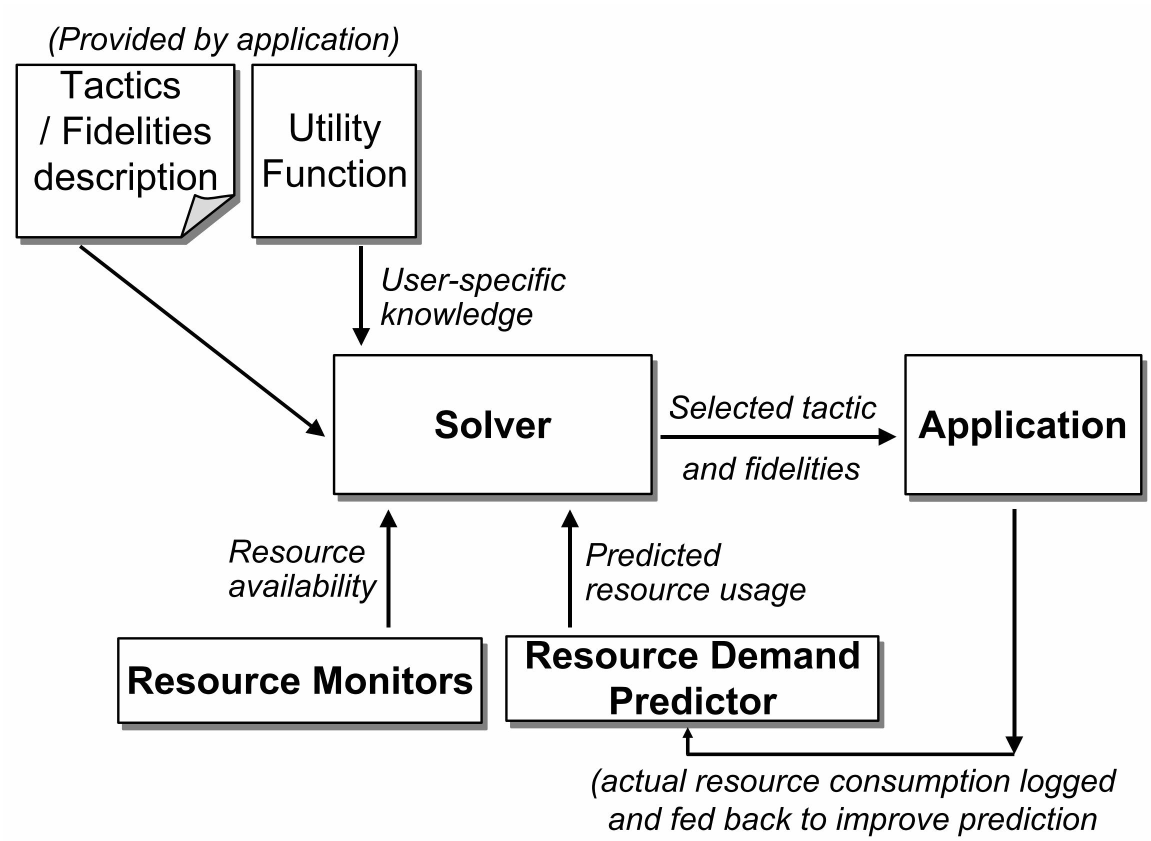


Figure A.1: Main Components of Chroma

It is thus vital that Chroma distract the users as little as possible. I achieve this by allowing the user to specify high-level preferences in advance to Chroma. With these preferences, Chroma will decide at runtime how and where to execute applications. The user is oblivious to these decisions in normal use of the system.

Effectiveness : The use of tactics allows Chroma to dynamically pick an optimal application partitioning. Chroma couples this with predictive resource management to achieve excellent application performance in dynamic mobile environments

Minimal Burden : This was accomplished using the RapidRe process described earlier in this thesis.

A.2 Chroma Overview

Chroma has full support for concurrent application execution. Chroma's goal is to, at runtime, decide the following two application settings:

Application Fidelity Variable Settings : These are application-specific variables that affect the quality of the application. The tradeoff is that reducing the application quality results in lower resource usage. Hence, in many cases, it is necessary to use a lower quality setting to satisfy the available resources. Examples of fidelity variables include the resolution setting for graphics applications, the data files used by language translators, and the size of the language model for speech recognizers. Chapter 2.5.1.2 showed how developers can specify the fidelity variables for their applications.

Tactic Plan : A tactic plan is a particular tactic, from the set of possible tactics, where a specific server has been selected for every RPC in the tactic. For example, a possible tactic plan is choosing the do_simple tactic from Figure 2.4 and deciding that RPC $step\_1$ will run on server X and RPC $step\_3$ will run on server Y. A tactic plan thus contains all the necessary information for an application to execute the chosen tactic. Chapter 2.5.1.3 showed how developers can specify the tactics for their applications.

In the rest of this appendix, I use the term *runtime setting* to refer to a combination of a tactic plan and the precise settings for the fidelity variables of an application. Operation settings thus represent the complete set of information needed by applications to successfully perform the operation.

To decide the optimal runtime setting, Chroma needs to know three things; Namely, the current resource availability, the predicted resource usage of every tactic plan and fidelity combination, and knowledge of the user's goals. Given these three things, Chroma will then have to decide the best tactic plan and fidelity settings and return the chosen settings to the application. Chroma uses a solver component, built for this thesis, that iterates through all possible tactic and fidelity settings and selects the optimal runtime setting. The solver is explained in more detail in Appendix A.6. The application then performs the operation using the determined runtime setting. The actual resource usage of the operation is measured by Chroma and used to update the resource usage predictors.

A.3 Resource Measurers

The current implementation of Chroma uses multiple resource measurers to determine the current resource availability. These measurers were developed by other researchers for the Odyssey [160, 76, 152] system and are not part of this dissertation's contributions. My work was to port the measurers to Chroma and update them to work under the Linux 2.6 kernel – the base measuring algorithms remained the same. The key resource measurers needed for this thesis were the available bandwidth and latency of operation measurers (both developed by Noble [160]), and the available system memory and available CPU cycles measurers (both developed by Narayanan [151]). The details of these measurers are provided below in the interest of completeness.

A.3.1 CPU

Chroma uses the $/proc$ filesystem of Linux to measure the current availability of both these resources. For CPU, Chroma reads the value of $/proc/loadavg$ every 0.5 seconds to discover the current system load. This load is then smoothed, to eliminate the effect of transient load spikes, using a weighted average of the measurements for the last one minute. This average load provides a realistic approximation of the expected system load. Chroma then converts this average load into expected available cycles using the following formula:

$$available\_cpu\_cycles \ = \ 1 \ - \ average\_load \ * \ machine\_clock\_speed,$$

The machine clock speed is obtained by reading the value of $/proc/cpuinfo$.

A.3.2 Memory

Chroma uses different algorithms to discover the available memory. On Linux 2.2 kernels, the available memory is given by the following formula:

$$MemAvail = MemFree + Buffers - 1024,$$

where $MemFree$ is the available free memory and $Buffers$ is the number of raw 4 Kilobyte disk blocks used for relatively temporary storage. These values are read from $/proc/meminfo$. The value of $Buffers$ is decreased, based on experiments performed by Narayanan, by 1024 to ensure that the system always has at least 4 Megabytes of free memory.

Linux 2.4 kernels have a more sophisticated virtual memory system and the available memory measurer uses the following formula instead:

$$MemAvail \ = \ MemFree \ + \ Inact\_clean,$$

where, as before, each of the values are read from $/proc/meminfo$. $MemFree$ is the available free memory (that is not being used by any application), $Inact\_clean$ is the number of easily freeable memory pages. These are memory pages that have not been accessed recently. Determining the available memory on Linux 2.6 kernels is similar to 2.4 kernels except that the $Inact\_clean$ variable is called $Inactive$ in $/proc/meminfo$. The memory measurer is purposely designed to only add memory pages that are safe to use and can be used immediately. For example, it does not include active pages or inactive yet dirty pages into these measurements. This ensures that Chroma does not try to use memory that it shouldn't or which is not immediately usable.

A.3.3 Available Wireless Bandwidth

Chroma uses passive network monitoring to measure the available wireless bandwidth. As such, Chroma observes normal network traffic to discover the available bandwidth. This is in contrast to active network measurement techniques such as BFind [9], Cartouche [105], Pathchar [117], and Pathneck [111] that add traffic to the network to discover the available bandwidth. These active schemes have the potential to provide better measurements but they can also overstress a network with large amounts of measurement traffic.

Chroma performs its passive measurements as follows; the low-level communication package used by Chroma, RPC2 [188], is able to measure the observed bandwidth and round trip times of any initiated network connection. Hence, whenever Chroma communicates with a remote server, for example, when checking if the server is running (done by the service discovery component) or when performing an operation, it timestamps this RPC2 information and writes it to a log.

Whenever Chroma needs an available bandwidth measurement, it reads this log and extracts the most recent bandwidth measurements (this is set to the last 30s of measurements). Chroma then performs a weighted average of these values and uses the final value as the expected available bandwidth. The weighting is performed to smooth out transient bandwidth fluctuations. The assumption is that the available bandwidth will not change drastically in such a short time period.

A.4 Resource Predictors

For a given operation, Chroma needs to be able to predict the resource usage and execution latency of every possible runtime setting. For this dissertation, I use the resource demand prediction work by Narayanan [151] to provide this information. To summarize this work, it uses history-based prediction as the main mechanism to do prediction. The key idea here is that the resource usage and predicted execution latency of any possible runtime setting, for a given set of application inputs, can be predicted from its recent resource usage. To index the prediction table, the predictors use the runtime settings of particular application variables called *parameters* that indicate the amount of resources the current operation will require. Parameter values are fixed and should have the property where a larger parameter value indicates more resource usage. For most applications, the size of the inputs to the operation make great parameters. Chapter 2.5.1.2 how parameters are specified using Vivendi.

To bootstrap the predictors and avoid a lengthy convergence phase, Chroma performs off-line logging where the resource usage of a large number of runtime settings, for a given set of training inputs, is measured and stored. During normal operation, the predictors are updated using online monitoring and machine learning to improve accuracy. A more thorough description and evaluation of this resource demand predictor is available in Narayanan's dissertation [151] or from the paper cited above.

A.5 Obtaining User Preferences and Goals

To effectively match resource demand to resource availability, Chroma needs to trade off resources for fidelity. How to perform this tradeoff is frequently context sensitive and thus dynamic. For instance, would the user of a language translator prefer accurate translations or snappy response times? Should an application running on a mobile device use power-saving modes to preserve battery charge, or should it use resources liberally in order to complete the user's task before he or she needs to board their plane?

I provide Chroma with these user-specific resource tradeoffs in the form of *utility functions*. A utility function is a user-specific precise mathematical function that quantifies the tradeoff between two or more attributes. The utility function takes a particular runtime setting, factors in the available resources and the predicted resource usage of that runtime setting, and returns a value in the range $[0, 1]$ (inclusive). This value represents how happy the user would be with this particular runtime setting – the higher the value, the happier the user would be. The solver then iterates through all possible runtime settings and picks the one that maximizes the utility function.

Unfortunately, obtaining utility functions that match the current user preferences is not a trivial task. An effective and non-intrusive system would have to infer the correct user preferences from mostly passive monitoring of user activities. However, the system will also have to provide the right interfaces for users to directly convey their desires, in cases where the monitoring is ineffective, to the system. However, even with this passive monitoring and direct inputs, frequently, the user preferences will only be specified in very broad general terms. For example, the preference might be "Conserve as much energy as possible", or "I want to use as little bandwidth as possible". The system will then have to convert these preferences into specific and detailed utility functions such as "The total energy consumption for applications A, B, and C, over a 10 minute period, must be less than 200 Joules. The quality and latency of these application does not matter."

Fortunately, there are already existing systems designed solely to address the issue of accurately obtaining detailed user preferences. One such system is Prism [199] and Chroma has been modified to work with it. This integration is one of the key new features of Chroma that differentiate it from the previous Odyssey system.

A.5.1 Integration with Prism

Chroma was first modified to allow asynchronous communication from Prism. This eliminated the need to tightly synchronize Chroma and Prism. Prism communicates with Chroma using self-describing XML messages. In particular, Prism uses 4 different types of messages, $activate$ (Figure A.2), $setState$ (Figure A.3), $getResources$ (Figure A.4), and $deactivate$ (Figure A.5) to communicate with Chroma. These messages all share common features. They all identify the task (using the $taskId$ tag in each message) the message pertains to and the location of the requesting Prism process (using the $location$ and $port$ tags in the message). A task refers to a user-specific grouping of application services to satisfy some larger goal. For example, a "write my dissertation" task may involve a text

```
<activate location="chroma-15" bind="5700" taskId="1" >
    <service id="1" type="translate" application="panlite">
    <-- constraints are for resources, which are tagged by name -->
         <constraints>
            <resource name="cpu" type="integer">
               <numEstimate average="10" variance="10" unit="%"/>
            </resource>
            <resource name="bandwidth" type="integer">
               <numEstimate average="50" variance="50" unit="Kbps"/>
            </resource>
         </constraints>
    </service>
</activate>
```

This message is used by Prism to tell Chroma which service (translate in this case) and application (panlite in this case) is required by the user. It also specifies the maximum resources that can be used by that application (CPU of 10% and bandwidth of 50 Kbps). Chroma is responsible for finding the optimal runtime setting for the application given these constraints.

Figure A.2: Example Prism Activate Message

editor, a graphics editor, and a compiler. Finally, each message has a unique $service\ id$ indicating which particular service in the task this message is pertaining to.

These messages are used as follows: When Prism first connect to Chroma, and when Prism detects that the user preferences have changed, Prism will send a $setState$ message to Chroma. This message will contain the utility function that encodes the current user preferences. When Prism detects that the user requires a particular application, it will send Chroma an $activate$ message containing the name of the application. This allows Chroma to setup any state needed to handle the application. The $activate$ message will also contain any resource constraints that this application should not exceed. When the application is no longer needed by the user, Prism will send Chroma a $deactivate$ message. Finally, Prism will periodically query Chroma, using the $getResources$ message, to discover the current resource availability. Chroma returns the current resource availability to Prism using a $ResourceSnapshot$ (Figure A.6) message. This resource availability is distilled by Prism and then presented to the user. This allows the user to provide realistic preferences.

The integration of Chroma with Prism proves to be useful to both systems. Chroma is able to obtain realistic user preferences from Prism and Prism is able to obtain accurate resource availability measurements from Chroma. In addition, Chroma provides the adaptation functionality needed to achieve the user's application goals (validated in Chapter 5).

```
<setState taskId="1" location="chroma-15" bind="5700" profile="fast">
   <service id="1" type="translate">
      <utility combine="product">
      <-- the utility functions for this service --->
         <QoSdimension name="latency" type="float">
            <function type="sigmoid" weight="1">
                <thresholds unit="second" good="1.5" bad="7"/>
            </function>
         </QoSdimension>
         <QoSdimension name="fidelity" type="float">
            <function type="sigmoid" weight="0.5">
               <thresholds unit="float" good="1" bad="0"/>
            </function>
         </QoSdimension>
      </utility>
      <-- resource constraints that Chroma should not exceed -->
      <-- these are tagged by name with exact units          -->
      <constraints>
          <resource name="cpu" type="integer">
               <numEstimate average="50" variance="10" unit="%"/>
          </resource>
          <resource name="bandwidth" type="integer">
               <numEstimate average="150" variance="50" unit="Kbps"/>
          </resource>
      </constraints>
   </service>
</setState>
```

Prism uses this message to provide Chroma with the utility function for a particular service (translate in this case). In this example, the utility function states that latency values under 1.5s are excellent (given a utility value of 1) and that any value above 7s is unacceptable (utility value of 0). Any latency value in between receives a proportional utility value. The function also states that any fidelity value of 1 is excellent (utility score of 1) and any value of 0 is unacceptable (score of 0). The final utility is calculated by first multiplying each individual function components (latency and fidelity in this case) by its weight and then by multiplying the weighted values together. In this case, latency has a weight of 1 while fidelity only has a weight of 0.5. The final utility, in this message, is thus $(utility_{latency} * 1.0) * (utility_{fidelity} * 0.5)$.

This message is also used to update the resource constraints for the service. In this case, translate can use a maximum of 50% of the CPU and 150 Kbps of bandwidth.

Figure A.3: Example Prism setState Message

```
<getResources taskId="1" location="chroma-15" bind="5700">
   <service id="1" type="translate"/>
</getResources>
```

Prism uses this message to determine, from Chroma, the available resources for a particular service (translate in this case). Chroma will reply with a ResourceSnapshot message (Figure A.6).

Figure A.4: Example Prism getResources Message

```
<deactivate taskId="1" location="chroma-15" bind="5700">
   <service id="1" type="translate" application="panlite"/>
</deactivate>
```

This message is used by Prism to tell Chroma that a particular service (translate in this case) and application (panlite in this case) is not needed by the user anymore.

Figure A.5: Example Prism Deactivate Message

A.6 Solver

At the heart of Chroma is the *solver* component. I used the Odyssey solver as a baseline, fixed some bugs, and added tactic support to it.

The solver is given the list of tactics and fidelity variables for the operation and constructs a solution space of all tactic-fidelity variable combinations. The goodness of any specific point in this space is computed using the utility functions described above. The solver then exhaustively searches this space for the optimal runtime setting. I.e., the point that maximizes the given utility function. This exhaustive search is feasible as the space is relatively small because there are only a few tactics and fidelity variables for most applications.

For the solver to work properly, it requires accurate resource supply measurements, resource demand predictions for all possible runtime setting permutations of the current application operation, and the utility function expressing the user's preferences. In addition to these three things, the solver requires two additional pieces of information. First, it requires the list of tactics and fidelity variables for the current application. This is provided by an application-specific configuration file that is read by Chroma when the application registers. This configuration file, shown in Figure A.11, is automatically generated from the information in the developer specified tactics file.

Next, the solver needs to know how choosing different tactics and fidelity variable settings affects the application quality. For example, the solver has to know that tactic a has twice the expected quality of tactic b. Without this information, the solver will not

```
<ResourceSnapshot taskId="1" location="localhost" bind="5700">
   <service id="1" type="translate">
      <-- resource constraints that Chroma should not exceed -->
      <-- these are tagged by name with exact units          -->
      <constraints>
          <resource name="cpu" type="integer">
               <numEstimate value="50" variance="10" unit="%"/>
          </resource>
          <resource name="bandwidth" type="integer">
               <numEstimate average="450" variance="50" unit="Kbps"/>
          </resource>
          <resource name="memory" type="integer">
               <numEstimate average="213" variance="14" unit="MB"/>
          </resource>
      </constraints>
   </service>
</ResourceSnapshot>
```

This message is used by Chroma to inform Prism of the current resources available for a particular service (translate in this case). In the above message, the resources available are 50% CPU, 450 Kbps network bandwidth and 213 MB of memory.

Figure A.6: Example Prism ResourceSnapshot Message

be able to pick the optimal runtime setting that maximizes the utility function. Chroma uses application-specific hint modules to provide this information. These hint modules are short C routines that list the relative fidelities of different tactics and fidelity variable settings. These routines are compiled into a library and dynamically loaded by Chroma as necessary.

In the normal case, the solver solves for an optimal runtime setting using the following high-level loop:

1. pick a possible application runtime setting
2. calculate the expected resource usage of this runtime setting using the resource demand predictor
3. calculate the goodness of this runtime setting using the utility function (which uses the hint module to calculate fidelity levels)
4. repeat for all possible application runtime settings
5. pick the runtime setting that had the highest goodness value

Algorithm Prototype: $Chroma\_Solver(Tactic, RPC, Server, params, fid\_vars)$**;**

```
chosen_tactic = 0, runtime_setting = ∅;

for (i = 0; i < num_tactics; i++) do    // For every application tactic
    Utility_Tactic_i = 0.0;

    for (j = 0; j < tactic[i].num_rpcs; j++) do    // For every RPC in the tactic
        Utility_RPC_j = 0.0, Server_RPC_j = 0, fid_vals_RPC_j = ∅;

        // For every available server (includes local machine)
        for (k = 0; k < num_avail_servers; k++) do
            Utility_k = 0.0 and i = 0;
            fid_vals ← Random settings for all fidelity variables

            while (Utility < 1.0 and i++ ≠ 1000) do
                Utility_k ← Utility_func(Tactic[i], RPC[j], Server[k], params, fid_vals);

                if (Utility_k > Utility_RPC_j) then
                    Utility_RPC_j ← Utility_k;
                    Server_RPC_j ← k;
                    fid_vals_RPC_j ← fid_vals;
                end if    // Check if current settings have higher utility
                Use gradient descent to pick better fidelity variable settings
            end while    // Fidelity Variables

        end for    // For every available server
    end for    // For every RPC in the tactic
    Utility_Tactic_i = Σ_{j=0}^{tactic[i].num_rpcs} Utility_RPC_j    // Sum all RPC utilities
    // The server and fidelity variable settings for each RPC are also saved
end for    // For every application tactics

chosen_tactic ← max(Utility_Tactic_i);    // Tactic with the highest utility
for (i = 0; i < tactic[chosen_tactic].num_rpcs; i++) do
    runtime_setting ← Concat(RPC_i_server, RPC_i_fid_vals);
end for    // Create runtime setting for the chosen tactic

return chosen_tactic and runtime_setting;
```

Figure A.7: The Chroma Solver Algorithm

The full algorithm representing the high-level intuition presented above is shown in Figure A.7. It performs an exhaustive search to determine the optimal runtime setting. It starts by searching through all possible application tactics (Line 2). The algorithm will then search through every RPC that makes up that tactic (Line 4). For each of these RPCs, the algorithm will search through every possible available server (Line 7).

