Simplifying Cyber Foraging for Mobile Devices

Rajesh Krishna Balan **Darren Gergle**
Mahadev Satyanarayanan **Jim Herbsleb**

August 2005
CMU-CS-05-157

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Cyber foraging is the transient and opportunistic use of compute servers by mobile devices. The short market life of such devices makes rapid modification of applications for remote execution an important problem. We describe a solution that combines a "little language" for cyber foraging with an adaptive runtime system. We report results from a user study showing that even novice developers are able to successfully modify large, unfamiliar applications in just a few hours. We also show that the quality of novice-modified and expert-modified applications are comparable in most cases.

This research was partially supported by the National Science Foundation (NSF) under grant numbers ANI-0081396 and CCR-0205266, and by an equipment grant from the Hewlett-Packard Corporation (HP). Rajesh Balan was supported by an IBM Graduate Fellowship in 2003-2005 and by a USENIX Graduate Fellowship in 2002. Darren Gergle was supported by an IBM Graduate Fellowship in 2003-2006. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, HP, IBM, USENIX or Carnegie Mellon University.

Keywords: Mobile Systems, User Study, Software Engineering

1 Introduction

By a curious paradox, applications of highest value to a mobile user are the hardest to support on lightweight and compact hardware with long battery life. Natural language translation and speech recognition, for example, would be helpful to a traveller in a foreign country. Optical character recognition of signs in a foreign script could help a lost traveller find his way. A wearable computer with an eyeglass display and a camera for face recognition could serve as an augmented-reality system for assisting an Alzheimer's patient. Alas, the CPU, memory and energy demands of these applications far outstrip the capacity of devices that people are willing to carry or wear for extended periods of time. On such hardware, improving size, weight and battery life are higher priorities than enhancing compute power.

One way to resolve this paradox is for a mobile device to perform remote execution on a nearby compute server over a wireless link. Cheap commodity machines widely dispersed for public use could act as compute servers for mobile devices in their vicinity. We refer to this transient and opportunistic use of resources as *cyber foraging*. Although deployment of compute servers for public use is not imminent, our work addresses future environments where they may be 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.

Implementing cyber foraging involves three steps. First, a mobile device must find a compute server. Second, it must establish trust in that server. Third, it must partition the application between local and remote execution. This decision may have to change with fluctuations in operating conditions such as wireless bandwidth and battery level.

We focus on the third problem in this paper, deferring to others for solutions to the first two. Service discovery [34] is an active area of research in pervasive computing, with solutions such as Jini [49], UPnP [22], and Bluetooth proximity detection [19, 39]. Establishing trust in hardware is a major goal of the security community, especially the Trusted Computer Group [47]. The recent work on trusted platform modules at IBM [37, 38], and Chen and Morris' work on tamper-evident remote execution [7] are of particular relevance here.

Our goal is to enable *rapid modification of applications for cyber foraging*. This is important because of 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 [10, 17, 24, 50]. With a typical market life of barely a year, fast delivery of new hardware with a full suite of applications is critical.

We propose a solution based on the well-known approach of *little languages* [2]. By developing abstractions that are well-matched to the problem of cyber foraging, our solution makes possible a compact static description of all the meaningful partitions of an application. Complementing this static description is a powerful runtime system that provides the dynamic components necessary for adaptation to fluctuating operating conditions. A stub-generation tool creates application-specific interfaces to the runtime system.

We report results from a user study showing that novice developers can modify large, unfamiliar applications in just a few hours. These applications span speech, natural language, and computer vision technologies and are relevant to domains such as travel, health care, and engineering. We also report results showing that the quality of novice-modified and expert-modified applications are comparable in most cases.

2 Design Considerations

2.1 Language-Independent & Coarse-Grained

An obvious design strategy for cyber foraging would require all applications to be written in a language that supports transparent 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 modifications would be needed. This language-based, fine-grained approach to remote execution has been well explored, dating back to the Emerald system [23] of the mid-1980s.