The heart of the solver is between Lines 10 and 18. In these lines, the solver searches for an optimal fidelity variable setting for this tactic, RPC, and server setting. The solver first picks a random value for each of the fidelity variables. It then uses gradient descent (Line 17; the actual algorithm for this descent is not shown) to pick new values for fidelity variables. The goal is to maximize the utility function (Line 11). This gradient descent search continues until either a setting that maximizes the utility (the maximum utility value is 1.0) is found or 1000 iterations have been performed. This loop saves the server and fidelity variable settings that achieve the highest utility for the current RPC.

After the optimal server and fidelity variable settings for each RPC has been determined, the algorithm (Line 21 will calculate the overall utility for the tactic. After each tactic has been processed, the tactic with the highest overall utility is chosen as the tactic to use (Line 24). The algorithm will then concatenate (Line 25) the individual servers and fidelity variable settings for each RPC in the tactic to create the final runtime setting.

The solver is thus able to pick an optimal fidelity variable setting for every RPC in the chosen tactic. However, my application model does not support this; applications assume that the same chosen fidelity variable settings are used for the entire tactic. Hence, the solver's output is adjusted such that only the fidelity variable settings for the first RPC in the chosen tactic are returned. Even though it is not optimal, this heuristic still proves to be effective in practice.

A.6.1 Data Decomposition Solver

When performing data decomposition, Chroma uses a different solver. The main difference is that this solver solves for a set of servers that are good enough to be used for the data decomposition. This is unlike the normal solver that solves for the optimal server settings for the given operation.

The algorithm for this solver is shown in Figure A.8. This solver is fairly simple and has a few limitations. In particular, the solver can only handle applications that have only a single tactic (that is decomposable). In addition, the solver will set all fidelity variables to their maximum values to simplify the task of decomposing the tactic. Otherwise, the decomposition routines would have to manage a situation where different sub results could have different qualities.

Given these limitations, the solver will return a set of servers that can be used for the decomposition. To determine if a server is acceptable, the solver checks if the utility (as shown in Line 5 of Figure A.8) of using the servers exceeds a given threshold. In my current implementation, the threshold is arbitrarily set to 0.70. In the future, I hope to obtain this threshold dynamically from the user (possibly via Prism).

Algorithm Prototype: $Chroma\_Decomp\_Solver(Tactic, RPC, Server, params, fid\_vars)$**;**

$chosen\_tactic = 0$, $server\_list = \emptyset$;
$fid\_vals \leftarrow$ maximum values for all fidelity variables

// For every available server (includes local machine)
for ($k = 0$; $k < num\_avail\_servers$; k++) **do**
    if $(Utility\_func(Tactic[0], RPC[0], Server[k], params, fid\_vals) \ > \ thresh)$; **then**
        $server\_list \leftarrow Append(Server[k])$; *// Add server to server_list*
    end if *// Check if server is acceptable*
end for *// For every available server*

return $chosen\_tactic$, fid_vals, and $server\_list$;

Figure A.8: The Chroma Data Decomposition Solver Algorithm

A.7 Remote Execution

In the previous sections, I described the components of Chroma that allow it to determine the runtime settings of an application at runtime. With these settings, the application can use the remote execution functionality of Chroma to perform the operation using the chosen tactic plan and fidelity variable settings.

In this section, I describe the components that comprise the remote execution functionality of Chroma. These components collectively allow Chroma to leverage the capabilities of remote servers and are vital in facilitating cyber foraging. To successfully use remote execution, Chroma needs to discover remote machines, start application servers on these machines, establish trust with these machines, and provide mechanisms to use these machines to execute operations. In this dissertation, I leverage prior research for the first three issues. However, for completeness, I discuss each of these components in more detail in the next few sections.

A.7.1 Service Discovery

Before Chroma can use remote servers, it must first discover them. Chroma has two different mechanisms for service discovery. The first mechanism is a very simple simple service discovery interface developed by Flinn [75]. This interface uses a static text file containing a list of possible remote servers to perform discovery. After reading this list, Chroma will dynamically query each server in this list to discover if that server can be used for remote execution. When using this simple interface, Chroma assumes that all available servers can be used for any applications. I.e., it is not able to determine that server a can only be used for application x and not application y. The text file, shown in Figure A.9, also speci-

```
d 1
l localhost
r chroma-1.aura.cs.cmu.edu
r chroma-2.aura.cs.cmu.edu
r chroma-3.aura.cs.cmu.edu
r chroma-4.aura.cs.cmu.edu
r chroma-5.aura.cs.cmu.edu
r chroma-6.aura.cs.cmu.edu
r chroma-7.aura.cs.cmu.edu
r chroma-8.aura.cs.cmu.edu
```

The first line (that starts with a "d") specifies the interval (in seconds) at which Chroma should query each server in the list. In this case, the query interval is set to 1s. The rest of the file contains the list of possible servers. The local server is explicitly specified (as "l localhost") as there may be cases where users may not want Chroma to perform any operations locally.

Figure A.9: Service Discovery Config File

fies the interval at which Chroma should query each server. This simple solution provides a nice tradeoff between a completely static and a completely dynamic service discovery mechanism.

Chroma can also use a more powerful service discovery interface called *SDF* (short for service discovery framework). This interface was developed by three masters students for the class project component of a mobile computing graduate-level class at Carnegie Mellon University. The students were Karthik Belur, Abhijit Deshmukh, and Mark Pariente. I supervised them for the project and played a key role in guiding the overall design of SDF.

SDF is a fully dynamic system that can detect new services in the environment. Chroma can use it to query for servers in the environment that specifically support the required applications. To provide this functionality, SDF provides a client and server framework component. Application servers register with the server framework component (running either on the application server or somewhere else) and tell the component what service they can provide and what specific characteristics their service has. For example, a print server can tell the component that it has a laser printer that supports letter size pages and can print at 600 dpi. This information is specified as a SQL string and is stored by the server component in a MYSQL database.

Client applications, such as Chroma, use the client framework component to make service discovery calls. The client application specifies, using a SQL string, the exact service that is looking for. The client framework component then broadcasts this query on a well-known port. This port is monitored by the server framework components in the environment and they will respond with any positive matches in their database. The client framework component accumulates these results and returns them to the client application when requested.

There are also other service discovery protocols (SDP) that Chroma can possibly use. This includes industry standard protocols such as Bluetooth Proximity Detection [103, 186], Jini [222], Service Location Protocol [99, 100, 101, 171], and UPnP [119] as well as more proprietary solutions such as the Intentional Naming System [6], Salutation [187], and the Secure Service Discovery Protocol [53].

A.7.2 Instantiating Servers

Discovering remote servers that can be used is just the first step in the remote execution process. The second step involves starting application servers on these servers that can respond to queries by application clients running on mobile devices. Similar to the service discovery problem, this problem is also not part of the scope of this dissertation. Instead, Chroma assumes that servers are manually started a-priori on all possible servers.

To make it easier to manually start a number of application servers, Chroma uses a simple mechanism, developed by Flinn, called $rmexec$. $rmexec$ reads a config file on startup and starts all the servers specified in the config file. It also has been modified to work with the SDF service discovery framework. In particular, $rmexec$ tells the server framework component of SDF exactly which application servers it has started.

There are more dynamic solutions that can be used by Chroma. These include prior research by Shaikh et al. [183] that focused on dynamically starting servers for distributed game infrastructures. Other possible solutions are object migration techniques as used in Emerald [123] and process migration techniques as used in V [216], Amoeba [150, 207, 208], Charlotte [16], Condor [214], Demos [174], Mach [239], Sprite [60, 61, 166] and Zap [165]. Nuttal [161] and Milojicic et al. [146] provide overviews of the various migration techniques that have been developed. It may be also be possible to use code migration systems such as ANTS [231] or language-supported code migration functionality [234] to dynamically execute code on remote servers. Finally, researchers such as Garlan et al. [83], Gribble et al. [94], Huang and Steenkiste [112, 113], and Raman et al. [175] have looked at the problem of service composition or the building of useful applications from smaller components already available in the environment. This may prove to be a viable way to build application servers in certain environments.

Recently, there has been work on instantiating servers specifically for cyber foraging. This includes work by Goyal and Carter [93] as well as the Slingshot system, that uses virtual machines to start application servers on remote servers, by Su and Flinn [202].

A.7.3 Security of Using Remote Servers

The third part involved in using remote servers is establishing trust with those servers. However, establishing this trust is not a trivial task. First, we have to verify the identity of the servers to the mobile clients and vice versa. Second, we have to prevent unauthorized entities from eavesdropping on the communications between the servers and clients. Third, we have to verify that the servers are actually doing what they are supposed to be doing. In particular, clients should be able to verify that the servers are actually performing the correct computations and are not returning made up results. These three problems are known as the authentication, encryption, and integrity problems respectively.

In this dissertation, I differ to prior research for most of these problems as there are already a number of excellent solutions for these problems. However, I do present a possible solution for the integrity problem and present an analysis of my solution in Section 5.6.

A.7.3.1 Authentication and Encryption

The authentication problem involves the mobile client discovering the true identities of the remote servers and vice versa. This is vital to prevent mobile clients from unintentionally communicating with rogue servers masquerading as legitimate servers. After the identities of remote servers have been validated, communication between the mobile clients and the remote servers can commence. However, to be secure, it is necessary to ensure that these communications cannot be read by other entities. This requires encryption.

Fortunately, both authentication and encryption can be achieved existing well-tested mechanisms. For authentication, protocols such as Diffie-Hellman [59], Needham-Schroeder [156], and RSA [181] can be used. These protocols have been heavily validated [72] and shown to be resistant to common attacks. For encryption, IPSec [130] with secure encryption protocols such as Blowfish, Camellia, Digital Encryption Standard (DES), and the Advance Encryption Standard (AES) [72] can be used.

A.7.3.2 Integrity

After the secure communication has reached the authenticated remote server, the server can now perform computation on behalf of the mobile client. However, how can the mobile client ensure that the remote server is doing the right computations and returning the correct results?

One way to ensure this is to setup a completely trusted software environment on each remote server. Establishing trust in hardware is a major goal of the security community, especially the Trusted Computer Group [217]. The recent work on trusted platform modules at IBM [184, 185] is of particular relevance here.

Another solution is to embed information into the data sent to the remote server such that any tampering can be detected by the mobile client. The work by Chen and Morris on hardware assisted tamper-evident remote execution [44], Seshadri et. al on tamper-proof code execution without hardware assistance [193], and by Necula on proof carrying codes [155] are examples of this line of research.

There has also been a large amount of research in using cryptosystems [26, 52, 92, 12, 118] and random oracles [24] to provide integrity and accuracy of results. However, these solutions are complicated and it is unclear how they would map to a mobile remote execution domain.

The use of fault-tolerant computation [5, 64, 125, 209] has been proposed as a solution to errors caused by transient server failures. However, these methods assume that the failures are transient and are not malicious in nature. Masking malicious or Byzantine failures requires the use of multiple servers and distributed agreement protocols [41].

Using the solutions described above, it might be possible to solve the integrity problem. However, these solutions require large amounts of access to the remote servers, complicated new networking protocols, or application changes on the mobile client. For this dissertation, I ask a simpler question. Namely, "What is the simplest solution that can be used by a mobile client to detect misbehaving servers?"

My solution is to use a simple probabilistic verification scheme. In this scheme, the mobile client sends a remote server a known query (i.e., a query for which the mobile client already knows the answer). The mobile client can then verify the answer returned by the server. By repeating this verification step randomly multiple times, the mobile client can build a trust index for each server in the environment. This simple probabilistic scheme is useful as it can detect even Byzantine failures and requires no infrastructure support, new networking protocols, or application changes. However, because it is probabilistic, the mobile client can still end up using misbehaving servers. In Section 5.6, I analyse the detection probability of this scheme and present its runtime overheads.

A.7.4 Remote Execution Mechanisms

The final component involved is using remote servers is developing the mechanisms used to perform the remote execution. Because I am using coarse-grained remote execution, I use the RPC model of communication for this dissertation. However, there are still further questions to address.

Because Chroma uses tactics, it is possible that the chosen tactic may involve more than one RPC stage. For example, tactic do_simple in Figure 2.4 involves 2 RPC operations, $step\_1$ and $step\_3$, performed consecutively. However, even for this simple tactic, there are multiple ways to "control" the tactic. For example, control can be retained at the mobile client. In this scheme, the mobile client will first initiate RPC $step\_1$. It will then initiate RPC $step\_3$, after the results of RPC $step\_1$ have returned. This method of control is the easiest to implement and debug.

Alternatively, control can be delegated to another machine. For example, the mobile client delegates control to machine Y and sends Y the inputs for the tactic. Y then executes the tactic and returns the results to the mobile client. This delegated control method is useful when the outputs of the intermediate RPCs are large or when the mobile client needs to conserve battery power. However, this method of control is much harder to implement and debug as it requires fairly sophisticated signalling protocols.

For this dissertation, Chroma uses the centralized method of control where the mobile client initiates all RPC requests. Even though this method is not as efficient as the delegated control method in certain situations, I show in Section 5 that it is still able to achieve excellent performance.

API Command	Description
Registering the Application	
int **register_fidelity** (char *conf_file, int num_rpcs, int *op_id)	Registers app with Chroma. $conf\_file$ is the name of the config file (example shown in Figure A.11). num_rpcs is the number of app RPCs (each RPC is given a unique connection to make executing parallel RPCs easier). Chroma returns a unique op_id on successful registration.
void **close_chroma_sockets** (int num_rpcs)	Closes all connections (num_rpcs is the number of open connections) with Chroma.
Determining an Optimal Operation Setting / Operation Resource Logging	
int **begin_fidelity_op** (int op_id, int num_params, fid_param_val_t *params, int num_fidelities, fid_param_val_t *fidelities, int *runid, int *chosen_tactic)	This multi-argument API call tells Chroma to pick an optimal runtime setting for the app. op_id$is$the unique application identifier. $params$ and $fidelities$ are the list of parameters and fidelity variables. This call returns the chosen tactic ($chosen\_tactic$), a unique runtime id ($runid$) and the fidelity variable settings (in $fidelities$). Chroma internally stores the server selection for each RPC in the chosen tactic. Chroma starts logging the resource usage of the operation when this call is made.
int **end_fidelity_op** (int runin, int opid, failure_code failed)	Tells Chroma to stop logging the resource usage of this operation (denoted uniquely by $runid$ and $opid$). The app-perceived success of the RPC is also reported ($failed$).
Executing the Chosen Operation Setting	
int **do_remote_op** (int rpc_num, int op_id, int runid, remoteop_t* rop, void* in_data, int in_len, char* in_file, int in_file_flag, void* out_data, int* out_len, char* out_file, int out_file_flag)	Performs a remote RPC. rpc_num is the RPC to perform. op_id and $runid$ are the app and runtime ids. The input data is provided in in_data (buffer with length in in_len) and/or in_file (file name). the output data is received in out_data (length in out_len) and/or out_file (output file name). $\{in\|out\}\_file\_flag$ specify permissions on the input/output files. Chroma remembers which server was chosen for this RPC (can be changed using set_rpc_server)
int **do_local_op** (int rpc_num, int optype_id, int opid, remoteop_t* rop, void* in_data, int in_len, char* in_file, void* out_data, int* out_len, char* out_file)	Perform an RPC locally. The arguments are the same as do_remote_op. The only difference is that local RPCs don't need to specify file permissions. This local RPC call is provided for efficiency reasons.
int **set_rpc_server** (int op_id, int runid, int rpc_num, char* server)	Allows apps to change the server used by a specific RPC. rpc_num is the RPC to change and $server$ is the server to use.

This figure shows the API calls that are used by application client components to communicate with Chroma's client component ($viceroy$). These calls are for normal operation. In particular, the $begin\_fidelity\_op\_decomp$ call used for data decomposition is omitted.

Figure A.10: Chroma's Client API

```
# Automatically generated Chroma configuration file by the
# Chroma stub generator
description foo:bar  # application name foo, operation bar
mode normal          # this is not the training mode
logfile /usr/chroma/etc/foo_bar.log # used for resource prediction
# provides the application-specific information needed by solver
hintfile /usr/chroma/lib/foo_bar_hints.so

# these are all the fidelities and parameters
param size ordered 0-infinity
fidelity resolution ordered 0-1

# number and definitions of each application RPC
num_rpcs 3
rpc 0 a
rpc 1 b
rpc 2 c

# number of application tactics
num_tactics 3

# definition of each tactic. each RPC in the tactic is uniquely
# numbered. seq is used to denote a sequential dependency between
# RPCs while par is used to denote a parallel dependency.
tactic 0 do_a_and_c 2 seq a 0 seq c 2
tactic 1 do_b_and_c 2 seq b 1 seq c 2
tactic 2 do_a_b_and_c 3 par a 0 par b 1 seq c 2
```

This is an example config file required by Chroma when using the $register\_fidelity$ client API call. This file provides all the relevant information needed by Chroma to choose a runtime setting for the application.

Figure A.11: Chroma Application Config File Example

API Command	Description
int **service_init** (int* argc, char*** argv)	Registers the application with $rmexec$
int **service_getop** (int* op_num, int* opid, char* in_file, void** data, int* datalen)	Retrieves an RPC request from $rmexec$. This call blocks until work arrives. op_num is the requested RPC's number and $opid$ is a unique identifier. in_file is the name of the input data file (if any). $data$ is the input data to the RPC and $datalen$ is the length of $data$.
int **service_retop** (int opid, char* out_file, void* data, int len)	Returns the results of an RPC operation. $opid$ should be the same $opid$ received in $service\_getop$ ($rmexec$ uses this id to determine which client to send the results to). out_file is used to return results stored in a file (out_file is the name of the file). $data$ is a buffer containing the results and len is the length of $data$.

This figure shows the API calls that are used by application server components to communicate with Chroma's server component ($rmexec$).

Figure A.12: Chroma's Server API

A.8 Chroma APIs

The APIs used by applications to communicate with Chroma are described in this section. Client application components running on a mobile users device connect with the client component of Chroma (called $viceroy$) using the API described in Figure A.10. Server application components communicate with Chroma's server component ($rmexec$) using the server API described in Figure A.12.

A.8.1 Client Component

The client API is used as follows. First the developer creates the application's Chroma config file (shown in Figure A.11). At this time, the developer must also decide on a consistent numbering scheme for the application's RPCs. This numbering scheme is used by do_remote_op and do_local_op.

With this config file and numbering scheme, the developer can start modifying the application itself. First, she must add the $register\_fidelity$ call at the start of the application to establish a connection with Chroma. She then must find the computationally intensive portions of the application (these are the parts that need to be remotely executed).

One this portion has been found, she must then do the following tasks; a) pack all the application parameters into a $param\_val\_t$ data structure, b) allocate enough space for all fidelity variables in a fid_val_t data structure, c) add a call to $begin\_fidelity\_op$, d) retrieve the returned tactic and fidelity variable settings, and e) use the fidelity variable settings to set any required state before performing the operation. Chroma will start logging the resource usage of the application when $begin\_fidelity\_op$ is called.

Next, she has to perform the operation. This requires her to first create the appropriate control function for the chosen tactic. For example, if the chosen tactic has 2 sequential RPCs, she will have to allocate intermediate buffers to save the results of the 1st RPC. She will also have to add the calls to execute each of the two RPCs. If the tactic has parallel stages, she will have to use a threads library (such as pthreads or LWP) to perform the parallelism. Before executing an RPC, she will have to determine if it is a local or remote RPC (by checking the server location fidelity values returned by $begin\_fidelity\_op$). She should use do_local_op for local RPCS, and do_remote_op for remote RPCs. She also has to ensure that all the arguments for the RPC are properly packed into a byte stream (using the in_data argument of do_local_op and do_remote_op), saved into a file (using the in_file argument), or both. After each RPC has completed, she has to unpack the results from either out_data and/or out_file. These results must then be either copied into application buffers or into temporary buffers depending on where in the tactic that RPC occurred. After the tactic is completed, she must call $end\_fidelity\_op$ with the appropriate success code. This tells Chroma to stop logging the resource usage of this operation.

The stub generator component of RapidRe (described in Section 2.6) greatly simplifies this entire process. It generates the Chroma config file needed by $register\_fidelity$. It also packs the parameters and fidelity variables into the appropriate data structures before calling the Chroma API calls. The stub also remembers the runtime setting returned by Chroma and generates the control code to correctly execute this runtime setting. In particular, it ensures that intermediate data and parallel RPC stages are properly handled. This control code is hidden under the $do\_tactics$ application-specific API call used by developers. The exact details of each generated API call is presented next.

The stub code contains an opaque data structure that contains all the application-specific state exchanged between Chroma and the application. This allows the application-specific APIs presented to the developer to be as simple as possible. For example, many of them have no arguments. In reality, the arguments for these API calls are automatically stored and maintained by the stub code.

set_size: Every set call for a parameter works as follows: the stub generator will allocate an entry for every parameter in the opaque data structure. Whenever a set call is made, the entry for that parameter is updated to the argument value of the call. The values in this data structure are used when performing the $find\_fidelity$ call.

$get\_resolution$: Every get call for a fidelity variable works as follows: the stub generator will allocate an entry for every fidelity variable in the opaque data structure. The runtime settings for these variables, as determined by Chroma when $find\_fidelity$ is called, will be stored in this structure by the stub. The get call will return the current stored value of this variable to the application.

$register$: This API call uses the Chroma $register\_fidelity$ API call to register the application with Chroma. The stub generator automatically generates the config file needed as one of the arguments to this call. It also statically determines the number of application RPCs (needed as the 2nd argument of $register\_fidelity$) from the tactics file. The stub code remembers the op_id returned by Chroma and automatically uses it for subsequent API calls. This id is used to uniquely identify the application to Chroma. This API call also initializes the opaque data structure used by the stub.

$cleanup$: This API call use the Chroma $close\_chroma\_sockets$ API call (with the statically determined number of application RPCs as the argument) to disconnect the application from Chroma.

$find\_fidelity$: The stub code automatically packs a data structure containing the current values of all application parameters. It also creates a fidelity variable data structure with enough space to store the return value for every application fidelity variable. It then calls the Chroma $begin\_fidelity\_op$ API call and gives it the parameter structure, the saved op_id, and the fidelity variable data structure. After the Chroma API call completes, indicating that Chroma has decided on the operation setting, the stub code saves the returned fidelity variable values in the opaque data structure. It also saves the returned fidelity variable and tactic choices in the opaque data structure. If data decomposition is used, the stub will use the $begin\_fidelity\_op$ API call instead and save the list of possible servers in the opaque data structure.

$do\_tactics$: The stub generates the most amount of code for this application-specific API. For every possible tactic, the stub generates the appropriate control function to correctly execute the RPCs in that tactic. If a tactic has parallel stages, the generate control function will perform parallel RPC calls using the pthread threading library. In addition to the control function, the stub generates the appropriate code to marshal the arguments and unmarshal the outputs for each RPC. The output of $find\_fidelity$ specifies the tactic to execute and the server to use for each RPC in the chosen tactic. The stub uses the do_local_op Chroma API call to perform an RPC on the mobile client itself and the do_remote_op Chroma API call to perform an RPC on a remote server. The stub generator ensures that developers do not need to care about any control, networking, and data packing issues involved with executing tactics. If data decomposition is used, the stub will generate the required buffers to store the partial data and generate the code to a) call the developer provided split function to split the input data, b) parallelize the decomposable stages, c) use a different server and piece of data for each parallel stage, and d) combine the partial results, by calling the developer provided join function, to create the final result.

A.8.2 Data Decomposable Client Component

To create a data decomposable client component, the developer must do the following. First the application is registered with Chroma using exactly the same method described above.

Next, the developer asks Chroma for a set of servers that can be used for the decomposable operation. This is different from regular clients where Chroma returns a single optimal runtime setting. The developer uses the $begin\_fidelity\_op\_decomp$ API call (instead of the regular $begin\_fidelity\_op$ API call. This API call is not shown in Figure A.10) to tell Chroma to use the data decomposition solver described in Section A.6.1. This new API call has the same output as $begin\_fidelity\_op$ except that it also returns the list of servers that can be used for the decomposable operation.

As described above, the developer is then responsible for creating the control function that executes the chosen tactic. Standard sequential and parallel stages are handled as described in Section A.8.1. Decomposable stages must be handled in the following way;

The developer must first decompose the original input data for the first RPC in the stage into a number of smaller pieces. The number of pieces cannot exceed the number of possible servers. However, it can be lower than the number of possible servers if too many splits would result in data chunks that are too small to be optimally remotely executed. The application developer is responsible for both writing the logic that performs this split and for creating the necessary buffers to store split data.

The decomposable stage can then be parallelized (using pthreads or other threading libraries) with each parallel stage given a different split piece of data as input. To ensure that each RPC in a parallel stage uses a different server (to avoid server contention), the developer must use the $do\_remote\_op\_server$ API call (not shown in Figure A.10) instead of the regular do_remote_op API call. This new API call has an extra argument that allows the developer to explicitly specify which server to use to perform the remote RPC. The developer should use the list of possible servers returned by $begin\_fidelity\_op\_decomp$ to pick a unique server for each parallel stage. The developer has to create the appropriate data structures to store the partial results returned by these parallel stages.

After every parallel stage has returned, the partial results must be combined to form the final result. The developer is responsible for writing the logic that performs this recombination. This combined result is then returned to the user (if the decomposable stage was the last stage in the tactic) or passed as input to the next stage in the tactic.

When using RapidRe, the entire control code logic (putting a different server for each chunk etc.) and intermediate data buffers is completely generated by the stub generator. The developer merely has to provide the functions to decompose and recombine the data.

A.8.3 Server Component

The server API is used as follows. First, the developer registers the server with Chroma ($rmexec$) by inserting a call to $service\_init$ at the start of the server. Next, she preserves any application-specific initialization routines. These have to be preserved to ensure that the server is properly started.

She then writes an event loop that a) receives new client requests using the $service\_getop$ API call, b) determines the RPC this request was for, c) unpacks the arguments for the RPC (the $data$ (for packed data) and in_file (for filedata) arguments of $server\_getop$), d) calls the appropriate RPC with the unpacked arguments, e) packs the outputs of the RPC into a buffer, a file, or both, and f) returns the outputs to the requesting client using the $service\_retop$ API call (with the correct $opid$). This event loop should loop continuously until the server exits.

Finally, she creates the RPC functions that the server has to support. This is usually an easy step as the functions usually already exist in the application (usually the code that was removed from the client). Hence, usually it is sufficient to just write a little bit of glue code that accepts the arguments from the RPC request and calls the correct existing application function. A little additional glue code that extracts the outputs of the existing functions and puts them into buffers to return to the user may also need to be written.

The RapidRe stub generator generates the event loop that polls $rmexec$ for client requests, executes the appropriate RPC function, and then returns the results to the client (via $rmexec$). This event loop is called by the run_chroma_server automatically generated call. The developer just has to provide the RPC functions. The exact operation of each generated API call is presented next.

$server\_init$: For this API call, the stub generates the code that performs the appropriate steps to connect and register the application with the underlying Chroma server mechanisms. These mechanisms allow clients to query the resource status of remote servers.

run_server: This API call is the main generated server function. The stub generator generates a continuous event loop that a) listens for client requests (using $service\_get\_op$), b) unmarshals the arguments for the request, c) calls the appropriate RPC function (these functions have to be created by the developer) with the correct arguments, d) retrieves the RPC output, marshals it, and sends it back to the client (using $server\_retop$). This generated API call completely shields the developer from the low level networking and data packing details necessary for an effective server.

A.9 Putting it All Together: Chroma in Action

When the mobile client enters a new environment, Chroma first discovers and authenticates the available remote servers. Chroma then starts application servers on each of these remote servers. Next, Chroma starts measuring the available resources, both locally and on the remote servers. Sometime during this process, Chroma will obtain utility functions that encapsulate the user's preferences from an external entity such as Prism.

When an application wants to perform an operation, Chroma will first obtain the list of tactics and fidelity variables for that application. Next, it predicts the resource usage and expected execution latency of every runtime setting using the history-based demand predictors and the supplied application parameters. Given these pieces of information and the measured resource levels, Chroma will then solve for an optimal runtime setting that maximizes the utility function. This runtime setting is returned to the application.

The application then performs the operation using the tactic plan and fidelity variable settings contained in the runtime setting. The actual resource usage of the operation is measured and used by Chroma to update the history-based demand predictors.

If the application is using data decomposition, the sequence is slightly different. Instead of returning an optimal runtime setting, the solver returns the chosen fidelity variables and the set of servers that can be used. The application then decomposes the input data into smaller pieces – up to the number of servers returned by the solver. Each server will then be given a piece of data and will compute a partial result. These partial results are then pieced together, on the mobile client, to form the final result.

A.10 Summary

In this appendix, I described the components of Chroma. Chroma is a dynamic adaptive runtime system that is able to determine the optimal runtime setting that meets the current resource availability and user preferences. To do this, Chroma provides resource monitoring, resource prediction, and a solver that is able to choose the optimal runtime setting. Chroma also has limited functionality to discover available remote servers and to start application servers on these servers. Finally, Chroma provides the appropriate mechanisms for applications to use these servers for remote executions.

Appendix B

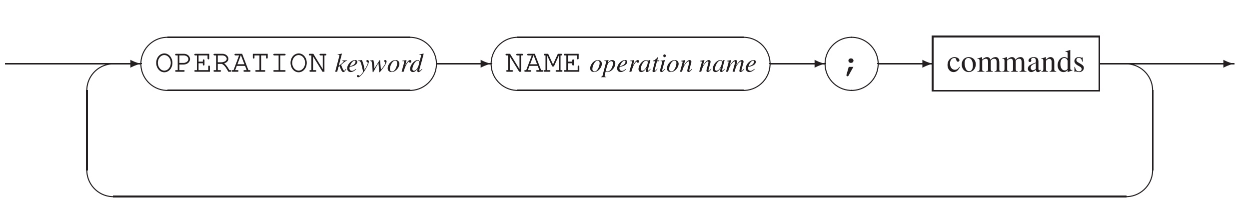
Vivendi Syntax

In this Appendix, the full Vivendi syntax is presented in the form of Backus Naur Form (BNF) diagrams. These diagrams should be read as follows: the starting state appears in bold font above each diagram. For example, $start$ is the first state in the syntax. Non-terminal states appear in rectangular boxes (e.g. $operation$ in the $start$ state). Terminal states appear in rounded boxes (e.g., $APPLICATION$ and $NAME$) and may have subscripted comments. These comments either state that the terminal name is a reserved language keyword or explains what value the state expects to be provided. For example, $APPLICATION$ is a keyword and $NAME$ requires the application name.

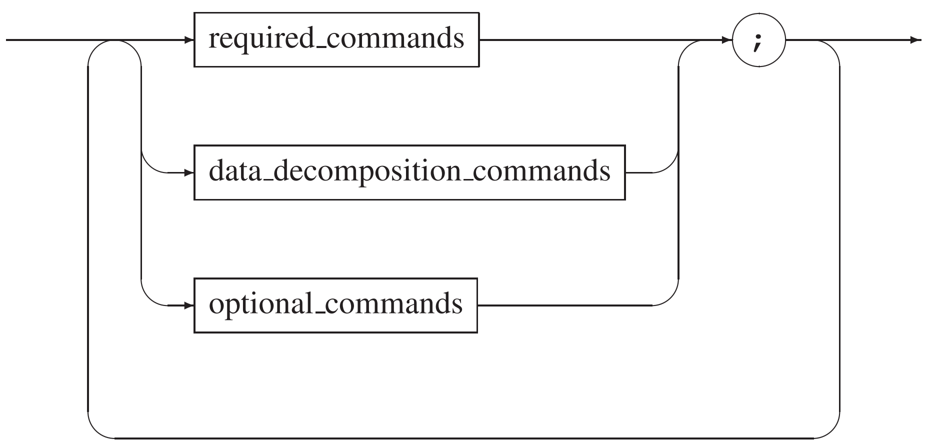
These diagrams use two types of lines. Rounded lines that curve back into a state indicate that the states between the lines can be repeated arbitrarily many times. An example of this type of line is shown in the $operation$ state. The other type of line (with arrows) indicates an alternative choice for the next state. An example of these type of "alternative choice" lines is shown in the $required\_command$ state. Note that some states (like $commands$) have both types of lines.

start

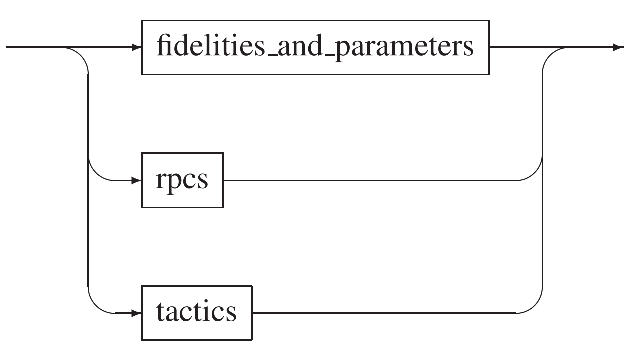
operation



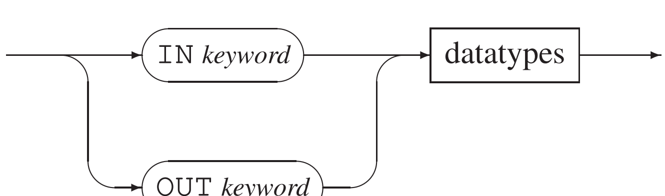
commands



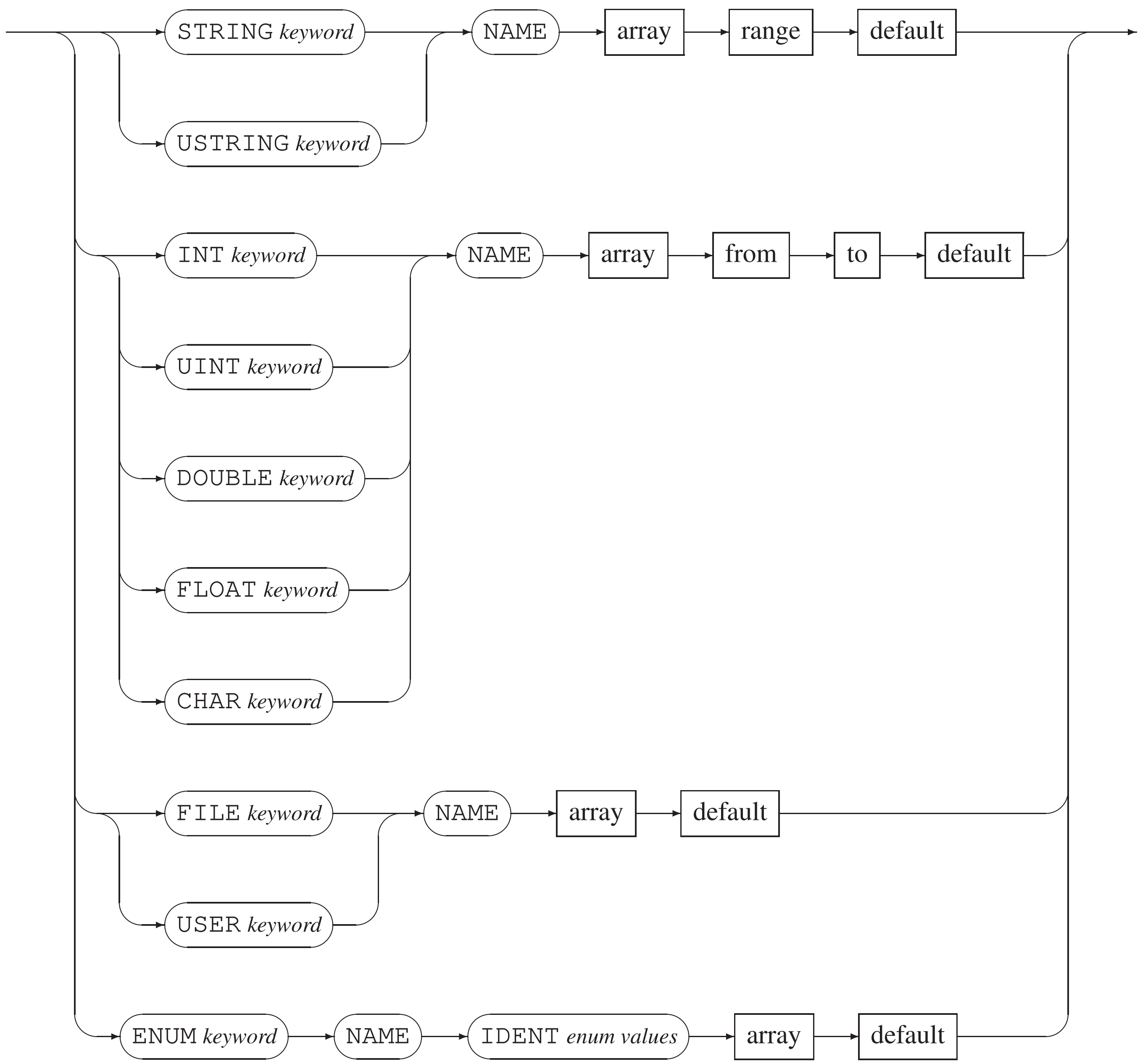
required_commands



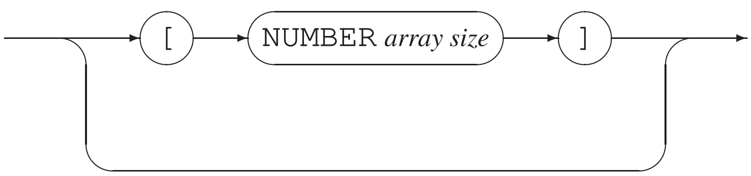
fidelities_and_parameters



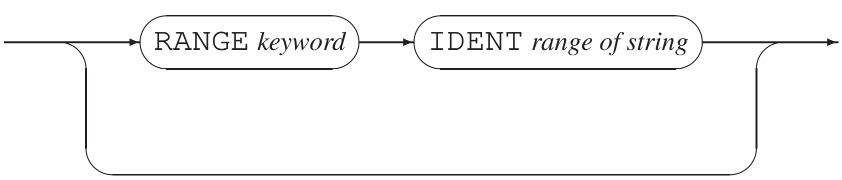
datatypes


STRING keyword
USTRING keyword
NAME
array
range
default
INT keyword
UINT keyword
DOUBLE keyword
FLOAT keyword
CHAR keyword
NAME
array
from
to
default
FILE keyword
USER keyword
NAME
array
default
ENUM keyword
NAME
IDENT enum values
array
default