We rejected this strategy because of its restriction that all applications be written in a single language. An informal survey of existing applications from the domains mentioned in Section 1 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, our validation suite in Section 4 includes applications written in C, C++, Java, Tcl/Tk and Ada.

Our decision to be language-independent had a number of consequences. First, it eliminated the use of fully automated code-analysis techniques since these tend to be language-specific. Second, it implied that applications had to be manually modified to use runtime support for cyber foraging. Third, 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, we lower complexity but give up on discovering the theoretically optimal partitioning. This is consistent with our emphasis on reducing programmer burden and software development time, as long as we are able to produce an acceptable cyber foraging solution.

2.2 Support for Runtime Adaptation

The fickle nature of resource availability in mobile computing environments has been well documented by researchers such as Forman et al. [13], Katz [25], and Satyanarayanan [40]. Dynamic change of *fidelity* (application-specific output quality) has been shown to be effective in coping with fluctuating resource levels by Fox et al [14], Noble et al. [32], de Lara et al [8] and Flinn et al. [12].

These findings, including the causes of resource variation that underlie them, also apply to a mobile device that uses cyber foraging. The device may be subject to additional resource variation if its compute server is shared. Many factors affect this variation, including the demands of other mobile devices, the admission control policy used in service discovery, and the compute server's resource allocation policy.

Clearly, a good design for cyber foraging must support the concept of fidelity. It must also include the runtime support necessary for monitoring resource levels and selecting an appropriate fidelity. The selection mechanism must take user preference into account when there are multiple dimensions of output quality. For example, a user in a casual conversation may prefer quick natural language translation even if it involves some loss of accuracy; for a business negotiation, however, accuracy may be much more important than speed.

2.3 Port Early, Port Often

Short device life and its implications for software development were dominant considerations in our design. Our 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 ported to older devices, but others may not. To attract new corporate customers, the vendor must also help them rapidly port their critical applications. The lower the quality of programming talent needed for these efforts, the more economically viable the proposition.

This leads to the central challenge of our work: *How can novice software developers rapidly modify large, unfamiliar applications for cyber foraging?* We assume that application source code is available; otherwise, the problem is intractable. Just finding one's way around a large body of code is time consuming. Our 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. Obviously, the quality of the resulting port must be good enough for serious use. In rare cases, a new application may be written from scratch for the new device. Our design does not preclude this possibility, but we do not discuss this case further in this paper.

3 Our Solution

3.1 Properties of a Good Solution

Given these considerations, how can we tell if we are successful? What defines a good solution? Such a solution would enable novice 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 novice can use our 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 our own experience and that of others we 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*. Since programming is an error-prone activity, it is unrealistic to expect a novice to produce error-free code with our solution. A more realistic goal is a solution that avoids inducing systematic or solution-specific coding errors by novices. 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 novices using our solution should be indistinguishable from those produced by an expert.

3.2 Solution Overview

Our solution is in three parts. First, we provide a "little language" called *Vivendi* for expressing application-specific information that is relevant to cyber foraging. A developer examines the source code of an application and creates a Vivendi file called the "tactics file." The tactics file contains the function prototype of each procedure deemed worthy of remote execution, and specifies how these procedures 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 our solution is a runtime system called *Chroma* that provides support for resource monitoring, adaptation, and learning-based prediction. Chroma supports history-based predictive resource management in a manner similar to that described by Narayanan et al. for the Odyssey system [29]. A call to Chroma allows the application to discover the tactic and fidelity it should use for the next compute-intensive operation. Chroma bases its estimate on current resource levels and predicted resource consumption of the next operation. Chroma has to be ported once to each new mobile device, and is then available to all applications.

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 [3]. 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 our solution. This is a set of application-independent 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.

To modify an application for cyber foraging, a developer proceeds as follows. She first examines the application source code and creates the tactics file. Next, she runs the Vivendi stub generator to create stubs. Then she modifies the application by inserting calls to the stubs at appropriate points in the source code. Finally, she compiles and links the modified application, stubs and Chroma. On occasion, there may be an additional step of modifying the user interface of an application for a new mobile device. Our work does not address this step, but defers to ongoing

```
APPLICATION graphix;
REMOTEOP render;

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

// 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;

// 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);
```

Figure 1: Example Tactics File in Vivendi

work on automated user interface generation [9, 30]. We describe our solution components in more detail in Sections 3.3 to 3.5.

3.3 Vivendi

Vivendi enables concise description of the tactics and fidelities of an application. Figure 1 shows the tactics file for a hypothetical application called `graphix`. Each application code component that may benefit from remote execution is called a *remoteop* (short for "remote operation") and is identified in Vivendi by the tag `REMOTEOP`. A remoteop's size and complexity determine the granularity at which cyber foraging occurs. We expect only a few remoteops for each application, possibly just one. For example, Figure 1 shows a single remoteop called `render` for the application `graphix`.

Next, the tactics file specifies the critical variables that influence the amount of resources consumed by executing this remoteop. In language translation, for example, the number of words in the sentence to be translated is the (single) critical variable. A scene illumination application may have two such variables: the name of the 3D image model and its current viewing position. We refer to such variables as *parameters* of the remoteop. Figure 1 shows a single parameter, called `size` for the remoteop `render`. Vivendi passes parameter information to Chroma, which uses this knowledge in its history-based resource prediction mechanism. Chroma's prediction specifies the fidelity at which the remoteop should be executed. Figure 1 indicates that `quality` is the variable corresponding to fidelity for the remoteop `render`. Parameters and fidelities are specified like C variables, with the keyword `IN` indicating parameters and `OUT` indicating fidelities. Vivendi supports a full suite of C-like primitive data types.

The tag `TACTIC` identifies a tactic for this remoteop. Each tactic represents a different way of combining RPCs to produce a remoteop result. Chroma selects the appropriate tactic and the binding of RPCs to compute servers. These choices are frozen for the duration of a remoteop, but

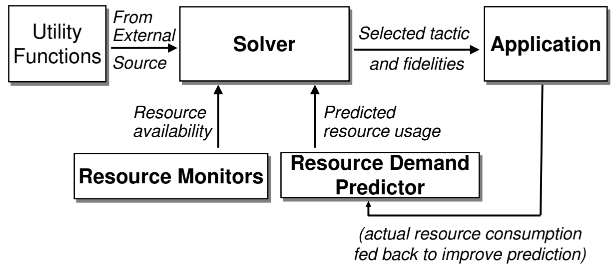


Figure 2: Main Components of Chroma

are re-evaluated for the next remoteop. Vivendi syntax allows any combination of sequential and parallel RPCs to be specified as a tactic. It also provides control over placement of specific RPCs on servers. This might be useful, for example, where a later RPC has to be run on the same server as an earlier RPC to take advantage of a warm cache or to avoid shipping a large intermediate result. For brevity, we omit these syntax details. A taste of the syntax can be obtained from Figure 1, which specifies two tactics: `do_simple` and `do_all`. Sequential RPCs are separated by an `&` operator while parallel RPCs are separated by commas and appear within parentheses.

Finally, the RPCs used in tactics are specified using a syntax similar to that for standard function prototype definitions. The tag `RPC` identifies the RPCs of remoteop `render` in Figure 1. Although we omit the details here, a wide range of basic data types can be used as RPC arguments. This includes uninterpreted binary data objects and file variables.

3.4 Chroma

Chroma provides resource measurement, prediction and fidelity selection functions that complement Vivendi. Through integration with Linux, Chroma is able to perform these functions even when concurrent applications use cyber foraging. Figure 2 shows the main components of Chroma.