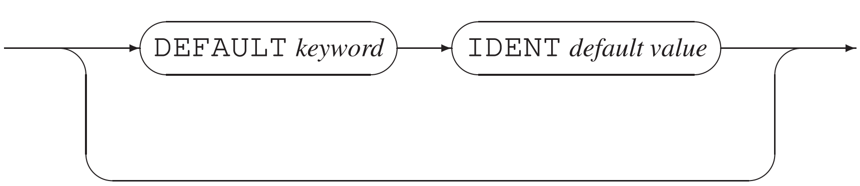
array



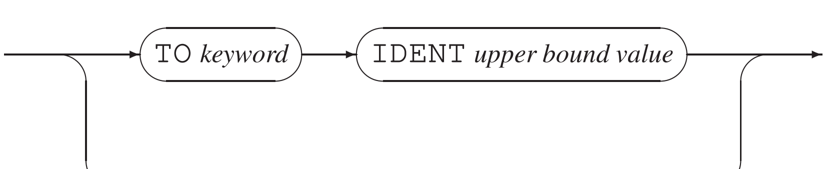
range



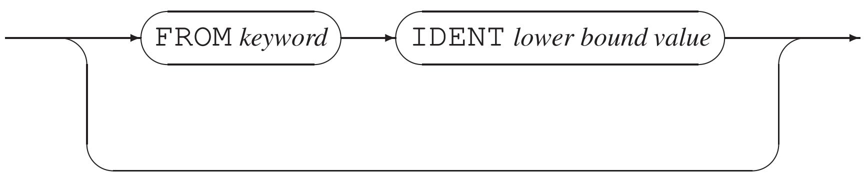
default



to



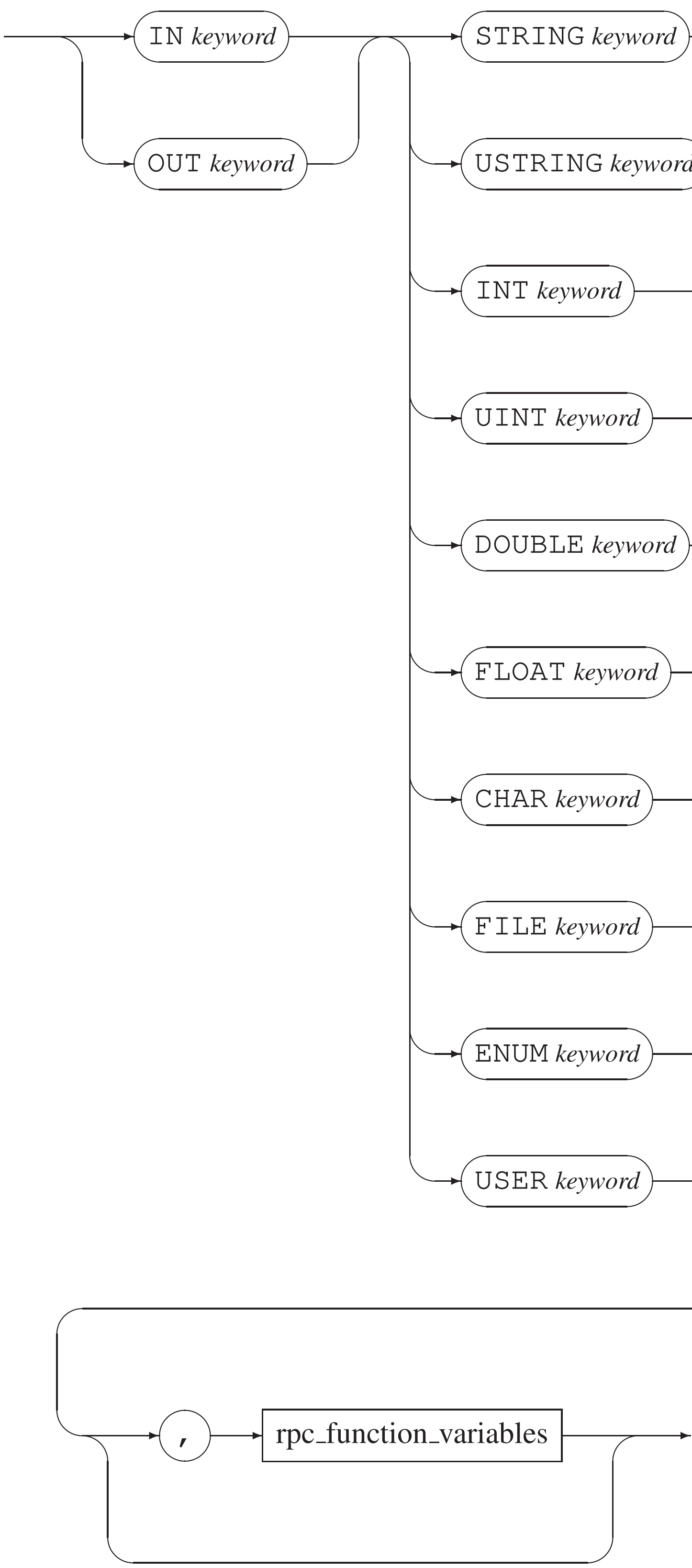
from



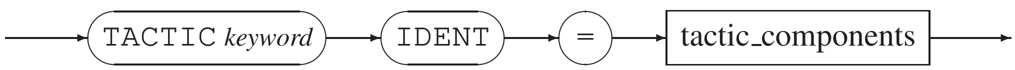
rpcs

rpc_arguments

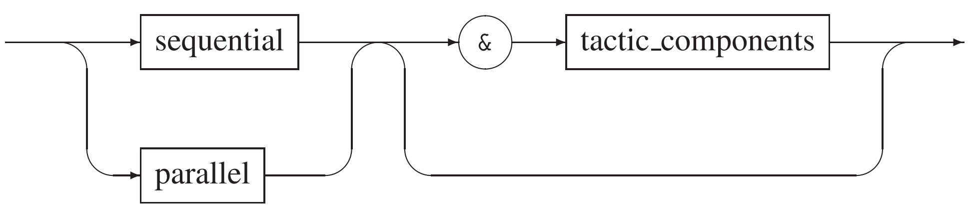
rpc_function_variables



tactics



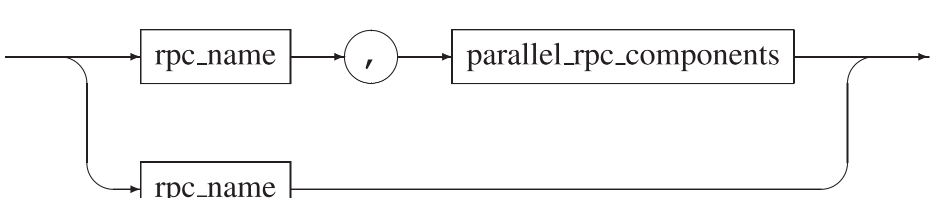
tactic_components



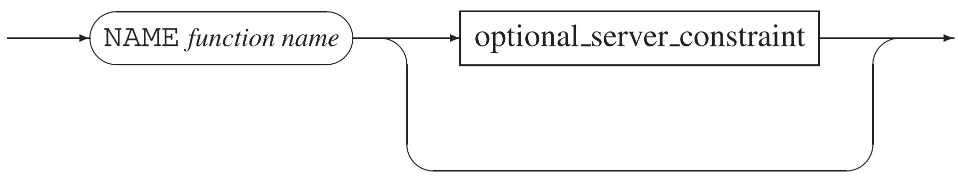
sequential

parallel

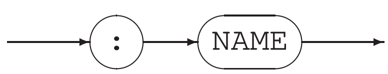
parallel_rpc_components



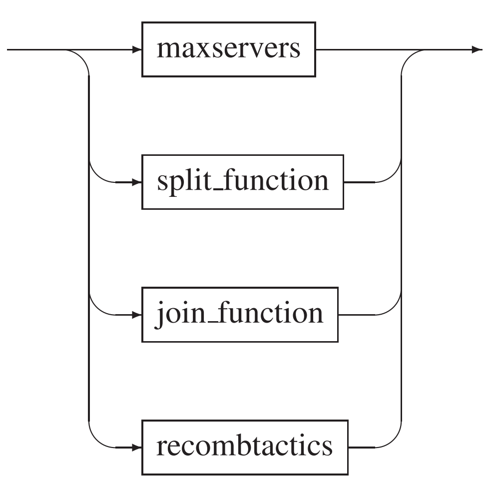
rpc_name



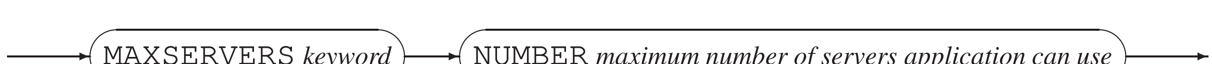
optional_server_constraint



data_decomposition_commands



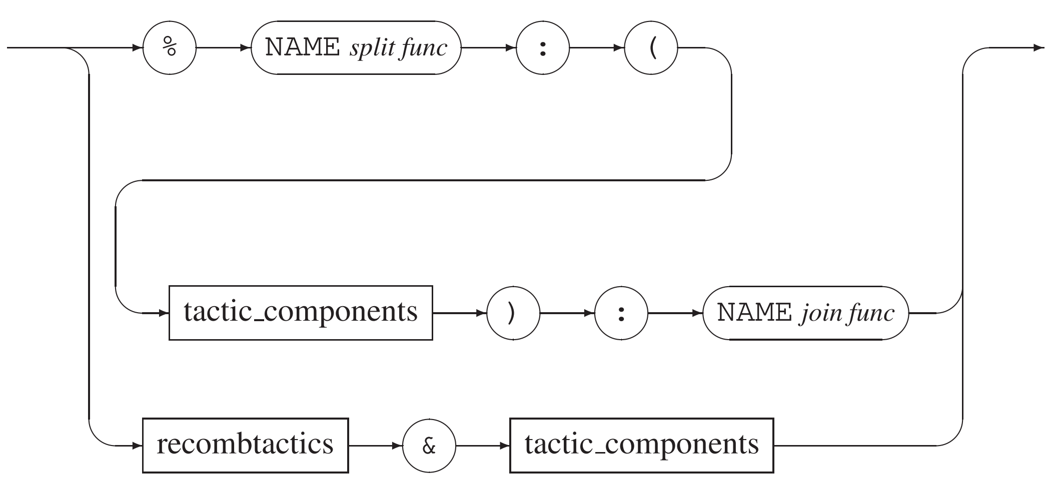
maxservers



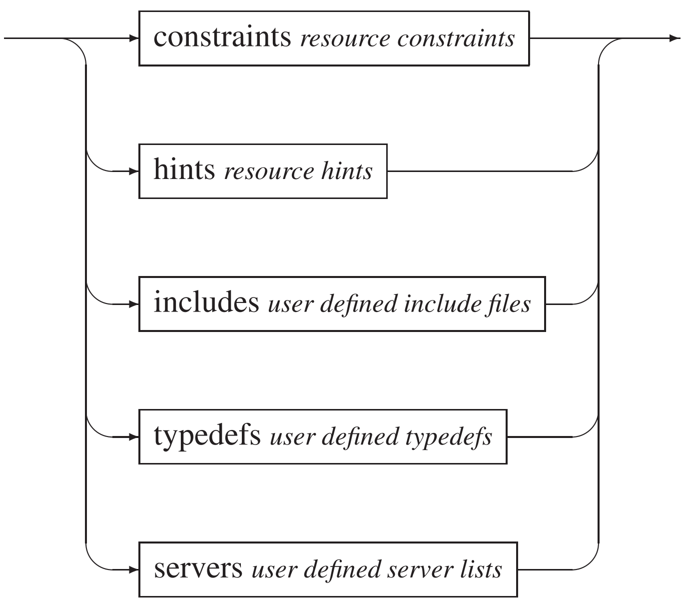
split_function

join_function

recombtactics



optional_commands

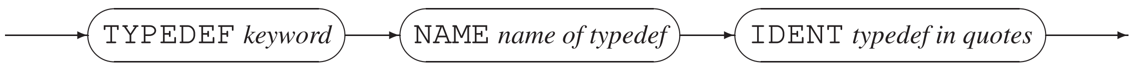


constraints

hints

includes

typedefs



servers

Appendix C

User Study Procedure

In this Appendix, I explain how the user study was conducted. Section C.1 provides the high level scenario that was used to setup the user study. Section C.2 gives the detailed procedure that was used for the user study.

C.1 Scenario for User Study

We have 3 entities. The original application developer, a domain expert, and a novice application developer whose goal is to add the application to an adaptive runtime framework.

The original application developer will develops his application for its intended purpose and leaves behind a good set of documentation that explains

1. the high level control flow of the application
2. specific details about individual files / procedures

When writing this documentation, the original application developer was not thinking that his application would be used in Chroma or by an adaptive remote execution system. The application may already be structured for client/server operation if that was it's original design.

A domain expert (someone who understands the domain which the application belong to. For example, video players, language translators, speech recognizers, 3D model displayers etc.) writes a brief document that broadly describes what parameters and fidelities are appropriate for that domain. This will be phrased in generic language.

For example, a 3D model display expert may say that parameters that affect resource usage are the size of the display window (height and width of output window), the number of polygons in the model, the dimensions of the model (using the coordinate space of the application), the current position of the viewer (using the application's coordinate space) and the current perspective of the model (how it is rotated etc. again using the application's coordinate space). After providing these parameters, the application will need to be informed of the appropriate resolution at which to render the display.

The task of determining which particular application variables and values map to these parameters and fidelities is left up to the novice developer. There may be hints in the original application developer documentation (for example, the documentation may specify what the application's coordinate space is and how to extract values corresponding to different points in the space).

Each application will thus have 2 sets of documentation. The specific application documentation left behind by the original developer and more general "what are the adaptive knobs of this kind of application" documentation left behind by the domain expert. Given these two sets of documentation, the novice developer will have to

1. create a tactics file for the application
2. make the application adaptive by correctly partitioning the application into client/server components and by correctly inserting the API/macro calls in the right places. Some new application glue code may also need to be written.

C.2 Detailed User Study Procedure

Pre-Experiment Checklist

- Ensure all documentation is ready for the participant. Refer to documentation checklist (Section D.1) for this.
- Bring the two windows XP laptops + power supplies + hub + 4 network cables to seminar room. don't even think of running the experiment over wireless.

Experiment Checklist

- participant arrives
  - greet the participant
- assign participant a unique number
- explain to them that they must use only the number of the questionnaires
- explain to them the consent form (Appendix D.2.1) and make them sign two copies.
  - give them one to keep

- explain to them the scenario / give them scenario handout (Appendix D.3)
- start training (process described in Appendix C.3)
  - go through the "How to create adaptive applications" handout (Appendix D.3.2) and explain operation, parameters, fidelities, RPCs, tactics
  - explain overall picture. They need to create a tactics file and then modify application
  - do think aloud protocol [140] training. Think aloud protocols require participants to think aloud as they are performing a set of specified tasks. Participants are asked to say whatever they are looking at, thinking, doing, and feeling, as they go about their task. This enables observers to see first-hand the process of task completion (rather than only its final product). Explain to the participants that they should think aloud when
    1. starting a new section
    2. when encountering a problem that they are having trouble with
  - start with tactics training with GOCR
- explain to them the software setup
  - 3 Xterms will be open. one xterm will be in the build directory. They should ignore that xterm for now
  - The other 2 Xterms will be open in the source directory for the application being retargeted
- give them the tactics file documentation ("Creating the tactics file" section in Appendix D.1. Explain each piece of documentation to them
- explain to them the user study procedure
  - they have to say when they are done with each section as listed in the "overview of process" handout (Appendix D.6.1)
  - fill in the stage questionnaire (Appendix E.1) after each section
- start Camtasia [210]. This software records the laptop's screen and captures every mouse and keyboard input made by the participant. This recording allowed me to review exactly what each participant did at a future time (and also allowed me to obtain very accurate timing information)
- let them begin
- at the end of the tactics file, stop Camtasia
  - save Camtasia recording as a meaningful file name participant_number,date,time,application,experiment_part

- let them take a break
- set up for modifying the application
- copy their tactics file to somewhere safe in /coda
- copy good tactics file into source directories (client and server)
- Do the modifying the application training. This training, using GOCR, will cover the steps needed to create both the client and server application components
- run make in the build tree to create the stubs
- give them the "modifying the client" documentation ("Creating the retargeted client application" section in Appendix D.1) and explain each piece of documentation
- remind them of the think aloud protocol and the need to say each stage and fill in the questionnaire
- start Camtasia
- let them modify the client
- at the end, stop Camtasia and save the recording using a meaningful filename
- let them take a break
- give them the "modifying the server" documentation ("Creating the retargeted server component" section in Appendix D.1) and explain each piece of documentation
- start Camtasia for server modification (remind them of the think aloud and stuff)
- let them modify the server
- at the end, stop Camtasia and save the recording using a meaningful filename
- give them the "Overall retargeting the application" questionnaire (Appendix E.5)
- if the participant was retargeting their 2nd or 3rd application, give them the "Effect of experience" questionnaire (Appendix E.7)
- give them the "Overall user study experiment" questionnaire (Appendix E.6)
- make them fill up a payment claim form (Appendix D.2.2) and fill in the appropriate amount
  - $120.00 for a successful modification
  - lower amounts for unsuccessful modification. actual amount to be determined on a case by case basis

C.3 Training Plan

This is the 4-step training process that was given to every participant.

1. Explain how to create a tactics file
   (a) explain what the APPLICATION line is
   (b) explain what the OPERATION line is
   (c) explain how to specify parameters
   (d) explain how to specify fidelities
   (e) explain how to specify RPCs
   (f) explain how to specify TACTICs
2. Hands on training on creating a tactics file
   (a) walkthrough how to create a tactics file using GOCR as the test application
3. Explain how to modify the application
   (a) explain how to identify what is client code and what is server code.
   (b) explain the APIs
      i. initialize
      ii. register
      iii. start_operation
      iv. stop_operation
      v. find_fidelity
      vi. do_tactics
      vii. cleanup_params
4. Hands on training on modifying application

- Walkthrough how to modify GOCR to use the stub generated API calls

Appendix D

User Study Documentation

D.1 Documentation Checklist

This is the set of documentation needed by every participant for every experiment.

Basic Documentation

- Consent form (Appendix D.2.1)
- Payment form (Appendix D.2.2)
- Scenario description (Appendix D.3)
- Definitions handout (Appendix D.3.1)
- How to create adaptive applications readme (Appendix D.3.2)

Creating the tactics file (Stage A in Table 4.3).

- Application specific readme (Appendix D.4)
- Application specific DOMAIN readme (Appendix D.5)
- Creating a tactics file overview (Appendix D.6.1)
- Creating a tactics file manual (Appendix D.6.2)
- Stage A questionnaire (they fill this in as they create the tactics file) (Appendix E.1)
- Overall tactics file questionnaire (given at the end) (Appendix E.2)

Creating the Retargeted Client Application (Stage B in Table 4.3).

- Application specific readme (Appendix D.4)
- Application specific DOMAIN readme (Appendix D.5)
- Application tactics file printed out (Chapter 3 has the tactics files for every application)
- Client side modification overview (Appendix D.7.1)
- Manual for modifying applications (Appendix D.8)
- Common routines cheat sheet (Appendix D.7.3)
- Programming notes handout (Appendix D.7.4)
- The C Programming Language (2nd Edition) reference book [131]
- C++ How to Program (4th Edition) reference book [58]
- Stage B questionnaire (they fill this in as they retarget the application) (Appendix E.3)

Creating the Retargeted Server Application (Stage C in Table 4.3).

- Application specific readme (Appendix D.4)
- Application specific DOMAIN readme (Appendix D.5)
- Application tactics file printed out (Chapter 3 has the tactics files for every application)
- Server side modification overview (Appendix D.7.2)
- Manual for modifying applications (Appendix D.8)
- Common routines cheat sheet (Appendix D.7.3)
- Programming notes handout (Appendix D.7.4)
- The C Programming Language (2nd Edition) reference book [131]
- C++ How to Program (4th Edition) reference book [58]
- Stage C questionnaire (they fill this in as they retarget the application) (Appendix E.4)

After the Application has been Retargeted Completely

- Overall retargeting the application questionnaire (Appendix E.5)
- Overall user study experiment questionnaire (Appendix E.6)
- Effect of experience questionnaire (only given to participants who are retargeting their 2nd or 3rd application) (Appendix E.7)

D.2 Basic Documentation

D.2.1 Consent Form

Research Participant Information and Consent/Authorization Form for Minimal Risk Studies

The purpose of the Study is to investigate the effectiveness of a set of tools and methods in enabling the rapid development of adaptive applications.

You will be asked to make an existing application work within an adaptive runtime system and then provide feedback as to how easy you felt this task was. The task will consist of three different phases and the total experiment time (including training) should not exceed seven hours. For this study, everything you type along with anything you say during the study will be recorded.

You should experience no risk or discomforts from participating in the Study.

There will be no cost to you if you participate in this Study.

There may be no personal benefit from your participation but the knowledge received may be of value to research in distributed systems.

Your participation is voluntary. Refusal to participate or withdrawal of your consent or discontinued participation in the study will not result in any penalty or loss of benefits or rights to which you might otherwise be entitled. The Principal Investigator may at his/her discretion remove you from the study for any of a number of reasons. In such an event, you will not suffer any penalty or loss of benefits or rights which you might otherwise be entitled.

You will receive $120 for successfully participating in this Study. You will receive $40 if you stay at least three hours but do not successfully complete the Study. Payment amounts for other situations will be at the sole discretion of the Principal Investigator. This payment is for your time and personal cost of participation.

Your anonymity will be maintained during data analysis and publication/presentation of results by any or all of the following means: (1) You will be assigned a number as names will not be recorded. (2) The researchers will save the data file and/or any video or audio recordings by your number, not by name. (3) Only members of the research group will view collected data in detail. (4) Any recordings or files will be stored in a secured location accessed only by authorized researchers.

If you have any questions about this Study, you should feel free to ask them now or anytime throughout the Study by contacting:

Professor Mahadev Satyanarayanan
School of Computer Science
Carnegie Mellon University,
Wean Hall 8208,
5000 Forbes Avenue,
Pittsburgh, PA 15213
Phone : (412) 268-3743
E-Mail : satya@cs.cmu.edu

If you have questions pertaining to your rights as a research participant; or to report objections to this Study, you should contact:

IRB Chair
Regulatory Compliance Administration
Carnegie Mellon University
5000 Forbes Avenue
Warner Hall, 4th Floor
Pittsburgh, PA 15213
Email: irb-review@andrew.cmu.edu
(412) 268-1901 or (412) 268 4727

The Carnegie Mellon University Institutional Review Board (IRB) has approved the use of human participants for this Study.

This study is funded by the Aura Project, which is supporting the costs of this research. **Neither** Carnegie Mellon University (CMU), **nor** Prof. Mahadev Satyanarayanan will receive any financial benefit based on the results of the Study.

I understand the nature of this Study and agree to participate. I received a signed copy of my consent. I give the Principal Investigator, and his/her associates, permission to present this work in written and/or oral form for teaching or presentations to advance the knowledge of science and/or academia, without further permission from me provided that my image or identity is not disclosed.

____ _____________________

PARTICIPANT SIGNATURE DATE:

D.2.2 Payment Form

Name : ____

Social Security Number : ____

Address : ____

Phone Number : ____

Payment Amount : ____

____ _____________________

PARTICIPANT SIGNATURE DATE:

D.3 User Study Scenario

You have just graduated from school and taken a job as an application developer at a company that develops applications for mobile devices. The company has developed an adaptive runtime system, called *Chroma*, for mobile devices that is able to dynamically execute applications on remote servers to improve performance (since mobile devices generally have limited hardware).

The company has access to a large number of applications that are useful for mobile devices. These include face recognizers, 3D model viewing application, language translators, speech recognizers, optical character recognizer and speech synthesizers. However, none of these applications were designed specifically for mobile devices and they are not currently integrated with Chroma.

Your job, as the new hire, is to take these applications and modify them to work with the adaptive runtime system.

Usually, this task would require you to completely understand each application and the internal workings of Chroma. You would then have to modify the application to work with Chroma (which involved packing and unpacking variables, adding calls to internal Chroma functions in the appropriate places, and handling remote execution of application code).

However, the company has developed a set of tools along with a modification methodology that they claim will greatly reduce the time needed for you to modify an application to work with the adaptive runtime system. The tools and methodology were designed such that you will not need to know anything about the internal workings of Chroma and very little about the application being modified. Even then, the tools allow you to create a modified application that is almost as effective (performance wise) as an application created using the old manual process.

The purpose of this user study is to quantify how effective these tools and methodology are. For this study, you will modify different applications to work with the Chroma adaptive runtime system.

D.3.1 Definitions Handout

Operation :- An application specific notion of work. For example, a graphics application would have an operation called render while a language translation application would have an operation called translate. The operation is thus the core functionality of the application.

Parameter:- Application specific variables who's values affect the resource usage of the application. For example, camera position for graphics applications and number of words for language translation applications. Each adaptive application will have **at least** one parameter. There is no upper bound on the number of parameters.

Fidelity :- These are values that the application asks the runtime to set. I.e., the application tells the runtime, "Tell me what the values of these variables should be for the given resource conditions". The runtime will decide what the appropriate value for the fidelity should be for the current resource conditions. The tradeoff is that using a lower fidelity value will use fewer resources but result in a lower quality output. For example, a graphics application may want the runtime to tell it what resolution to use for the current operation. A higher resolution results in a higher quality rendering but will consume more resources (and hence take too long to complete on certain hardware platforms). A lower resolution will consume less resources (and render faster) but result in a poorer output.

After the runtime returns values for these variables, the application will need to use the values to set any internal parameters that depend on those values. For example, a graphics application may ask the runtime to determine the resolution for each render operation. After obtaining this value from the runtime, the application may need to set certain rendering parameters appropriately before performing the operation. An application may not have any fidelities.

Chroma :- An adaptive remote execution system that is able to adapt applications using two orthogonal mechanisms.

1. It can change the values of the fidelities
2. It decides which tactic to use to perform the operation

RPC :- Remote Procedure Call. This is a method of remotely executing application procedures at different servers.

Tactic :- Tactics are different ways of combining various RPC calls together to satisfy the operation. At runtime, Chroma will pick one of these ways based on the criteria explained below. For example, language translation can be done using anywhere from 1 to 4 procedure calls. The tactics specification would list all possible valid ways of doing those 4 procedure calls to satisfy the operation. At runtime, Chroma will decide which tactic to choose (similar to fidelities, the tactics differ in their resource usage and quality) that optimizes the application performance (achieve the best quality while using the optimal amount of resources).