At the heart of Chroma is a *solver* that responds to queries from Vivendi stubs regarding the tactics and fidelity to use for a remoteop. The solver constructs a solution space of tactic-fidelity combinations and then exhaustively searches this space for the current optimum. The space is relatively small since there are few tactics. The goodness of a specific point in this space is computed by a *utility function* that quantifies informal directives such as “conserve battery”, “maximize quality” or “give best quality under 1 second”. Our prototype uses closed-form utility functions provided by an entity outside Chroma. A more complete system would derive the utility function from current user preferences.

The inputs to the solver include resource supply measurements and resource demand predictions. The supply measurements are provided by the *resource monitors* shown in Figure 2. These are software sensors in the mobile client and compute server that report values of network bandwidth, CPU utilization, memory usage, battery level and so on. As shown in Figure 2, resource demand predictions are made by a history-based predictor. This predictor continuously improves its accuracy by comparing previous predictions with actual resource usage, and refining its prediction model. The predictor can be initialized using off-line training or history from an older mobile device. Parameter values from the Vivendi stub are factored into the prediction model.

```
/* 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);

/* Parameters and fidelity variables */
void set_size (int value);
float get_quality ( );
```

Figure 3: Vivendi Wrapper Stubs for Chroma Interactions

3.5 Generated Stubs

The Vivendi stub generator creates two kinds of stubs from a tactics file: standard RPC stubs and wrapper stubs. The standard RPC stubs perform packing and unpacking of arguments. They also provide a server listener loop with opcode demultiplexing. We omit further discussion of these since they follow well-known RPC practice. The wrapper stubs simplify application modification by customizing the Chroma interface to the application.

Figure 3 shows the wrapper stubs for the tactics file shown in Figure 1. A developer inserts `graphix_render_register` at the start of the application and `graphix_render_cleanup` just before its exit. She inserts `graphix_render_find_fidelity` just before the code that performs the `render` remoteop. Right before this, she inserts `set_size` to set the parameter value for this remoteop. Right after this, she inserts `get_quality` to obtain the fidelity recommended by Chroma. Finally, she removes the actual body of code for the remoteop and replaces it by a call to `graphix_render_do_tactics`. This will create the client that performs the operation remotely, using a tactic selected by Chroma.

To create the server, starting with an unmodified application, she inserts two API calls, `service_init` and `run_server` into the application's main routine to initialize the server and to start the server's listening loop respectively. Finally, she creates the required RPC server functions using the remoteop code removed from the client.

4 Validation Approach

The primary goal of our validation study was to assess how well our solution meets the goodness criteria laid out in Section 3.1. A secondary goal was to gather detailed process data to help identify areas for future research. Our approach combines well-established user-centric and system-centric evaluation metrics. User-centric metrics for programmers focus on measures such as ease-of-use, ease-of-learning, and errors committed [43]. System-centric metrics focus on measures such as application latency or lines of generated code.

We combined these techniques in a laboratory-based user study with two parts. In the first part, novice developers modified a variety of real applications for cyber foraging. We describe this part in Section 4.1 and report its results in Sections 5 to 7. In the second part, we compared the performance of these modified applications to their performance when modified by an expert. We describe this part in Section 4.2 and report its results in Section 8

4.1 User-Centric Evaluation

Following the lead of Ko et al. [27] and Klemmer et al [26], we took user-centric evaluation methods originally developed for user interface investigations and adapted them to the evaluation of programming tools.

4.1.1 Control Group

In designing the user study, a major decision was whether to incorporate a control group in our 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.

Our 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, our 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. We therefore chose to forego a control group.

4.1.2 Test Applications

We chose eight applications of the genre mentioned at the beginning of this paper. Table 1 shows their salient characteristics. The applications were: *GLVU* [46], a virtual walkthrough application that allows users to navigate a 3D model of a building; *Panlite* [15], an English to Spanish translator; *Radiator* [51], a 3D lighting modeler; *Face* [41], a face recognition application; *Janus* [48], a speech recognizer; *Flite* [4], a text to speech converter; *Music* [20], an application that records audio samples and finds similar music on a server; and *GOCR* [42], an optical character recognizer.

None of these applications was written by us, nor were any of them designed with remote execution, adaptation, or mobile computing in mind. As Table 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.

4.1.3 Participants and Setup

We selected participants whose characteristics match those of novice developers, as discussed in Section 2.3. In many companies, the task of porting code falls to junior developers. We modeled

Application	Lines of Code	Number of Files	Language	Fidelities	Parameters	RPCs	Total RPC Args	Tactics
Face (Face Recognizer)	20K	105	Ada w/C interface	0	1	1	2	1
Flite (Text to Speech)	570K	182	C	0	1	1	2	1
GLVU (3D Visualizer)	25K	155	C++, OpenGL	1	15	1	18	1
GOCR (Character Recognizer)	30K	71	C++	0	1	1	2	1
Janus (Speech Recognizer)	126K	227	C, Tcl/Tk, Motif	1	1	3	9	2
Music (Music Finder)	9K	55	C++, Java	0	2	1	2	1
Panlite (Lang Translator)	150K	349	C++	0	1	4	11	7
Radiator (3D Lighting)	65K	213	C++, OpenGL	2	1	1	4	1

Table 1: Overview of the Test Applications

this group by using undergraduate seniors majoring in computer science. In addition, we used a group size large enough to ensure the statistical validity of our findings. While the exact numbers depend upon the variability within the participants and the overall size of the effects, widely accepted practices recommend between 12 and 16 users [31]. We used 13 participants, which falls within this range and represents the limit of our resources in terms of time (six hours per data point).

On average, our participants were about 21 years old. Our selection criteria required them to know C programming and be available for a contiguous block of six hours. None of them were familiar with the research of our lab, any of the tools under development, or any of the test applications. Table 8 shows the assignment of participants to applications. As the table shows, several participants returned for additional applications. In keeping with standard HCI practice, we counter-balanced the assignment of participants to applications to avoid any ordering effects. These additional data allowed us to investigate learning effects and to determine whether our one-time training was adequate.

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

The participants worked alone in a lab for the duration of the study. We 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 [45]. This provided us with detailed logs of user actions as well as accurate timing information.

4.1.4 Experimental Procedure

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 remoteops, parameters, fidelities, RPCs and tactics. We then conducted a hands-on training session using the GOCR application where we demonstrated how to identify and describe these concepts in Vivendi. The participants were provided with documentation on Vivendi syntax, with many examples. We then guided the participants in modifying GOCR. Training sessions 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: We provided participants with a structured task and a set of general instructions. The task structure consists of three stages, as shown in Table 2. In Stage A, the primary activity is creating the tactics file; in Stage B, it centers on creating the client code component; in Stage C, it centers on creating the server component. We wanted to cleanly isolate and independently study the ability of novices to perform each of these stages. We 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.

As Table 2 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, we asked participants to answer a set of questions about it.

4.1.5 Data Collected

Timing: Using Camtasia recordings, we obtained completion times for each subtask. These could be aggregated to find completion times for stages or for the overall task.

Task Process: From Camtasia recordings, we collected data on how participants completed all of the subtasks, noting where they had trouble, were confused, or made mistakes.

Self-Report: We collected questionnaire data of several types, including quality of training, ease of use of our solution, and performance in each subtask.

Stage A	Stage B	Stage C
Tactics file	*Client component*	*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 2: Task Stages

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

4.2 System-Centric Evaluation

The goal of the system-centric evaluation was to understand whether rapid modification by a novice resulted in adequate application quality. For each application, we asked an expert who had a good understanding of our solution and the application to create a well-tuned adaptive version of the application. 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.

4.2.1 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, we performed the comparisons for six scenarios that might typically occur in cyber foraging.