D.3.2 Adaptive Applications Overview

Applications can be made adaptive in at least two different ways

1. By adjusting variables that affect the resource usage of the application. The trade-off is that the quality of the application also changes when you change the values of these variables. These variables are called ***fidelities***. For example, for graphics applications, you can adjust the rendering resolution from the maximum value of 1.0 (full resolution) to 0.0 (no polygons at all). Making the resolution lower requires less CPU and memory resources to render the model (as the number of polygons to render is lower) but results in a lower quality output
2. By remotely executing parts of the application on faster remote servers. For example, in the same amount of time, you may be able to either render a scene with 0.5 resolution locally or render it at 1.0 resolution on a faster remote server. Clearly, rendering it remotely results in a better quality result and uses less resources locally (This might be important for mobile devices as using more resources increases battery usage). Of course, the time taken to transfer data to and from the remote server must also be taken into account (along with the energy used for this transmission if energy usage is a concern) before deciding to use remote servers.

The Chroma adaptive runtime system is able to use both these methods to adapt applications. These methods are orthogonal. For example, the best solution might be to render remotely using a resolution of 0.8.

D.3.2.1 What are Tactics?

However, automatically deciding how to remotely execute an application is impossible in practice. To make this problem more tractable, Chroma uses the concept of ***tactics***. Tactics are a specification of all the useful ways an application can be remotely executed. For many applications, there is only 1 tactic. For example, for the 3D application, the only tactic is to remotely execute the rendering procedure. However, for other applications, there may be more than one way to generate a correct result. Again, similar to fidelities, these different ways will differ in their resource usage and output quality.

For example, assume that a language translation application (application that translates one language into another) is able to run multiple independent translation engines on the provided text file. The intermediate output from all these engines is then sent to a final "combiner" stage that combines these partial results to create the final output. For this application, there would be multiple tactics. Each of these tactics would use a different number (and set) of the translation engines before finally sending the output to the combiner. Obviously, the best result is achieved by the tactic that uses all available translation engines. However, this tactic also uses the most resources.

D.3.2.2 What are Parameters and Fidelities

Parameters : The Chroma runtime system contains the necessary machinery to determine, at runtime, what the best values for the application's fidelities should be and which tactic should be used. Choosing a tactic also involves determining the actual servers to use for each component in the tactic.

To make its decisions, Chroma requires the application to provide it with certain information. These are the values being used for the current ***operation*** (an operation is an application specific instance of work. For example, each time the graphics application re-renders its display is a different operation) that affect the resource usage of the application. We call these values ***parameters.***

For example, to render a 3D model, the model being rendered (identified by its name), the size of the rendering window, the current camera positions, and the current perspective of the model affect the resource usage of the operation. These values are static in that Chroma is not allowed to change them. The application sets these values and Chroma cannot change them. Basically, the application is telling the runtime this

"I need to do some work. The work uses these settings which affect my resource usage. With this information, decide what values I should use for my fidelities and decide which tactic I should use."

Fidelities : Parameters should not be confused with fidelities. Parameters are values set by the application that affect the resource usage of the application. These may not be actual variables in the application. For example, the size of the input file may be an important parameter. This value will have to be calculated.

Fidelities, on the other hand, are actual application settings that need to be set before the operation can be performed. The application is letting Chroma determine the appropriate values for these settings (based on the current resource usage) to achieve optimal performance. For example, always rendering the display at 1.0 resolution may take too long in cases where the CPU capability is limited. It may be better to reduce the resolution and achieve faster renderings.

Chroma uses the parameter values to determine the resource usage of the application. Based on this, Chroma sets the values of the fidelities and decides which tactic to use such that they will not exceed the available resources. These resources could include remote servers if they are available.

D.3.2.3 Adding Chroma Support to an Application

An application will need to do the following to use Chroma.

1. Tell Chroma the current values of the parameters. This is accomplished by using the *set_xxx* macro calls generated by the stub generator (*xxx* is the name of the parameter).

2. Call *foo_bar_find_fidelity*. Chroma will then set the values of the fidelities and decide which tactic to use

3. Read the values of the fidelities. This is done using the *get_xxx* macro calls.

4. Tell the application to use the fidelity values. For example, for a graphics application, after using the *get_resolution* macro call to read the value of the resolution, the developer will need to call the routines that tell the application to use that particular resolution value.

5. Call *foo_bar_do_tactics* to actually perform the operation. This call will use the tactic chosen by Chroma to perform the operation. This could involve remote execution for some or all of the stages in the tactic.

D.4 Application Documentation

In this section, the documentation for each application is presented. This is the documentation that was given to the participants. The documentation can be found as follows: Face (Figure D.1), Flite (Figure D.2), GLVU (Figures D.3, D.4,and D.5), GOCR (Figure D.6), Janus (the XSpeech client (Figure D.7) and the Janus server ((Figure D.8)), Musicmatch (the Java client (Figure D.9) and the C++ server (Figure D.10)), Panlite (Figure D.11), and Radiator (Figure D.12).

```
face_README.txt       Wed May 24 18:21:02 2006       1
face_detector application developer notes
-----------------------------------------

Face was written in ADA. The main ADA function
to recognize faces is located in detect-objects.adb

To make it easy to use the face detector, we also
wrote a C interface procedure that allows C/C++/Java programs
to easily use the Ada routine.

The C interface procedure is

void Detect_Objects (char * input_file, int infile_len,
                     char * output_file, int * output_len);

you pass the routine the path to the image file to recognize faces in
along with the length of the file. You also pass in a buffer
that can contain the name of the output_file along with a integer
to store the length of the output_file name.

Detect_Objects will assign a name to the output file. it does this by taking
the input_file name, stripping away the extension and creating an output
file with the same basename but with extension v3a.overlay.pgm.

For example, given an input file of /home/images/test.pgm
the routine will return an output file name of test.v3a.overlay.pgm

Detect_Objects will *NOT* add the path where the file can be found into the
output_file variable. The file will be placed in the working directory of
the application. The working directory can be found using the getcwd( ..)
command.

An example C application using our C procedure is provided. Look at face.c
```

Figure D.1: Face Readme File

```
flite_README.txt       Wed May 24 18:20:53 2006       1
flite speech synthesis program
------------------------------

flite synthesis text into speech. I.e., it take typed words (either from a
file or stdin) and speaks out those words. It can also produced wav files
containing the spoken words.

flite is a large program containing many libraries.

The main program, called flite_main.c, is located in the main/ subdirectory.

This program does the following (starting from the main () routine)

- initializes the flite synthesis engine
- processes command line parameters (flite_usage explains what these are)
- calls the synthesize procedure

The synthesize procedure (also found in flite_main.c) does the following

- creates a synthesis model

- reads the text input. By default, it will read from stdin unless the
  -t flag is used to provide an input file name

- loops the following routines if flite_loop or flite_bench are set
  at the command line. Looping is mostly used for performance testing

- calls the flite_phones_to_speech if the explicit_phones option is
  set. This is used to train new synthesis models

- calls flite_text_to_speech if no input filename is provided. Input is
  read from stdin. This is the default function that will be called if no
  cmdline parameters are specified.

- calls flite_file_to_speech if an input filename is specified. This
  function will open the file and synthesize the text in it. The output is
  played using the soundcard (if outtype is set to "play") or stored in a
  file otherwise (name specified in outtype).

- prints timing information (if requested) and exits (after deleting the
  synthesis model)
```

Figure D.2: Flite Readme File

glvu_README.txt   Wed May 24 18:20:59 2006   1

```
GLVU allows you to walkthrough a 3D model. It is comprised primarily of
libraries that allow you to quickly navigate and render a 3D model. It uses
OpenGL to perform the 3D rendering.  Any functions you see, in the code,
that start with gl_function_name are OpenGL functions. You can lookup these
functions using the system man command. For example,

man glReadPixels

Directory Tree
--------------

camera
math
object
texdepthmesh
video
tracker
timer
images
lightcam
glutils
glvu :  all these directories contain runtime libraries used by applications
        to render 3D models

example_client
example_server : these 2 directories contain examples applications that use
                 the runtime libraries. For this user study, you will only
                 need to look at the rgl.cpp example application.

-----------------------------------------------------------------------

An example application (in the example subdirectory) called rgl.cpp is
provided to help you understand how to use GLVU to render 3D models onto a
display. There are other examples applications in the same directory.
However, rgl.cpp should be sufficient for figuring out the basic
functionality of GLVU.

rgl.cpp uses the Glut toolkit. Glut is a 3D toolkit that makes it easy to
obtain user input (mouse events) and to create a user interface for
displaying OpenGL output. rgl.cpp reads the name of a 3D model (from the
command line inputs) and then displays the model in a window (using the GLUT
toolkit). The user is able to rotate and walkthrough the model using the
mouse (rgl.cpp will call the appropriate GLVU functions to re-render the
model everytime the user changes the model or the viewing position).

Glut works as follows. The application registers functions to handle various
events. This include functions to handle mouse input, functions to handle
keyboard input and functions to handle display updates (i.e., re-render the
display to react to user input). Glut will call the registered functions
when the appropriate event occurs. In the main program of the application,
you register the various callback functions that Glut should call. You then
make a call to glutMainLoop(). This procedure never returns and starts the
Glut event processing loop. When user input occurs, Glut will call the user
input function (that was registered using the glutKeyboardFunc call for
keyboard events and the glutMouseFunc call for mouse events). When the
display needs to be re-rendered, Glut will call the function registered
using the glutDisplayFunc call.

Detailed Explanation of Example Application rgl.cpp
---------------------------------------------------

This example application shows how to use GLVU to do a simple virtual
walkthrough of a 3D model.

This application has three main functions.

userDisplayFunc0()  /// Glut callback function that renders the model to screen

userIdleFunc0()     /// handles automatic walkthroughs of the model i.e.,
                    /// user specifies randomwalk or randomcam as a command
                    /// line input

userKeyboardFunc0() /// Glut callback function that handles user keyboard
                    /// input (user can change position of model, rotate it,
                    /// go through it etc.) the mouse support and menu
                    /// callback functions are created when the rendering
                    /// model is initialized.  I.e., when rgl.Init
                    /// (explained below) is called


These 3 procedures (along with main() ) comprise the entire application. The
procedures call many internal GLVU libraries and functions (located in other
files) to render the object and handle user input. But conceptually, all the
code to handle the rendering of the 3D model are in these 3 procedures.

The main program does the following

  - handle command line parameters

  - initialize the rendering data structure and create the display window

  - intializes rendering defaults and specify the functions to use to handle
    keyboard input and display rendering

  - initialize the object model

  - start the interactive application and wait for user input.
    glutMainLoop() does this and uses the registered functions to handle
    rendering and user input. You only need to call glutMainLoop() if you
    are creating an interactive application that processes user input. This
    function will retain control (control never returns from glutMainLoop()
    unless the application exits or crashese) and call the various callback
    functions, registered using glutDisplayFunc, glutKeyboardFunc and
    glutIdleFunc, when the appropriate event occurs.

userKeyboardFunc0 function does the following

  - handle various keyboard presses either explicitely or by using the
    default keyboard handler rgl.Keyboard(Key,x,y);

userIdleFunc0 function

  - does a random walk of the 3D model.  call userDisplayFunc0() to render
  - the display at every step of the walk

userDisplayFunc0 function

  - is responsible for rendering the display everytime the model changes
    - usually because the user navigates the model

  - this function is called everytime something happens that requires the
    display to be re-rendered.

  - this function first checks the resolution to render the model and sets
    the number of polygons appropriately

  - it then extracts all the values needed to render the display (as
    mentioned at the top of this file)

  - it then sets the size of the rendering window

  - it then sets the camera and perspective positions properly

  - it then renders the model

  - displays the rendered data on screen
```

Figure D.3: GLVU Readme File Page 1 / 3

glvu_README.txt   Wed May 24 18:20:59 2006   2

```
the procedure contains documentation that further explains the above points.
Overall, userDisplayFunc0 calls a bunch of minor routines to set up the
rendering engine and then calls Obj->Display() to actually render the model.
It then displays the rendered 2D image (created from the 3D model) onto the
display. In the new version of the code (this version is the new version),
Obj->Display () automatically displays the rendered image onto the screen.
Previously, further calls needed to be made to extract the rendered pixels
from the rendering buffer and display them on the screen. This old code has
been commented out in the procedure (it is still available in case it is
ever needed again).


To render the 3d model, you need the following:

a) The name of the model (to load the model file)

b) The dimensions of the model

c) The dimensions of the rendering window

e) The number of polygons being rendered

f) The Currrent viewpoint of the viewer. This is expressed as
   three different vectors (x,y,z values).

   Eye (The x,y,z position of the viewers eyeball).

   CentrePoint (The x,y,z position of the center of the model. This value
                never changes for a given model)

   Up (The x,y,z matrix for going up in the world. This is constant and
       the matrix is (0,1,0)).

g) The perspective of the model. This is expressed as 4 different floating
   point values (not vectors)

   Fov    :- The field of view
   Aspect :- The aspect of the model
   Near   :- The near rendering point of the model
   Far    :- The far rendering point of the model



GLVU renders objects using two different data structures. The
ObjModelMultiRGL data structure contains information specific to the model
being rendered (the name, dimensions, number of polygons).

The RGL data structure contains information specific to the rendering
environment (the current viewpoint and the perspective).

Each of these data structures is explained in more detail below

ObjModelMultiRGL
----------------

    The Object model is initialized by doing

1)  ObjModelMultiRGL(const char *tmp_filename, RGL *tmp_rgl);

    where tmp_filename is the name of the model and tmp_rgl is a pointer to
    the rendering data structure.

2)  Obj->GetNumTris()
    returns the total number of polygons in the model

3)  Obj->GetNumVisible()
    returns the number of visible polygons that will be rendered. This is
    calculated from the total number of polygons and the current camera position
    and perspective by the rendering engine

4)  Obj->SetVisible(int)
    will set the number of visible polygons to render

5)  Obj->SetVisible(float)
    will multiply the current number of total polygons (obtained using
    Obj->GetNumTris()). GetNumVisible returns only the
    *visible* polygons) to render by the value of the float. The float should
    thus have a range of between 0.0 and 1.0

6)  Obj->getfilename()
    returns a char * to the name of the object

7)  Obj->Min
    the Vec3f value of the minimum position of the model.

    Vec3f is defined as
    typedef Vec3<float> Vec3f;

    and contains 3 floating point variables called x, y and z.

    for example,

    Vec3f tmp;

    tmp.x = 1.0;
    tmp.y = 1.0;
    tmp.z = 1.0;

    The Vec3f class is defined in math/vec3f.hpp and contains many nice
    operator overloads. In particular you can do

    Vec3f Min = Obj->Min
    and the 3 elements of Obj->Min (Obj->Min.x, Obj->Min.y, Obj->Min.z) will
    be copied into the 3 element of Min
    Obj->Max;

8)  Obj->Max
    the Vec3f value of the maximum position of the model.

9)  Obj->Display()
    render the model (using the current camera and perspective specified
    in the RGL data structure).
    This call must be preceded by a

    rgl.BeginFrame();

    which will set the rendering engine with the values of the cameras and
    perspective.

    after calling Obj->Display(), you must call
    glFlush ();     /// to flush all GL buffers
    rgl.EndFrame(); /// to actually display the rendered image on screen

    The call sequence is thus

    rgl.BeginFrame();
    Obj->Display();
    glFlush();
    rgl.EndFrame();


RGL
---

  The RGL data structure is initialized as follows

1) rgl.Init("GLVU Basic Example",
            GLUT_DOUBLE | GLUT_DEPTH | GLUT_RGBA,
```

Figure D.4: GLVU Readme File Page 2 / 3

glvu_README.txt   Wed May 24 18:20:59 2006   3

```
            50, 50, width, height);

  "Glvu Basic Example" is the name of the display window
   GLUT_DOUBLE | GLUT_DEPTH | GLUT_RGBA are options that should be specified this w
ay
   The next four parameters are the lower x, lower y, upper x, upper y
   positions of the display window. Thus in this example, the display window
   will be called "Glvu Basic Example" and have a size of width by height
   (since the lower x,y coordinates are 0,0).


   Camera and Perspective Settings
   -------------------------------

2) rgl.GetCurrentCam()->GetLookAtParams(&Eye, &ViewRefPt, &ViewUp);

   returns the current camera positions of the users eye position (in &Eye),
    the center of the model (in &ViewRefPt) and the vector to go up in the world
   (in &ViewUP). Each of the 3 arguments is a pointer to a Vec3f data type.

3) rgl.GetCurrentCam()->GetPerspectiveParams(&Yfov, &Aspect, &Ndist, &Fdist);

   returns the current perspective in 4 floating point variables (not Vec3f
   vars). Yfov is the field of view, Aspect is the aspect ratio, Ndist is
   the near rendering point and Fdist is the far rendering point.

4) rgl.SetAllCams(Min, Max, Eye, LookAtCntr, Up, Yfov, Aspect, Ndist, Fdist);

   set the camera and perspective to the values specified in the 5 Vec3f
   vars (Min, Max, Eye, LookAtCntr, Up) and the 4 float vars (Yfov, Aspect,
   Ndist, Fdist).  Min and Max are the minimum and maximum values of the
   model being rendered.


5) rgl.GetCurrentCam()->LookAt(Eye, LookAtCntr, ViewUp);
   to set just the camera position

6) rgl.GetCurrentCam()->Perspective(Yfov, Aspect, Ndist,Fdist);
   to set just the perspective

   The full list of prototypes offered by the RGL data structure can be
   found at glvu/rgl.hpp and glvu/glvu.hpp (since RGL inherits GLVU)

   The functions listed above are the minimum you will need to set the
   cameras and perspective for a simple application.
```

Figure D.5: GLVU Readme File Page 3 / 3

gocr_README.txt   Wed May 24 18:20:58 2006   1

```
gocr is an application that performs character recognition on image files.

it has a number of command line parameters.

run ./gocr -help to see what these are.

To run gocr with it's basic settings, do

./gocr <image_file>

the recognized characters found in the image_file (.pcx format only is
supported) are printed on stdout.

gocr has the following main functions (all found in gocr.c with 1 exception):

process_arguments :- this procedure processes the command line arguments

mark_start :- checks that the data structure used to perform the recognition
              (data structure is explained in more details below) exists. prints
              debugging information if the verbose option is enabled

mark_end :- ends the recognition. prints debugging information if the verbose
            option is enabled

read_picture :- calls the readpcx function to read the input .pcx image into
                the data structure.

pgm2asc :- This is the function that performs that actual OCR recognition.
           This function is found in pgm2asc.c

print_output :- prints the characters found in the image (the character list
                is stored in the data structure) to stdout.


main :- initialize the data structure, processes the command line arguments
        and then performs the recognition by calling the following functions
        (in this order).
        mark_start(JOB);   /* print debugging information if debugging is set */

        read_picture(JOB); /* read provided picture into job_t data structure */

        /* call main recognition loop */
        pgm2asc(JOB);      /* perform the recognition. This function calls many oth
er functions
                              the recognized characters are stored in the job_t dat
a structure */

        print_output(JOB); /* takes the data structure and prints out the recognize
d
                              characters to stdout
   */

        mark_end(JOB);     /* print ending debugging information */

       job_free(JOB);     /* clear data structure and reset it to accept another pi
cture file */


Gocr Data Structure
-------------------


Gocr stores all its data in a job_t data structure. There is only one
instance of this data structure for the entire recognition.
A global pointer to this data structure is declared in gocr.c as
job_t *JOB;

The actual data structure is declared in gocr.c (in main()) as
job_t job;

This data structure is defined as follows

/* job_t contains all information needed for an OCR task */
typedef struct job_s {
  struct { /* source data */
    char *fname;  /* input filename; default value: "-"  or stdin*/
    pix p;        /* source pixel data */
                  /* pixelmap 8bit gray */
  } src;
  struct { /* temporary stuff, e.g. buffers */
#ifdef HAVE_GETTIMEOFDAY
    struct timeval init_time; /* starting time of this job */
#endif
    pix ppo; /* pixmap for visual debugging output */

    /* sometimes recognition function is called again and again, if result was 0
       n_run tells the pixel function to return alternative results */
    int n_run; /* num of run, if run_2 critical pattern get other results */
               /* used for 2nd try, pixel uses slower filter function etc.*/
    List dblist; /* FIXME jb: desc */
  } tmp;
  struct { /* results */
    List boxlist; /* FIXME jb: desc */
    List linelist; /* FIXME jb: desc */

    struct tlines lines; /* used to access to line-data */
                         /* here the positions (frames) of lines are */
                         /* stored for further use */

    int avX,avY; /* average X,Y (avX=sumX/numC) */
    int sumX,sumY,numC; /* sum of all X,Y; num chars */
  } res;
  struct { /* configuration */
    int cs; /* FIXME jb: grey level 0<160<=255; default value: 0 */
            /* critical value (pixel<cs => black pixel) */

    int spc; /* spacewidth/dots (0 = autodetect); default value: 0 */
    int mode; /* operation modes; default value: 0 */
            /* operation mode (see --help) */
    int dust_size; /* dust size; default value: 10 */
    int only_numbers; /* numbers only; default value: 0 */
    int verbose; /* verbose mode; default value: 0 */
                 /* verbose option (see --help) */
    FORMAT out_format; /* output format; default value: ISO8859_1*/
    char *lc; /* list of chars (_ = not recognized chars); default value: "_" */
    char *db_path; /* pathname for database; default value: NULL */
    char *cfilter; /* char filter; default value: NULL */
  } cfg;
} job_t;

this data structure consists of structs within a struct. accessing values in
this structure is done in the following way.

assuming JOB points to this data structure.

accessing the spc variable would be done as

JOB->cfg.spc

changing the input filename would be done as

JOB->src.fname = new_name;

etc.
```

Figure D.6: GOCR Readme File

```
xspeech_README.txt       Wed May 24 18:20:59 2006       1
xspeech is a C application that uses that Motif Windowing library to record
voices into wave files. It then calls the janus libraries to recognize the
text in the wave files and display what was spoken as a text file to the
user.


Motif requires xspeech to use an event callback execution model. I.e.,
xspeech creates buttons and actions associated with the buttons. Motif will
draw the buttons on the screen and wait for user input. When this input
happens, Motif will call the application function associated with the button
that was pressed.

The heart of xspeech is the Recognize_Callback function. This function is
called when the user decides to record audio and recognize it (into text).

This function is located in speech_dialog.c

It updates various buttons (to indicate that work is being done).  It then
calls record_audio to record (in .wav format) spoken audio into a buffer.
This buffer is then saved to a file on the disk.

it then calls recognizeFile to recognize the wave file and return the text
translation.

This translation is then displayed to the user (or an error message if
recognizeFile failed).


recognizeFile (also in speech_dialog.c) does the following

sets the flag to do ADC to HYP recognition.

There are 2 ways to do recognitions

take the .wav file (this is the ADC file) and convert it directly to a text
file using ADC to HYP recognition.

The other way is to convert the ADC file into an intermediate format called
MEL and then convert MEL into the final HYP recognition.

i.e., you do ADC to MEL and then MEL to HYP.

refer to the janus README file for more information.
```

Figure D.7: XSpeech Readme File

janus_README.txt   Wed May 24 18:21:03 2006   1

Janus is application, written in Tcl/Tk, that accepts .wav file inputs and recognizes the text spoken in those .wav files.