These six scenarios are shown in Table 3. We used two values of load on compute servers: light (1% utilization) and heavy (95% utilization). We used two bandwidth values: high (5 Mb/s) and low (100 Kb/s), based on published measurements from 802.11b wireless networks [28]. This yielded four scenarios (labeled "LH," "HH, "LL" and "HL" in Table 3). 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.

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 3: Scenario Summary

4.2.2 Experiment Setup

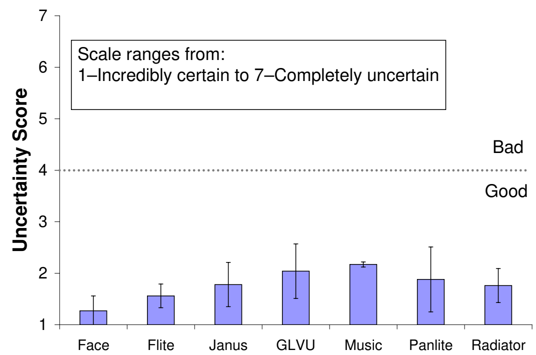
To model a resource-poor mobile device, we used an old Thinkpad 560X laptop with a Pentium 266 MHz processor and 64 MB of RAM. We modeled high and low end compute servers using two different kinds of machines: *Slow*, with 1 GHz Pentium 3 processors and 256 MB of RAM, and *Fast*, with 3 GHz Pentium 4 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 confounding effects due to Chroma's history-based mechanisms, we initialized Chroma with the same history before every experiment.

4.2.3 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. Our system-centric results are presented in Section 8.

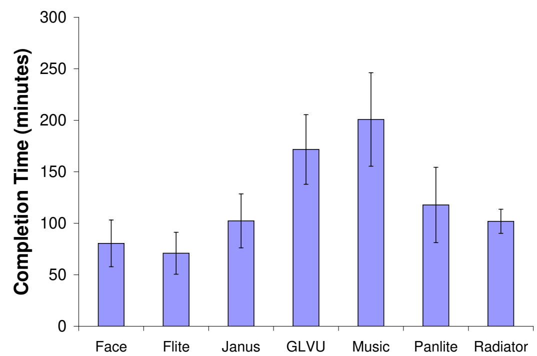
5 Results: Little Training

The first criterion for a good solution relates to training duration, as listed in Section 3.1: "Can novices face complex applications confidently with little training?" Our training process was presented in Section 4.1.4. As stated there, the training session was one hour or less for all participants,



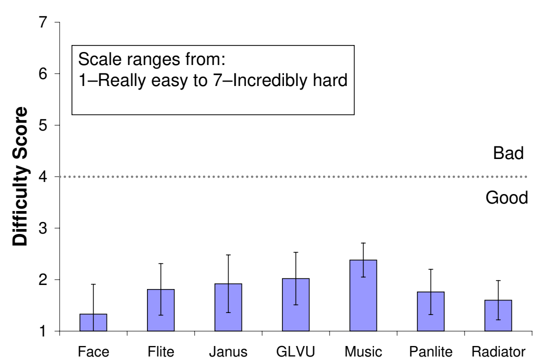
For each application, the height of its bar is the mean uncertainty score on the Likert scale shown in the legend, averaged across all participants. Error bars show the standard deviation.

Figure 4: Self-Reported Uncertainty Scores



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

Figure 5: Measured Application Completion Times



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. Error bars show the standard deviation.

Figure 6: Self-Reported Task Difficulty Scores

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 our participants have in actually modifying applications. These results are reported in the rest of the paper. A secondary test is the subjective impression of participants. We asked participants several questions after task completion to help us judge whether they felt adequately prepared.

Our questions probed directly about the value of the training and training materials. Participants responded 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). In response to the question, "Was the training helpful?" 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.

In addition, after every subtask of Table 2, we 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, 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 highly with the task performance times presented in Section 6. 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. We will discuss these results in more detail in Section 10, where we identify opportunities for improving our solution. Overall, these results suggest that the participants believed their training prepared them well for the modification tasks they faced.