Tcl/Tk uses an event callback model where you register callback functions to handle specified events. When those events occur, the tcl interpreter will call those functions.

the main program for Janus is located in JRTK/main.c

and contains

```
#ifndef DISABLE_TK
  Itf_Main (argc, argv, Tcl_AppInit);
#else
  Tcl_Main (argc, argv, Tcl_AppInit);
#endif
```

by default, this will execute Itf_Main (DISABLE_TK will not be defined)

Itf_Main (in JRTK/itfMain.c) does the following

it handles commands line arguments and sets up the necessary variables and conditions needed to initialize Tcl and Tk. It then call Tcl_AppInit when it exits.

Tcl_AppInit (found in JRTK/main.c) does the following.

It initializes the Tcl and Tk libraries. It then initializes the Janus recognition libraries (Itf_Init and Janus_Init). It then initializes the Server (Server_Init) that processes input from clients. Finally, it tells tcl to read the .janusrc config file (this may not exist).

JRTK/odyitf.c contains the routines that handle client input.

Server_Init is called by Itf_Main and calls

```
Tcl_CreateCommand (interp, "server", ServerProc, 0, 0);
```

This command basically tells Tcl to run the ServerProc function if the Tcl interpreter recieves the "server" command. For Janus, this command is sent to the Tcl interpreter at startup (the command is in a .tcl script read at startup). Hence, the ServerProc function will always be called.

This function calls Janusserver.

Janusserver is a function that does the following

- reads client input (the wav file to recognize, the model to use and flag indicating what to do to the inputs).

- the value of the flag argument determines which of 5 possible things should be done.

  if the flag is 0, Janusserver will call the JanusADCToHYP function to completely reognize the text in the wav file. The output is a string containing the recognized text.

  if the flag is 1, Janusserver will call the JanusADCToMEL function which takes a .wav file and converts it into an intermediate MEL format output file.

  if the flag is 2, Janusserver will call the JanusMELToHYP function which takes a MEL file and recognizes the text in the file. This function must be provided a MEL file and not a normal .wav file. The reason for using the 2-stage process of ADCToMEL and then MELToHYP (instead of always doing ADCToHYP) is that MELToHYP is less computationally intensive than ADCToHYP and ADCToMEL is relatively cheap computationally. Hence, by separating the ADCToHYP process into 2 stages, it is possible to achieve better throughput in cases where you have more than one Janus server and some servers are computationally weaker than others. The weak servers can be used to do ADCToMEL conversions while the stronger servers are used for MELToHYP recognitions. However, if all servers are equally capable, breaking the recognition into 2 stages just increases overhead (you have to transfer the MEL file back and forth between the client and the servers). It is better, in these cases, to just use ADCToHYP.

  if the flag is 3, Janusserver will load a particular model into memory. This speeds up the recognition time if that model is used later. This function is not normally used and is provided as a performance optimization. if MELToHYP, ADCToMEL, or MELToHYP are called with a model that is not in memory, that model will be loaded automatically.

  if the flag is 4, Janusserver will unload a particular model from memory. Again, this function is provided as an extra performance optimization option.

- the output is then sent back to the client. This is a string containing the recognized words for ADCToHYP and MELToHYP. The ADCToMEL returns the MEL file itself to the client. This is a binary file. LoadModel and UnloadModel return short strings containing error/success codes

- this function loops forever.

The various commands, MELToHYP, ADCToMEL, MELToHYP etc. are sent to the Janus library as Tcl commands. These commands are then interpreted by Tcl and the appropriate Janus functions are called (which perform the actual operations). These functions are registered by Janus in the Tcl_AppInit function call. The Itf_Init and Janus_Init calls register the functions that should be called when Tcl receives commands to process ADC files etc. This is similar to how Tcl is told to call ServerProc when it receives the "server" command.

Refer to JanusADCToHYP, JanusADCToMEL, JanusMELToHYP, JanusLoadModel and JanusUnloadModel to see the actual Tcl commands sent to the interpreter.

Figure D.8: Janus Readme File

mrclient_README.txt Wed May 24 18:20:54 2006 1

```
The Music Recognizer client is written in Java (to be platform independent)

what this program does is it allows clients to determine if a particular
piece of music is similar to other pieces of music (or whether it is the
same piece of music). It does this in the following way. It allows uses to
record a sample of music. It then sends this sample to a server to be
recognized. The server will return a list of songs that best match the
sample. The client will then display this list to the user. The client also
allows the user to playback the music.

The main routines for the client are found in RetrieveMusicApp.java (in the
musicretrievaldemo subdirectory).

This entire file contains the RetrieveMusicApp which is the main class for
the client.

The class works as follows

public static void main(String args[])

initializes the client using the command line arguments provided by the
user. in particular, the client is run as follows

java RetrieveMusicApp <list of wave files> [mrserver_name] [mrserver_port]

<list of wave files> (the only mandatory argument) is a filename of a file
containing a list of songs. This is used to provide more information to the
user about songs that were matched by the server. E.g.,

java RetrieveMusicApp wavlist.txt

mrserver_name is the name of the server (set to localhost by default)

mrserver_port is the port the server is running on (set to 2000 by default)


the main routine then calls

new RetrieveMusicApp(songlistfn, name, port).setVisible(true);

to start the application with the provided command line arguments
(songlistfn is the filename containing the list of songs).

This pops up a gui that allows the client to record a new song sample, to
recognize a recorded sample and to play back samples.

The sequence of steps is as follows. The user presses a button to record a
sample, the recorder function is called (this is in another file) and the
recording begins. When the recording stops (uses presses another button),
the doneRecording function (in the same file) is called and this takes the
recorded music and calls the searchDB function which sends it to the server
to be recognized. The doneRecording function than takes the output of
searchDB (an array of songs the sample matches) and displays it.


The heart of the recognize functionality of the client is in

private Vector searchDB(byte [] buffer, int freq)  (also in the same file).

This function is called by

public void doneRecording(byte [] buffer, int freq) {

and does the following

- opens a socket to the server

- sends the sample buffer and the frequency (all inputs to searchDB) to the
  server

- receives the reply from the server and extracts the individual song
  entries from the reply. The SongData class (also in the same file) has the
  code necessary to understand the output sent by the server.

- it then returns the array of song entries to be further processed.
```

Figure D.9: Musicmatch Client Readme File

mrserver_README.txt   Wed May 24 18:20:54 2006   1

```
mrserver.cc  is the main program for the music recognizer server backend.

broadly, this application tries to decide if a particular piece of music
matches other pieces already in a directory. It does this by using FFT
filters combined with "keypoints" technology. This is where the server
stores points of interest from other pieces of music and compares these
points with the provided piece of music. Based on how many points match, the
server predicts that the provided piece of music matches with other pieces
of music in its database.

The server works as follows:

in the main program, it

- initializes based on the command line parameters

- reads the EM parameters

- initializes the filters

- reads in the keypoints database from disk. this database contains the
  points of interests for many other songs.

- builds a hash table of the keypoints for first searching

- after all these initialization steps are done, the server calls
  mainloop ( )  which does the actual work


mainloop () (also found in mrserver.cc) does the following

- opens a listening fd to listen for connections from clients

- does the following in an infinite loop

- accepts input from clients. This input is a packed binary character buffer
  of the following form

  1st integer in buffer (1st 4 bytes)   ->  number of bytes in the music
                                            sample being sent. This is
                                            stored in the nbytes variables.
                                            The nsamples variable is set to
                                            nbytes/2 to denote the unique
                                            samples in the music (divide by
                                            2 since we assume stereo
                                            samples). The endianness of this
                                            integer is swapped before it is
                                            read.

  2nd integer in buffer (next 4 bytes)  ->  frequency of the music piece.
                                            this byte will need to have its
                                            endianness swapped. This is
                                            stored in the freq variable

  remainder of bytes                    ->  the music sample itself. the 1st
                                            integer in the buffer refers to
                                            the length of this part of the
                                            buffer. It doesn't include the
                                            length of the first 2 integers
                                            in the buffer. These bytes are
                                            stored in the Sample array

 refer to ReadFromSocket () (also in mrserver.cc) for the actual details.

- takes the samples sent by the client and creates a bit pattern using
  wav2bits ( ).

- creates a set of keys (used in the search) from these bits using bitsTokeys ( )

- then it does the actual searching for song matches. It does that using the
  following four steps

  - finds all possible matches using FindMatches ( )

  - filters the matches to remove obvious errors using FilterMatches ( )

  - performs a verification step on the filtered matches (this is further
    filtering) using verify4em ( )

  - picks the final list of matched songs using chooseBestSong ( )

- the keys used for the matching are then deleted

- and the matched songs (chosen by the final chooseBestSong ( ) step)
  are sent back to the client using writeResults ( )

  The format of the data sent back to the client is as follows:

  This function effectively writes back a string to the client. The string
  is formatted in the following way (which the client knows how to decipher)

  Data written

  Bytes  DataType  Descr
  4      int       Number of songs

  for each song:
  11     char      cddb id, underscore, track #.  Example "1234567A_08"
  4      int       db start frame
  4      int       db end frame
  4      int       query start frame
  4      int       query end frame
  4      int       string length
  x      char      string (name of song)


  refer to writeResults ( ) for the exact code to send data back to the
  clients. Note that every time we send an int, we use writeInt ( ) which
  actually swaps the endianness of the integer. This is a convention we use
  between the client and the server to ensure that we don't run into
  problems when using this code on different hardware. Effectively, it is
  doing what ntonl ( ) and htonl ( ) would do.
```

Figure D.10: Musicmatch Server Readme File

panlite_README.txt Wed May 24 18:21:02 2006 1

```
Panlite Readme
--------------

Panlite is a natural language translation application that is able to
convert english sentences into spanish and vice versa.

It works as follows: sentences provided by users are processed by 1 to 3
translation engines. These engines are the dictionary, glossary and example
based machine translation (ebmt) engines. The results from these engines are
fed into a language modeller stage that combines the various results
together and creates the final translated sentence.

The language modeller must be provided with output from at least one of
these engines. Providing it with the output of more engines results in
higher quality translations. However, using more engines requires more time
and cpu/memory resources. Each of these engines can be run independently
from the others.


Application Guide
-----------------

panlite is comprised of 4 main directories (each of them has a src/
subdirectory containing the actual C++ zsource files)

framepac
  |
  |----> src/   contains the src code for the framepac library.
                This library contains the routines needed to handle
                word tokens. The line to be translated is split into
                individual word tokens and the routines in this
                library allow these tokens to be easily manipulated.

ebmt
 |
 |-----> src/   contains the src code for the translation library.
                This library contains the routines needed by the
                dictionary, glossary and ebmt translation engines.

lm
 |
 |-----> src/   contains the src code the language modelling library.
                This library takes the output from the translation
                engines and creates the final translated sentence.

panlite
 |
 |-----> src/   contains the src code for the main application.
                the main routine is found in panlite.C


Panlite can be run in many modes. However, the only mode that really works
at the moment is the normal mode. The batch mode and network modes are still
under testing.

The main routine of panlite is found in panlite/src/panlite.C
It does the following:

- initialize the libraries
- process the command line input
- initialize the translation engines
- call process_Panlite_input to process the input


process_Panlite_input is found in panlite/src/plmain.C
- read the sentences provided as input
- extract individual sentences from the input
- send each sentence to Panlite_process_line to be translated

Panlite_process_line is also found in panlite/src/plmain.C
- checks if the sentence provided is valid
- if it is, calls translate_sentence to perform the translation

translate_sentence is also found in panlite/src/plmain.C
- There are 2 prototypes for this function.
- Panlite_process_line uses the char *translate_sentence(const char *line)
  prototype
- this translates each line by sending it to the three translation engines
  and then sending the 3 outputs to the language modeller (which creates the
  final translated output)
- the functions called are
  - do_gloss_translation
  - do_dict_translation
  - do_ebmt_translation
  - do_lm_modelling
- these functions are all found in panlite/src/plchr.C
- the translation functions take a sentence as input and
  returns a partial result
- the modelling function takes 1 to 3 partial results (some can be NULL)
  and returns a translated sentence

Currently, this function uses all 3 translation engines. However, it is
possible to envision cases where it might be better (to conserve resources
on limited hardware for example) to only use 2 or even 1 of these
translation engines only. Although we have taken the first step by allowing
the modelling function to accept NULL inputs from any particular translation
engine, full support for this feature will only appear in future versions of
this software

Data Structures
---------------

The engine functions and the modelling function (do_gloss_translation,
do_dict_translation, do_ebmt_translation and, do_lm_modelling) use an FrList
structure internally to store the results of the various translation engines
and the language modeller. However, they all return a simple character
buffer to the calling "main" function translate_sentence.

This structure is large and contains many other inherited classes. It's
definition is found in framepac/src/frlist.h
```

Figure D.11: Panlite Readme File

```
radiator_README.txt       Wed May 24 18:20:53 2006       1

radiator is a library that calculates lighting for 3D models. It takes as
input a 3D scene file and produces rendered images of the 3D model. For each
rendering, you can change the resolution and algorithm to use for the
rendering.

an example program, called RadMain.cc, is available in rad_client/src/

the main program processes the command line options and then calls the main
routine, app.DoRadiosity();

app.DoRadiosity() (also found in RadMain.cc) does the following

- it reads the provided scene object file

- it sets up the matrixes used for the lighting models

- it decides what resolution and algorithm to use for the lighting (these
  are currently set to default values)

- it then calls the library functions to render the scene with the new
  lighting

- it saves the rendered scene to a new scene object file

If you need to load the new object files, you can do so using

        path.SetPath(sceneFile);
        scene = SceneReader::Load(path);

as used at the top of app.DoRadiosity
```

Figure D.12: Radiator Readme File

D.5 Application-Specific Domain Expert Information

In this section, the domain information for each application is presented. This is the domain information that was given to the participants. The documentation can be found as follows: Face (Figure D.13), Flite (Figure D.14), GLVU (Figure D.15), GOCR (Figure D.16), Janus (the XSpeech client (Figure D.17) and the Janus server ((Figure D.18)), Musicmatch (Figure D.19)), Panlite (Figure D.20), and Radiator (Figure D.21).

```
face_domain_README.txt       Wed May 24 19:14:37 2006       1

For adaptive face detection programs, the size of the image in which faces
are to be recognized fundamentally determines the resource usage of the
application.
```

Figure D.13: Face Domain Information File

```
flite_domain_README.txt       Wed May 24 18:20:54 2006       1

For adaptive speech synthesis programs, the size of the text file being
synthesized fundamentally determines the resource usage of the
application.

the easiest way to make these kind of programs remotely executable is to
just send the text file to synthesize to the server and then receive the
synthesized wav file at the client side.
```

Figure D.14: Flite Domain Information File

```
glvu_domain_README.txt       Wed May 24 18:20:58 2006       1

For 3D modelling applications, the following are important for making them
adaptive.

The values that affect the resource usage are

1) The name of the model being rendered.

2) The height and width of the rendering window

3) The size of the model being display (usually specified as the minimum and
maximum coordinates of the model).

4) The number of polygons in the model that will be rendered.

5) The current viewing position of the model. This is usually expressed as a
combination of 2 or 3 different values. Each value will usually be expressed
as a x,y,z coordinate.

6) The current perspective of the model. This is also usually expressed as a
combination of multiple values. Refer to your specific application for
details on how it stores the current viewing position and perspective.


If there is an adaptive runtime system, the 3D application could obtain the
appropriate resolution to render the model at from the system dynamically.
This resolution will then be used to decide the exact number of polygons to
render.


For adaptive systems that perform remote execution, you can partition 3D
modelling applications as follows:

The client will receive the user input, figure out the rendering window
size, camera position, pespective, model name, model size and the number of
polygons to render. It will then send all this information to the remote
server. The remote server will then render the image and send a raw data
stream (containing the rendered pixels) back to the client. The client will
then take these rendered pixels and display them on the screen. This allows
the client to use a server to perform the computationally intensive task of
rendering the 3D model for the current viewing position and perspective.
```

Figure D.15: GLVU Domain Information File

```
gocr_domain_README.txt       Wed May 24 18:20:57 2006       1

For adaptive OCR programs, the size of the image being recognized
fundamentally determines the resource usage of the application.
```

Figure D.16: GOCR Domain Information File

```
xspeech_domain_README.txt       Wed May 24 18:20:59 2006       1

For adaptive speech detection programs, the size of the wave file being
recognized affects the resource usage.
```

Figure D.17: XSpeech Domain Information File

```
janus_domain_README.txt       Wed May 24 18:21:02 2006       1
For adaptive speech detection programs, the size of the wave file being
recognized affects the resource usage.

the main operation for speech recognizers is the code that takes a wav file
as input and produces recognized text as input (It could use 1 or more
intermediate stages to produce the text file)

speech recognizers also have a secondary operation that is used by clients
to load/unload models on the server (for improved performance). These
functions should only be supported as a secondary optimization (you can
specify more than one OPERATION block in a .input file. Each separate
OPERATION block must have a full set on IN, OUT, RPC and TACTIC
specifications appropriate to that operation. You should start a new
OPERATION block only after defining everything required for the previous
OPERATION). For loading/unloading the model, there are no parameters or
fidelities for this operation. This is because this operation will be
manually called by the client when it is necessary (by calling the
do_tactics call with the appropriate arguments). As such, only specify the
RPCs and TACTICs for this secondary operation. Refer to the janus server to
see the functions needed to perform loading and unloading of models.

When modifying the client, focus only on the main operation (i.e.,
converting speech to text). You do not need to add code to handle the 2nd
operation. On the server side, however, you must be able to handle any RPC
(for all OPERATIONS). The process is similar to the case with only one
OPERATION. The only difference is that you will need to create wrappers for
more RPC functions (the set of RPCs for both OPERATIONs). Everything else
remains the same.
```

Figure D.18: Janus Domain Information File

```
mrserver_domain_README.txt       Wed May 24 18:21:01 2006       1
for music recognition / categorization applications, the number of samples being
categorized and the frequency of the music being categorized determine the
resource usage.


Many of these kinds of applications already use a client / server
architecture. Adding these applications to an remote execution framework is
not too difficult. Replace the client socket calls with the API calls for
the runtime you are targetting (you should be sending the same data that
went over the socket to the runtime API calls).

On the server side, the server would not need to open any listening sockets
or accept connections anymore. Instead it receives its input from the
runtime. On the output side, instead of writing data directly to a socket,
you write it into the buffer provided by the runtime code. Assume that this
buffer is large enough. Once you have written the data into the buffer
correctly (i.e., in the same order it was written to the socket), the
runtime will send the buffer directly to the client for you. The server
should also send the number of songs in the buffer separately to the client
so that the client can process the buffer accordingly. This number of songs
value can be stored as the first integer in the buffer also. How it is done
is left up to the individual applications.

On the client side, the client will receive all the output in one go
(instead of possibly receiving it piecemeal). This shouldn't be a problem
for most clients and you can reuse whatever client code exists to decipher
the buffer sent by the server (since you havn't changed the contents of the
buffer. you've only changed the mechanisms used to transfer the buffer
between the client and the server). you should also reuse existing client
code to create the buffer to send to the server. The only difference is that
you don't write the buffer to a socket. Instead, you write it to a buffer
instead. On the server side, you should preserve any code that already
exists that is able to take inputs from the client and decipher it. The only
difference is that this code will not be receiving data from a socket. It
will receive it from a buffer provided by the runtime.

When deciding how to partition these kinds of applications for remote
execution, there are many possible ways to do it.

One way is to make every possible data mining step (these application
usually do a lot of machine learning steps on the data to achieve the final
result) a separate remote function. The disadvantage of this is that you
will need to be able to send all the intermediate data between the remote
servers.

Another way is to use one remote procedure and simply let the server do
everything. i.e., you give it the raw samples, it does a whole bunch of
steps on this and then it returns the results. This might be the simplest
solution if the intermediate data produced by the the categorization
functions are complication while the initial input to the categorization
functions is simple and the final output from these functions is also simple

There is of course, the middle ground where you make some of the data mining
steps a separate remote function and combine others together in one
function. The exact decision of what to do depends on the actual
application. If it is simple to send intermediate values between servers,
then creating separate functions for the steps with easy to manage output is
encouraged. Otherwise (if the data sent between stages is complicated), then
just take the easy way and put everything in one big remote procedure.
```

Figure D.19: Musicmatch Domain Information File

```
panlite_domain_README.txt        Wed May 24 18:20:57 2006        1

For adaptive language translation programs, the number of words in the
sentence being recognized fundamentally determines the resource usage of the
application.
```

Figure D.20: Panlite Domain Information File

```
radiator_domain_README.txt        Wed May 24 18:20:55 2006        1

For 3D radiosity applications (applications that calculate the lighting for
3D models), the following parameters affect the resource usage.

- the number of polygons in the 3D model being used

if there is an adaptive runtime system, the application can benefit by
knowing the appropriate values for the following quantities.

- the resolution to use to calculate the new lighting for the model

- the algorithm to use when calculating the new lighting
```

Figure D.21: Radiator Domain Information File

D.6 Tactics File Documentation

D.6.1 Tactics File Creation Overview

Creating a tactics file requires the following step.