6 Results: Quick Modifications

In this section, we address the second criterion listed in Section 3.1: "Can novices modify complex applications quickly?" To answer this question, we examined overall task completion times across the range of applications in our validation suite. We 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 5 shows the distribution of task completion times, and Table 4 presents the breakdown of these times across task stages. 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.

The proportion of the original code base that was modified is another measure of task simplicity. Table 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 stub-generated code as hand-written code. In addition to the reduction in coding effort, the use of stubs allowed participants to get away with minimal knowledge of Chroma.

Finally, we 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 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 our 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 our validation suite.

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: Completion Time by Task Stage

App	Lines of Code	File Count	Tactic File Size	Stage B: Client Modifications				Stage C: Server Modifications			
				Lines Added	Lines Removed	Stub Lines	Files Changed	Lines Added	Lines Removed	Stub Lines	Files Changed
Face	20K	105	10	31 – 68	12 – 15	556	2	26 – 45	15 – 24	186	2
Flite	570K	182	10	29 – 39	1 – 5	556	2	13 – 30	3 – 87	186	2
GLVU	25K	155	38	62 – 114	3 – 21	1146	2	88 – 148	12 – 32	324	2
Janus	126K	227	25	28 – 47	2 – 7	1538	3	59 – 130	7 – 70	434	4
Music	9K	55	11	61 – 77	4 – 6	1127	2	131 – 269	23 – 147	203	2
Panlite	150K	349	21	30 – 66	1 – 39	1481	3	12 – 73	18 – 39	406	3
Radiator	65K	213	15	41 – 51	1 – 47	643	2	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 Removed columns show how many lines were added and removed when performing the task. Stub Lines gives the number of stub-generated lines of code. Files Changed gives the maximum number of files that were actually modified by the participants.

Table 5: Application Modifications

7 Results: Low Error Rate

In this section, we examine the third criterion listed in Section 3.1: "Can novices modify complex applications with few errors?" Since programming is an error-prone activity, we expect novice-modified applications to contain ordinary programming errors of the types described by Pane et al. [33]. In addition, we expect a few additional simple errors because participants could not test their solution, except to verify that it compiled cleanly. We 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 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, we 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 our 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 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 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. We examine the performance impact of these errors in Section 8.

In summary, we 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. We are 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 10 we examine ways to reduce this error rate even further.

8 Results: Good Quality

The fourth criterion listed in Section 3.1 pertains to the quality of modified applications: "Can novices produce modified applications whose quality is comparable to those produced by an expert?" To answer this question, we conducted the system-centric evaluation described in Section 4.2.

For each novice-modified application, we conducted 36 experiments comparing its performance to that of the same application modified by an expert. As explained in Section 4.2.3, these

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 6: Total Errors for Stage A Across All Participants

36 experiments explored combinations of compute server loads, network bandwidths, user preferences, and server speeds. For each experiment, we 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. We report latency in seconds of elapsed time.

We 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. We evaluated the performance of client and server components of each application separately. No anomalies were observed for server components: all 25 were indistinguishable from their expert-modified counterparts.

Table 8 presents our 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 8 shows that novice- and expert-modified applications were indistinguishable in 16 out of 25 cases.

Table 9 shows details of the anomalies. To save space, it only shows the performance of one anomalous version of GLVU; the other 3 anomalous versions were similar. For each application, we provide the relative fidelity and latency obtained for all 3 inputs in all 6 scenarios. The relative fidelity is 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.

We observe that 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

	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 7: Total Errors for Stages B and C Across All Participants

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 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.

In summary, our results confirm that 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 give us ideas for improving our solution. We discuss these improvements in Section 10.