1. Read the application-specific documentation and understand what the parameters and fidelities of the application are. Get a sense of what the RPCs and TACTICs might be.
   (a) Fill out the Questionnaire labeled I1 after you are done
2. Create the APPLICATION tag
   (a) Fill out the Questionnaire labeled I2 after you are done
3. Create the OPERATION tag
   (a) Fill out the Questionnaire labeled I3 after you are done
4. Identify and specify all the parameters of the application using the IN tag
   (a) Fill out the Questionnaire labeled I4 after you are done
5. Identify and specify all the fidelities (if any) of the application using the OUT tag
   (a) Fill out the Questionnaire labeled I5 after you are done

6. Identify and specify all the remote procedures of the application using the RPC tag
   (a) Remember, you can use the *string* data type for packed data because the stub will automatically generate a length variable for every *string* variable. Refer to the *RPC* section of the documentation for more information.
   (b) Fill out the Questionnaire labeled I6 after you are done
7. Specify the tactics of the application using the TACTIC tag
   (a) Fill out the Questionnaire labeled I7 after you are done

D.6.2 Tactics File Creation Manual

A tactics file has 6 components. They are denoted by the following keywords.

APPLICATION
OPERATION
IN
OUT
RPC
TACTIC

Each of these keywords is explained in more detail below.

D.6.2.1 APPLICATION

The *APPLICATION* tag is used to specify the name of the application. For example, for an application called *foo*, you would specify

```
APPLICATION foo;
```

in the tactics file

Note that the *APPLICATION* line *must* be the top line in the tactics file and that the line must be terminated by a semicolon (;) (like C and C++).

D.6.2.2 OPERATION

The *OPERATION* tag is used to specify the operation that this application performs. An operation can be thought of as the functionality that the application provides. For example, language translators would provide the *translate* operation, speech recognizers *recognize* and image recognizers *recognize*. Assume that the application called *foo* is a graphics application that renders 3D models to screen. You would thus specify

```
OPERATION render;
```

in the tactics file

Note that the *OPERATION* line *must* be the 2nd line in the tactics file (after the *APPLICATION* line) and that the line must be terminated by a semicolon (;) (like C and C++).

The next 4 tags can appear in any order in the tactics file. But usually, they appear in the following order. All the *IN*s followed by the *OUT*s followed by the *RPC*s and finally the *TACTIC*s.

D.6.2.3 IN

The *IN* tag is used to specify the **parameters** of the application. Parameters are variable settings of the application that affects its resource usage. Hence, the values of these variables needs to be conveyed to the underlying runtime system so that it can decide what resources the application will need to use.

For example, the resource usage of language translation applications is determined by the size of the sentence to be translated. 3D graphics applications, in particular, may have a number of parameters. These include the size of the rendering window, the number of polygons being rendered, the position of the viewing camera, and the current perspective of the 3D model.

IN parameters are specified as follows

```
IN <variable_type>  <variable_name>  <optional_parameters>
```

Arguments : The *variable_type* field is used to specify the data type of the variable. The following values are legal for this field.

int :- the variable is a signed integer
uint :- the variable is an unsigned integer
float :- the variable is a float
double :- the variable is a double
char :- the variable is a character
string :- the variable is a string (char * in C/C++)
ustring :- the variable is an unsigned string (unsigned char * in C/C++)

the *variable_name* is the name of the variable (as it appears in the application).

optional_parameters are used to specify the range of the variable. They should be used only when necessary. The examples section shows how optional parameters may be used.

Examples of use :

```
// This specified that the variable file_size is an integer var
// that has a range of 0 to 1000000
IN int file_size FROM 0 TO 10000000;

// The parameter is a string variable called filename. It has no
// specified range.
IN string filename;
```

Again, note that all the lines are terminated with a semicolon.

D.6.2.4 OUT

The *OUT* tag is used to specify the **fidelities** of the application. Fidelities are variables whose values should be set by the runtime system according to the current resource availability. I.e., the application doesn't set the values of these variables. Rather, the runtime system tells the application what these variables should be set to, to achieve maximum performance.

For example, a graphics application may want the runtime to tell it what the rendering resolution should be set to. In this case, the variable resolution will be a fidelity and should be specified using the *OUT* tag.

The syntax for fidelities is exactly the same as that for parameters. The only difference is that you use the keyword *OUT* instead of *IN* to specify that a variable is a fidelity and not a parameter. The *OUT* tag uses the same data types and optional parameters as the *IN* tag.

OUT fidelities are specified as follows

```
OUT <variable_type>  <variable_name>  <optional_parameters>
```

Note that not all applications will have fidelities. All adaptive applications should have at least one parameter though. Otherwise, the runtime system will not know how to predict what resources the application will need at any point in time.

Examples of use :

```
// the double variable resolution is a fidelity that can range
// from 0.0 to 1.0
OUT double resolution FROM 0.0 TO 1.0;
```

Again, note that all the lines are terminated with a semicolon.

D.6.2.5 RPC

The *RPC* tag is used to specify which procedures in the application can be remotely executed.

These procedures could be actual application procedures or even new procedures that accept all the inputs needed to call existing application procedures and that return the required outputs needed by the application. You may need to create a new function because no existing function has the exact set of inputs and outputs necessary to perform the operation. For example, for a graphics application, the RPC may be a procedure called *render* that takes as inputs the name of the model file, the number of polygons to render, the camera position, the current perspective and returns a string containing the rendered pixels. The server application will accept these inputs and call the appropriate application rendering function and then return the rendered pixels to the client (after extracting those pixels from the rendering buffers). The client will receive those pixels and display them on the screen (by calling the appropriate display function).

You specify the procedures using the following syntax:

```
RPC <procedure_name> (argument list);
```

The *argument list* is specified using the same data types as used by the *IN* and *OUT* tags. I.e., *int, uint, float, double, char, string, ustring*.

The *RPC* argument list contains an extra data type, called *FILE*, which is not available for *IN* and *OUT* tags.

The *FILE* data type is used to transfer data files between local and remote servers. If a procedure has a *FILE* variable, it will accept a *string* (*char* *) containing the name of a data file. This file will then be transferred to the remote machine. This file can contain arbitrary amounts of data. Thus, this is very useful mechanisms for transferring large amounts of data between local and remote machines.

For *FILE* and *char* * arguments, the stub generator will automatically create a length variable associated with this arguments. You do no need to explicitly specify any length variables for these two data types. You can thus use *char* * types even to send packed binary data as the stub generator will not perform string operations on the buffer. Instead it uses the lengths of the buffer as set by the associated length variable. For output arguments, the length of the output buffer is returned to the application. Refer to the actual stub generated *RPC* prototype to see the exact length arguments that need to be given to the stub generated *RPC* function.

There is no array type though. If you need to send an array, you must enumerate every array element individually. For example, instead of saying

```
RPC foo (IN int bar[3], OUT float out[3]);
```

You would do

```
RPC foo (IN int bar_1, IN int bar_2, IN int bar_3,
         OUT float out_1, OUT float out_2, OUT float out_3);
```

And pass in the individual array elements as inputs and recombine the 3 output floats in the application code to form the required array. We may add array types in the future if enough applications require it.

Each variable in the argument list needs to be pre-pended with either an *IN* or *OUT* tag. *IN* variables are inputs to the procedure while *OUT* variables are outputs of the procedure.

In the argument list, it is vital that all the *IN* variables are listed **before** any *OUT* variable.

The RPC argument lists should contain all the inputs necessary to perform the operation. This should include all the fidelity variables (or values obtained by using the fidelity values) and possibly some of the parameters. Not all the parameter values should be specified as some may not be relevant for the actual operation (like the size of the input. Instead, you should send the input itself).

The stub generator will generate the necessary code to ensure that all the variables specified in the argument list are sent correctly between servers.

Examples of use :

```
// procedure foo takes as input a string containing the name of
// the data file to be transferred. The} procedure returns its
// output in a string (char *) variable called buffer. The stub
// generated will automatically generate a length variable for
// buffer that returns the exact length of the data} returned in
// buffer by the remote server.
RPC foo (IN FILE name, OUT string buffer);

// it is okay to capitalize the types
RPC foo (IN double a, IN DOUBLE b, IN DOUBLE c, OUT int value);

// this is not allowed. OUT before IN
RPC foo (OUT int value, IN int type);

// the procedure reads a data file as input and uses a data file
// as output the stub generator will generate length variables
// associated with in and out
RPC foo (IN FILE in, OUT FILE out);
```

D.6.2.6 TACTIC

The *TACTIC* keyword is used to tell the system how to combine the *RPC*s of the application to perform useful work. For example, an application may have 4 *RPC*s (*a, b, c and d*), but not all combinations of doing these 4 *RPC*s may result in correct output.

The *TACTIC*s keyword allows you to specify that *RPC a* must be followed sequentially by *RPC b*. It also allows you to say that *RPC*s a, b and c can be done in parallel. The syntax is described using the examples below:

Examples of use : For these examples, assume that the application has 4 *RPC*s (specified using the *RPC* tag described above). These *RPC*s are a, b, c and d.

```
// do RPC a and then after it finishes, do RPC b. the & symbol
// is used to denote a sequential relationship.
TACTIC foo = a & b;

// do RPCs a, b and c in parallel, wait for all of them to
// finish and then do RPC d. any RPCs appearing in ( ) are to be
// done in parallel. A parallel stage *MUST* be followed by a
// sequential stage.
TACTIC foo = (a, b, c) & d;
```

We do not allow parallel stages to be followed by other parallel stages. Also that can be no further sequential or parallel stages inside a () block. I.e., the fanout inside a parallel stage is 0. For example,

```
// Illegal. Sequential stage inside a parallel stage is not
// allowed.
TACTIC foo = (a & b, c) & d

// also illegal. Parallel stage inside a parallel stage is not
// allowed.
TACTIC foo = (a, (b, c, d)) & a;

// is also illegal. No sequential stage after the parallel
// stage.
TACTIC foo = (a, b, c);

// completely legal
TACTIC foo = (a, b, c) & d;
```

More good examples:

```
// do only RPC a. This may look trivial but RPC a could be done
// remotely and this may greatly improve performance. The
```

```
// decision of where to do a is decided by the runtime. Many
// application only have 1 RPC (and only 1 TACTIC) that does all
// the computation.
TACTIC foo = a;
```

An application may have more than 1 *TACTIC*. This represents the case where the application has more than one procedure that does the computation necessary to support a computation and these procedures can be combined in different ways to do useful work. The different ways will result in different quality outputs but they will also have different resource consumptions. Hence, at runtime, the system will decide which tactic should be chosen that will provide the best quality while not exceeding any resource usage bounds set by the application.

For example, given the 4 *RPC*s a, b, c, and d, assume that the application processes inputs in the following way: the input is sent to one or more of procedures a, b, and c (in parallel) and the partial results from these procedures is sent to d which produces the final result. Given this, you could write 4 *TACTIC*s as follows

```
TACTIC do_a_and_d = a & d;

TACTIC do_a,b_and_d = (a, b) & d;

TACTIC do_a,b,c_and_d = (a, b, c) & d;

TACTIC do_b_and_d = b & d;
```

The complete *TACTIC* list for this application would contain 7 tactics (only 4 are shown) as a full enumeration of this pattern (do one or more of a, b and c in parallel and then do d) will result in 7 tactics in total.

Tactic Constraints : Other than the specification constraints (no fanout in a ( ) and a parallel step must be followed by a sequential step) listed above, the following two constraints must also be observed. These apply to applications that have more than one tactic.

1. For an application, the **1st** RPC of all its *TACTIC*s must have the same set of *IN* parameters. I.e., the 1st *RPC* of every tactic must have the same inputs. These inputs must also have the same names. If the 1st *RPC* is a parallel stage, every *RPC* in the parallel set must have the same set of inputs (that have the same names).

2. For an application, the **last** *RPC* of all its *TACTIC*s must have the same set of *OUT* parameters. I.e., the last *RPC* of every tactic must have the same outputs. These outputs must have the same names.

These requirements are necessary to ensure that the stub generator can generate a common *do_tactics ( )* procedure that will work for all of the application's tactics.

For example, given the following *RPC* definitions

```
RPC a (IN int input, OUT int partial_result);

RPC b (IN int input, OUT int partial_result2);

RPC c (IN int partial_result, IN int partial_result2,
       OUT int final_result);
```

The following *TACTIC* specifications

```
TACTIC do_a = a & c;

TACTIC do_b = b & c;

TACTIC do_c = (a, b) & c;
```

are perfectly okay because for every TACTIC, the input is *IN int input* and the output is *OUT int final_result*.

Note that *RPC*s a and b have different outputs and that RPC c has 2 different inputs to accept the different outputs of a and / or b. This is perfectly fine.

The example above also highlights the final tactic constraint:

3. If you want an output of a *RPC* to be an input of another *RPC*, the two variables (in the *RPC argument list*) **must** have the same type and name.

i.e., like the example above, if you want *RPC b*'s output to be an input in *RPC c*, *RPC b* must have an output variable with the same type and name as an input variable of *RPC c*.

By default, all arguments to an *RPC* are set to *NULL* or equivalent (0 for integers etc.). Hence, in cases where *RPC b* is not run (for example, tactic *do_a_and_d* is chosen which doesn't include *RPC b*), the input for *RPC c* that would have corresponded to the output of *RPC b* will be set to NULL. When creating the *RPC* functions, you must thus be prepared to accept *NULL* inputs.

This naming constraint arises because we wanted to keep the syntax simple. Hence, we decide which variables are passed between *RPC*s by looking at the variable names. If we didn't do this, we would need additional syntax to allow programmers to specify which variables are to be sent to which *RPC*s.

D.7 Retargeting the Application Documentation

D.7.1 Client Retargeting Overview

Modifying the client requires the following steps (there are 7 in total):

1. Read the application-specific documentation and identify the routines that need to be modified.
   (a) Fill out the Questionnaire labeled C1 after you are done
2. Including the stub generated client header files
   #include "foo_bar.chroma.h"
3. Fill out the Questionnaire labeled C2 after you are done
4. Register the application using the *register_app ( )* API call
5. Fill out the Questionnaire labeled C3 after you are done
6. Add the *cleanup ( )* call at the end of the application
7. Fill out the Questionnaire labeled C4 after you are done
8. Use the *find_fidelity ( )* call to tell the runtime that work is about to be done. This call will decide the appropriate fidelities and tactics to use for the current operation. This call requires the following sub steps.
9. Identifying the parameters, fidelities and tactics of the application
   (a) Setting the parameters before calling *find_fidelity ( )* by using the appropriate *set_var_name ( )* macro call.
   (b) Reading the fidelities after calling *find_fidelity ( )* by using the appropriate *get_var_name ( )* macro call.
   (c) Using the fidelity values to set actual application settings before doing the actual work.
10. Fill out the Questionnaire labeled C5 after you are done
11. Replace calls to the functions that perform the operation with a call to *do_tactics ( )*. This call will perform the operation using the selected tactic (either locally or remotely) and return the results to the client.
12. If necessary, use the results from the *do_tactics ( )* call. For example, the results may need to be displayed to the screen or the output may need to be printed to stdout.

13. Remember that you need to allocate enough space in the client for the outputs from *do_tactics ( )*.
14. Fill out the Questionnaire labeled C6 after you are done
15. Compile the client code and fix any compilation errors.
16. Fill out the Questionnaire labeled C7 after you are done

D.7.2 Server Retargeting Overview

Modifying the server requires the following steps (there are 7 in total):

1. Read the application-specific documentation and identify the routines that need to be modified.
   (a) Fill out the Questionnaire labeled S1 after you are done
2. Add the Stub generated server header files into the server code.
   (a) Fill out the Questionnaire labeled S2 after you are done

   #include "foo_bar.chroma.server.h"
3. add a call to *int service_init (&argc, &argv)* to initialize the application with the underlying runtime system. The runtime will remove any parameters that were runtime specific from the argument set. This has to be done before any specific application specific processing of the command line arguments.
   (a) Fill out the Questionnaire labeled S3 after you are done
4. Preserve any application specific initialization routines.
   (a) Fill out the Questionnaire labeled S4 after you are done
5. Create the RPC functions specified in the tactics file (if they do not already exist) and make sure that they can accept the inputs and return the outputs specified in the tactics file. All output variables will be created as global variables by the stub generator and a pointer to that variable passed to the RPC functions. You should not declare the output variables or allocate space for them. The actual RPC functions that need to be created (along with the arguments to these functions) will be shown in the *foo_bar.chroma.server.h* stub generated header file.

   for example, given the following application main routine and the following RPC specification.

```
RPC perform_work (IN int value, IN string position,
                  OUT FILE outfile);
```

You will have to ensure that the server has a function called *perform_work* with the following prototypes

```
void perform_work (int value, char * position,
                   int position_len, char * outfile,
                   int * outfile_len);
```

Remember that length variables are automatically generated for string and FILE variable types. The length variable will be passed as a pointer if it is an output value (i.e., it is a length associated with an OUT variable). You can see the exact RPC function prototypes that the stub code is expecting by looking at the *foo_bar.chroma.server.h* file.

6. Fill out the Questionnaire labeled S5 after you are done

7. Replace the rest of the application code with a stub generated (automatically) API call

   ```
   void run_chroma_server ( );
   ```

   This function will receive parameters and requests from the client, and will return outputs to the client.

8. The main routine of the application should just have

   (a) Call to *service_init ( )*

   (b) Application initialization routines that initialize any libraries or functions needed by the server to perform the operation.

   (c) *run_chroma_server ( )*

   (d) The existing code in the main routine that process any inputs and calls the routines that do the operation should not be run. Anything appearing after *run_chroma_server ( )* will not run as *run_chroma_server ( )* will loop forever (accepting client requests, calling the right procedures and returning results).

9. Fill out the Questionnaire labeled S6 after you are done

10. Compile the server code and fix any compilation errors.

11. Fill out the Questionnaire labeled S7 after you are done

D.7.3 Common Routines Cheat Sheet

This handout explained how to do certain common programming tasks that the participants may or may not need to know.

D.7.3.1 Common Programming Tasks

These are example C / C++h code snippets for various tasks that you may need to solve the problem given to you. Please note that you may not need to use any of these tasks to solve the problem.

D.7.3.2 To Determine the size of a File

use the *stat()* system call as follows

make sure the following 3 include files are included

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
```

In the code where you want to check the size of file, do this

```
int check_file_size (char * file_name) {

    struct stat buf;

    if (stat(file_name, &buf) < 0) {
        perror("error stating file");
        exit(1);
    }

    printf("The filesize of %s is %d bytes\n",
            file_name, buf.st_size);
}
```

D.7.3.3 Creating a temporary file

Frequently, servers will need to create temporary files to store results of computations.

this can be done using the *tmpname( )* command

```
char *tmpfile;

tmpfile = tempnam (NULL, "temp");
```

for example, this will make *tmpfile* point to a temp name that starts with temp (5 characters maximum) and be placed in the directory contained in the environment variable *$TMPDIR*. For more details,

man tempnam

D.7.3.4 Redirect Output to a File

There might be times when you need to redirect output to a file (instead of *stdout).*

There are two ways to do this

1. open a *FILE* pointer and use that in *fwrite*, *fputs* etc.
2. use *freopen(file_name, "w", stdout)* to map *stdout* to the file with the name given in *file_name*. After this, all writes to *stdout* will go to *file_name* instead.

Example

```
#include <stdio.h>
      .
      .
      .
FILE *fp;
      .
      .
      .
    fp = freopen ("/tmp/logfile", "w+", stdout);
      .
      .
    // this write will go to /tmp/logfile instead
    fputs (buffer, stdout);
```

D.7.3.5 Include C headers in a C++ program

You must put the include directive in an extern "C" { ... } statement (if the included .h file doesn't have the extern command already). The stub generated header files are all C header files.

For example,

```
extern ``C'' {
   #include ``foo_bar.chroma.h''
}
```

D.7.4 Programming Notes Handouts

This handout explained exactly how space allocation for input and output variables was handled by the stub. The allocation was different for the client and server stubs.

Allocating Space for Output Variables

1. When modifying the **client**, the space for all output variables for the *do_tactics ( )* API call must be allocated by the client (the stub code will not do this). This include string (char *) variables and FILE variables. In the case of FILE output variables, you must provide a valid string buffer that contains the name of the file in which the incoming data is to be placed into.

   For example, for an RPC and TACTIC specification of

   ```
   RPC foo (IN int value, OUT string buffer, OUT FILE file_name);

   TACTIC do_it = foo;
   ```

   The created *do_tactics ( )* API call will look something like this

   ```
   int foo_bar_do_tactics (foo_bar_struct *params, int value,
                           char * buffer, int * buffer_size,
                           char * file_name, int * file_name_size);
   ```

   in the application code, you must do something like this.

   ```
   buffer[5000]; // allocate space for buffer

   // allocate space for file_name and
   // make it contain the name of the output file
   file_name[256] = "/tmp/output.txt";

   int buffer_size;
   int file_name_size;

   foo_bar_do_tactics (params, value, buffer,
                       &buffer_size, file_name,
                       &file_name_size);
   ```

2. When modifying the **server**, you do not need to worry about allocating space for the output variables of individual RPCs. The stub will create global variables for the outputs of the RPCs. You just have to create functions that match the name and parameters of the RPCs specified in the tactics file.

   Using the example above, you will have to create a function called *foo* as follows:

```
int foo (int value, char * buffer, char * file_name) {

    printf("value of value is %d\n", value);

    // space has already been allocated for buffer by the stub
    strcpy (buffer, ``This is a test'');

    // space has already been allocated for file_name
    strcpy (file{\_}name, ``/tmp/file.txt'');

}
```

The stub will ensure that the values in *buffer* and *file_name* are sent back to the client.

D.8 RapidRe Application Retargeting Guide

D.8.1 Four step guide to adding applications

Step 1: Create a tactics file for the applications. This process is described in a separate document.

Step 2: Run the chroma stub generator on the tactics file to create an application specific client file containing application specific APIs and macros to interface with Chroma.

Step 3: Embed these application specific APIs and macros calls into the application at the appropriate places. This process might involve packing and unpacking variables for remote execution. This process is explained later in this document.

Step 4: Compile the application together with the chroma stub generated code and the Chroma runtime library. This process is also described later in this document.

For the rest of this document, assume that the application name is ***foo*** and that the operation provided by the application is ***bar***.

D.8.2 Step 2 : Running the stub generator

For this experiment, run "*make*" in the application build directory and the stub generator will be automatically executed. The stub generated files can be found in the source directory of the application's build directory. For example, the client stubs for *gocr* will be found in the *gocr/src_client* build directory and the server stubs will be found in the *gocr/src_server* build directory. The stub generator will create the following files:

D.8.2.1 Interface to Chroma

foo_bar.chroma.conf : configuration file read by Chroma on application startup. You do not need to use or change this file.

D.8.2.2 Client Side Code

foo_bar.chroma.h : header file containing definitions of application specific APIs, macros and data structures used by the client side of the application

foo_bar.chroma.c : file containing actual code for the APIs, macros and data structures used by the client side of the application

D.8.2.3 Server Side Code

foo_bar.chroma.server.h : header file containing definitions of application specific APIs used by the server side of the application

foo_bar.chroma.server.c : file containing actual code for the APIs used by the client side of the application

D.8.3 Step 3 : Embedding Generated APIs and Macros in an Application

D.8.3.1 Client Side APIs

The following API calls are generated for the client side of the application

```
int foo_bar_register ( );
int foo_bar_cleanup ( );
int foo_bar_find_fidelity ( );
int foo_bar_do_tactics (char * input0, int * inlen0,
                        char * output0, int * outlen0);
```

D.8.3.2 Client Side Macros

IN Vars (Parameters) For every IN variable (the parameters of the application) specified in the tactics file, the following macro will be generated.

```
// used to set the value of the parameters
set_var_name (var_type value);
```

where var_name is the name of the variable and var_type is the type of the variable (int, float, char * etc)

OUT Vars (Fidelites) For every OUT variable (the fidelities of the application) specified in the tactics file, the following macro will be generated.

```
// used to read the value of the fidelity
var_type get_var_name ( );
```

where var_name is the name of the variable and var_type is the type of the variable (int, float, char * etc)

D.8.3.3 Server Side APIs

```
void run_chroma_server ();
```

D.8.3.4 Server Side Macros

There are no macros for the server side.

D.8.4 Client Side Modifications

Modifying the client requires the following steps

1. Read the application-specific documentation and identify the routines that need to be modified.
2. Including the stub generated client header files

   ```
   #include "foo_bar.chroma.h"
   ```

3. Register the application using the *register_app ( )* API call
4. Add the *cleanup_params ( )* call at the end of the application
5. Use the *find_fidelity ( )* call to tell the runtime that work is about to be done. This call will decide the appropriate fidelities and tactics to use for the current operation. This call requires the following sub steps.
   (a) Identifying the parameters, fidelities and tactics of the application
      i. Setting the parameters before calling *find_fidelity ( )* by using the appropriate *set_var_name ( )* macro call.
      ii. Reading the fidelities after calling *find_fidelity ( )* by using the appropriate *get_var_name ( )* macro call.
      iii. Using the fidelity values to set actual application settings before doing the actual work.
6. Replace calls to the functions that perform the operation with a call to *do_tactics ( )*. This call will perform the operation using the selected tactic (either locally or remotely) and return the results to the client.
   (a) If necessary, use the results from the *do_tactics ( )* call. For example, the results may need to be displayed to the screen.

D.8.5 Detailed Description of Each Client Modification Step

D.8.5.1 Including the Stub Client Header Files

Add the Stub generated header files into the client code.

```
// This will work for C and C++ applications
#include "foo_bar.chroma.h"
```

D.8.5.2 Using the Client APIs

1) int foo_bar_register ( ) This call is used to register the application with the underlying adaptive runtime system. This call should be made as soon as possible.

The register call returns 0 if the call was successful and a non zero number if unsuccessful.

Example code stub

```
void main () {

     if (foo_bar_register( ) != 0) {
         /* register call was unsuccessful */
         printf("Unable to register application\n");
         exit(1);
     }

     /* rest of application */

}
```

2) int foo_bar_cleanup ( ) This API call is used at the end of the program to deregister the application from the underlying runtime system. This call should be made just before the application exits. It also returns 0 on success and non zero otherwise.

Example code stub

```
void main () {

     if (foo_bar_register( ) != 0) {
         /* register call was unsuccessful */
         printf("Unable to register application\n");
         exit(1);
     }

     /* rest of application */
```

```
     if (foo{\_}bar{\_}cleanup( ) != 0) {
         /* cleanup call was unsuccessful. no need to explicitly exit as
            application is already ending */
         printf("Unable to deregister application\n");
     }
}
```

3) int foo_bar_find_fidelity ( ) Before the application can ask the runtime to do work on its behalf, the application will need to use the

```
foo_bar_find_fidelity ( );
```

API call to let the runtime determine what the correct runtime parameters for the application should be.

Before making this call, the application will need to set the values of all it's parameters using the appropriate set macro functions. After this call has successfully completed, the application will be able to read the values for all it's fidelities using the appropriate get macro functions. The *find_fidelity( )* call will also determine the appropriate tactic to use for this instance and the servers to use for individual RPCs in the chosen tactic. These decisions are hidden from the application in the application specific data structure. The programmer can still access these values using the appropriate get and set macro calls. look at the *foo_bar_chroma.h* for the exact prototypes. This API call returns 0 on success and non zero otherwise.

Example code stub For this example, assume that the application has 2 parameters called length and size and 2 fidelities called resolution and framerate.

```
void main () {

     if (foo_bar_register( ) != 0) {
         /* register call was unsuccessful */
         printf("Unable to register application\n");
         exit(1);
     }

     /* rest of application */

     /* non computationally intensive GUI operation */

          .
          .
          .
```

```
    /* setup code */

    /* set the value of the 2 parameters */

    set_length (length_value);
    set_size (size_value);

    if (foo_bar_find_fidelity( ) != 0) {
        /* register call was unsuccessful */
        printf("Find Fidelity API call unsuccessful\n");
        exit(1);
    }

    /* read the values of the 2 fidelities */
    resolution = get_resolution ( );

    framerate = get_framerate ( );

    /* perform operation using values of fidelities */

    /* end of operation */

    /* rest of application */

    if (foo{\_}bar{\_}cleanup( ) != 0) {
        /* cleanup call was unsuccessful. no need to explicitly exit as
           application is already ending */
        printf("Unable to deregister application\n");
    }
}
```

4) int foo_bar_do_tactics (argument list); This API call is used by the application to tell the runtime to perform the operation on behalf of the applications. Before using this call, the *find_fidelity( )* call must have been called to allow the runtime to determine which tactic to use for the operation.

The application will have to pass in the appropriate inputs and output variables to this call. It is the responsibility of the application to allocate space for any output variables. This API call also returns 0 on success, non-zero otherwise. The actual arguments to the *do_tactics ( )* call will be found in the *foo_bar.chroma.h* stub generated header file.

Example code stub For this example, assume that the tactics prototype takes a string as an input and returns a string as output. i.e., the prototype is

```
int foo_bar_do_tactics (char * input0, int * inlen0,
```

```
                        char * output0, int * outlen0);

void main ( ) {

     /* create input and output buffers for do_tactics ( ).
        Declare length variables needed by do_tactics ( ) */

     char output_string[MAX_OUTPUT_LENGTH;
     char input_string[MAX_INPUT_LENGTH];
     int intput_length;
     int output_length;

     if (foo_bar_register( ) != 0) {
         /* register call was unsuccessful */
         printf("Unable to register application\n");
         exit(1);
     }

     /* rest of application */

     /* non computationally intensive GUI operation */

     /* setup code for operation */

     /* perform operation using tactics */

     strcpy(input_string, "Do Some Computation");
     input_length = strlen(input{\_}string);

     if (foo_bar_do_tactics (input_string, &input_string,
                             output_string, &output_string_len) != 0) {
         /* register call was unsuccessful */
         printf("Do Tactics API call unsuccessful\n");
         exit(1);
     }

     /* end of operation */

     /* rest of application */

     if (foo{\_}bar{\_}cleanup( ) != 0) {
         /* cleanup call was unsuccessful. no need to explicitly exit as
            application is already ending */
         printf("Unable to deregister application\n");
     }
```

```
}
```

D.8.6 Server Side Modifications

1. Read the application-specific documentation and identify the routines that need to be modified.

2. Add the Stub generated server header files into the server code.

   ```
   #include "foo_bar.chroma.server.h"
   ```

3. add a call to *int service_init (&argc, &argv)* to initialize the application with the underlying runtime system. The runtime will remove any parameters that were runtime specific from the argument set. This has to be done before any specific application specific processing of the cmdline arguments.

4. Preserve any application specific initialization routines.

5. Create the RPC functions specified in the tactics file (if they do not already exist) and make sure that they can accept the inputs and return the outputs specified in the tactics file. All output variables will be created as global variables by the stub generator and a pointer to that variable passed to the RPC functions. You should not declare the output variables or allocate space for them. The actual RPC functions that need to be created (along with the arguments to these functions) will be shown in the *foo_bar.chroma.server.h* stub generated header file.

6. Replace the rest of the application code with a stub generated (automatically) API call

```
void run_chroma_server ( );
```

This function will receive parameters and requests from the client, and will return outputs to the client.

Example for example, given the following application main routine and the following RPC specification.

```
RPC perform_work (IN int value, IN string position, OUT FILE outfile);
```

Original Code

```
int main (int argc, char * argv[]) {

    int value;
    char position[255];
    char output_buffer[255];

    /* initialize application */

    process_cmdline_arguments();

    initialize_app_code();

    /* start application proper */

    /* obtain user input and do client related work */

    process_client_inputs();

    do_work(value, position, output_buffer);

    /* cleanup */

    cleanup_app_code();
}
```

Modified Code to use Chroma, the main routine is modified as follows.

```
#include "foo_bar.chroma.server.h"

int perform_work (in value, char * position, char * outfile) {

    /* this is still declared as it is not part of the RPC      */
    definition                                                  */
    char output_buffer[255];

    do_work (value, position, output_buffer);

    /* new code to save contents of output_buffer into a file   */
    /* must create a file that the server can write to and save */
    /* the name in outfile                                      */

    strcpy (outfile, "/tmp/output.txt");
```

```
    /* routines to open the file and copy the contents of       */
    /* output_buffer into it                                    */

}

int main (int argc, char * argv[]) {

    /* note that the variables used by the RPC are not declared */
    /* anymore                                                  */

    /* initialize application with runtime                      */
    /* note that this *MUST be done before the application      */
    /* processes argc and argv                                  */

    if (service_init &argc, &argv) != 0) {
        perror("Unable to initialize with the runtime!");
        exit(1);
    }

    /* initialize application itself */

    process_cmdline_arguments();

    initialize_app_code ( );

    /* start application proper */

    /* make call to stub generated server API call */

    /* this will wait for input and call perform_work ( ) with
       the input values} sent by the client. It will also return
       the output values to the client. */
    run_chroma_server ( );

    /* cleanup */

    cleanup_app_code ( );

}
```

compile the code and use the application binary as the server. For ease of use, it will be advisable to rename the binary differently from the client side code. For example, the server binary might be *app_server* and the client binary might be *app_client*.

Appendix E

User Study Questionnaires

In this Appendix, I present the actual questionnaires given to participants during each user study experiment. In every questionnaire, the ordering of the answers for each questions has been randomized. In particular, some questions are phrased such that the participant has to agree with the question (in the best case) while other questions are phrased such that the participant has to disagree with the question (in the best case). This random biasing prevents a well known sampling error called "pleasing the questioner" where a participant always picks one particular answer because they sense a pattern to the questions. In each question, the words that affect the bias of the question have been highlighted to make it easier for the participant to notice the bias.

E.1 Stage A Questionnaire

Each participant had to complete 7 tasks (listed in Appendix D.6.1 and Table 4.3) to satisfy Stage A of the user study. The questionnaire for each task was the same and is shown in Appendix E.1.1. Hence, each participant was given 7 sheets of the questionnaire shown in Appendix E.1.1 (I wrote the appropriate task names on the sheets before giving it to them).

E.1.1 Standard Questionnaire

Task Name: ____

1. On a scale from 1 to 7, how easy did you find this task? (circle the appropriate number).

1	2	3	4	5	6	7

Really easy. equivalent to writing a simple hello world program	Incredibly hard. equivalent to adding concurrency support to an operating system

2. On a scale from 1 to 7, how certain are you that you performed the task correctly. I.e., you did what was asked and the lines you added to the tactics file are correct (circle the appropriate number).

1	2	3	4	5	6	7

Completely uncertain. I would be wasting money betting on my correctness even given 1000-1 odds	Incredible certain. I would bet my house that I'm correct

3. Comments on the task (if any). Use this space to highlight any problems you might have encountered in solving the task. This includes aspects of the task that you found particularly challenging or anything about the task that confused you. You can also use this space to provide suggestions for improving the task.

____

____

____

____

____

____

____

E.2 Overall Tactics File Creations Questionnaire

This is the questionnaire that was given to each participant after they had completed Stage A. It was designed to obtain more specific information about how easy they found the Stage to be.

E.2.1 1) Parameters

For the following questions, please circle the answer that best describes how you feel about the given statement.

1a) **I understood** the concept of *parameters*.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

1b) After reading the documentation, **I did not know** what the *parameters* for the type of application I was modifying would be.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

1c) **I was able** to specify the *parameters* easily in the tactics file.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

E.2.2 2) Fidelities

For the following questions, please circle the answer that best describes how you feel about the given statement.

2a) **I did not understand** the concept of *fidelities*.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

2b) After reading the documentation, I concluded that this application **did not have** any *fidelities*.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

2c) Answer this question only if the application had any *fidelities*. After reading the documentation, **I did not know** what the *fidelities* for the type of application I was modifying would be.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

2c) Answer this question only if the application had any *fidelities*. **I was able** to specify the *fidelities* easily in the tactics file.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

E.2.3 3) RPCs

For the following questions, please circle the answer that best describes how you feel about the given statement.

3a) **I understood** the concept of *RPCs*.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

3b) After reading the documentation, **I did not know** what the *RPCs* for the type of application I was modifying would be.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

3c) **I was able** to specify the *RPCs* easily in the tactics file.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

E.2.4 4) TACTICs

For the following questions, please circle the answer that best describes how you feel about the given statement.

4a) **I understood** the concept of *tactics*.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

4b) After reading the documentation, **I did not know** what the *tactics* for the type of application I was modifying would be.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

4c**)** **I was not able** to specify the *tactics* easily in the tactics file.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

E.3 Stage B Questionnaire

Each participant had to complete 7 tasks (listed in Appendix D.7.1 and Table 4.3) to satisfy Stage B of the user study. The questionnaire for each task was the same and is shown in Appendix E.1.1. Hence, each participant was given 7 sheets of the questionnaire shown in Appendix E.1.1 (I wrote the appropriate task names on the sheets before giving it to them).

E.4 Stage C Questionnaire

Each participant had to complete 7 tasks (listed in Appendix D.7.1 and Table 4.3) to satisfy Stage C of the user study. The questionnaire for each task was the same and is shown in Appendix E.1.1. Hence, each participant was given 7 sheets of the questionnaire shown in Appendix E.1.1 (I wrote the appropriate task names on the sheets before giving it to them).

E.5 Overall Retargeting the Application Questionnaire

This is the questionnaire that was given to each participant after they had completed Stages B and C. It was designed to obtain more specific information about how easy the entire process of retargeting the application into client and server components. The verbal "this is what the stub generation does" description given to the participants (for question 5a) is shown in Appendix E.5.2.

E.5.1 Overall Questionnaire About Modifying the Application

1) Knowledge of Chroma

1a) How much did you end up knowing about the internal workings of Chroma?

Please tick the answer that you agree most with (tick only 1 answer).

_____ Nothing at all (Chroma is a complete black box)
_____ Learnt a little bit (For example, saw the main routine and understood it)
_____ Learnt the basic functionality (For example, understood the high level control path)
_____ Learnt a large amount of functionality (For example, understood the details of some of the internal libraries)
_____ Learnt every detail of Chroma (understood the exact workings of all the libraries)

Please circle the answer that best describes how you feel about the given statement.

1b) The level of knowledge I had of Chroma **was sufficient** to successfully complete all the tasks.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

2) Knowledge of the Application

For the following questions, please circle the answer that best describes how you feel about the given statement.

2a) I **did not acquire** an understanding of the application control flow related to the operation.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

2b) Understanding the high application control flow related to the operation **was necessary for** successfully completing the task

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

2c) I **did not acquire** a detailed understanding of the entire application control flow.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

2d) Understanding the entire application control flow **was necessary for** successfully completing the task.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

2e) I **ended up understanding** the inputs and output to the functions that perform the required operation.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

2f) Understanding the inputs and output to the functions that perform the required operation **was not necessary** to successfully complete the task.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

2g) I **ended up understanding** the inner workings of the functions and libraries that performed the required operation.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

2h) Understanding the inner workings of the functions and libraries that performed the required operation **was not necessary** to successfully complete the task.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

2i) What else, if anything, about the application did you need to know (that was not covered above)?

3) Usefulness of Tool

Please circle the answer that you agree most with for the following questions.

3a) How much do you think the stub generated code and the modification process assisted your task in separating the application into client and server components that could work with an adaptive runtime system (Chroma in this case)?

Didn't help at all   A little bit   Somewhat   Quite a lot   Helped Immensely

3b) How long do you think you would have needed to modify the application if you didn't have the stub generated code and modification guide?

5 – 6 hrs   1 day   3 – 5 days   1 – 2 weeks   1 month   More than 1 month

4) Comparative Effort

For the following questions, please circle the answer that best describes how you feel about the given statement

4a) Writing a simple web proxy **is easier than** making this application adaptive using the methods presented in this user study.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

4b) Adding concurrency support to an operating system **is harder than** making this application adaptive using the methods presented in this user study.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

4c) Writing an implementation of malloc **is harder than** making this application adaptive using the methods presented in this user study.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

4d) Creating a stateful firewall application **is easier than** making this application adaptive using the methods presented in this user study.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

4e) Writing a simple html webpage that only says "Hi There" **is harder than** making this application adaptive using the methods presented in this user study.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

4f) Writing a ftp/irc client **is easier than** making this application adaptive using the methods presented in this user study.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

4g) Writing a kernel device driver **is easier than** making this application adaptive using the methods presented in this user study.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

4h) Writing the application you modified **is harder than** making the application adaptive using the methods presented in this user study.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

4i) Writing a simple "Hello World" program **is easier than** making this application adaptive using the methods presented in this user study.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

5) Final Question

5a) Rajesh will verbally tell you what the stub code is actually doing to interface the application with Chroma. The stub code is made possible by the modification process (providing specific information in the tactics file). Given this information, how long do you think you would have needed to modify the application if you didn't have the stub generated code and modification guide?

Please circle the answer that you agree most with:

5 – 6 hrs 1 day 3 – 5 days 1 – 2 weeks 1 month More than 1 month

E.5.2 Description of What the Stub Generator Does

This is the description that was given to the participants for question 5a) in the "Overall Application Modification Questionnaire" shown in Appendix E.5.

E.5.2.1 Stuff the Stub Generator Does for You

- connects to the runtime
- calls the right runtime initialization routines with the right parameters
- calls the correct runtime API calls
  - to set parameters
  - retrieve fidelities
  - ask the runtime to decide where to do stuff
- shields you from having to pack / unpack variables to send to the runtime
- do_tactics performs the appropriate remote execution calls for you
  - it will create the right demultiplexor to do the right sequence of RPC calls for the appropriate tactic
  - it will also do the pthreads stuff required to do parallel execution
  - calls the appropriate runtime calls to do the RPC call at the chosen server
  - it will marshall / unmarshall the data to be sent to / from the remote servers

E.6 Overall Experiment Questionnaire

This questionnaire was given to each participant after they had completed all three stages. It was designed to obtain feedback about the user study process itself.

E.6.1 General Questionnaire About The Entire User Study

1. Was the training helpful?

   Please circle the answer that you agree most with:

   Didn't help at all A little bit Somewhat Quite a lot Helped Immensely

2. How can we improve the training?

   ____
   ____
   ____
   ____
   ____
   ____
   ____

3. Was the documentation helpful?

   Please circle the answer that you agree most with:

   Didn't help at all A little bit Somewhat Quite a lot Helped Immensely

4. How can we improve the documentation?

   ____
   ____
   ____
   ____
   ____
   ____

____

5. What could we have done to improve the User Study experience?

____
____
____
____
____
____
____

6. Any other comments / suggestions / observations?

____
____
____
____
____
____
____

E.7 Effect of Experience Questionnaires

E.7.1 Effect of Experience on Creating the Tactics File

For the following questions, please circle the answer that best describes how you feel about the given statement.

1. Creating a tactics file for an application **was much easier** the second time round.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree

E.7.2 Effect of Experience on Modifying the Application

For the following question, please circle the answer that best describes how you feel about the given statement.

1. Modifying an application (to make it adaptive) **was much easier** the second time round.

Strongly Agree	Somewhat Agree	Neutral	Somewhat Disagree	Strongly Disagree