Participant Number

	1	2	3	4	5	6	7	8	9	10	11	12	13
Face	100%			100%							67%		
Flite								100%			100%	67%	
GLVU	44%		44%		44%		100%		44%				
Janus		100%	100%				100%						
Music			100%		100%								100%
Panlite		100%		83%		100%		100%		100%			
Radiator				94%						100%		78%	

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 8: Relative Performance of Novice-Modified Client Component

9 Why Our 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?

The key to explaining our success is to recognize 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, we were 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

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	…	…	…

(a) Face (Participant 11)

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	…

(b) Flite (Participant 12)

	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

(c) GLVU (Participant 1)

	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	…	…	…	…	…	…
	…	…	…	…	…	…
	…	…	…	…	…	…

(d) Panlite (Participant 4)

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

(e) Radiator (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

(f) Radiator (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 9: Detailed Results for Anomalous Application Versions

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 our applications. Most applications of this genre already have "tuning knob" parameters that map easily to fidelities — another pleasant surprise.

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

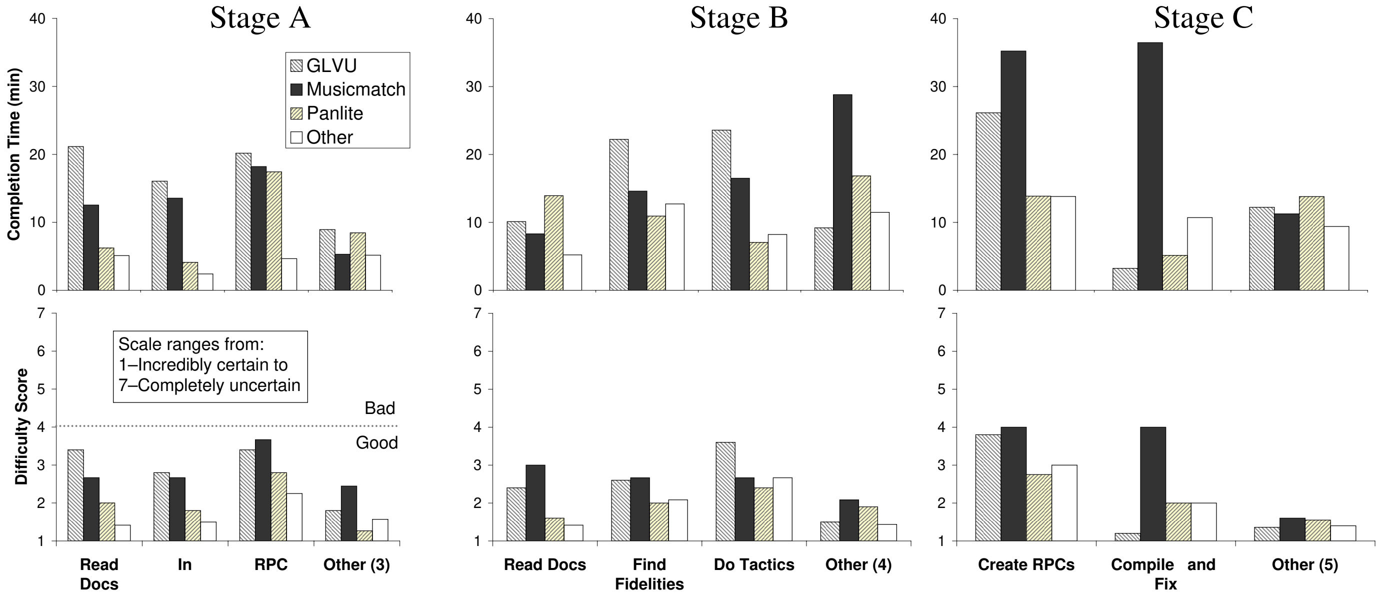
In addition to reducing programmer burden, there is also a significant software engineering benefit to the clean separation of concerns implicit in our design. The application and the runtime system can be independently evolved, with many interface changes only requiring new stubs.

10 Improving the Solution

Our solution 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, we analyze all the non-trivial errors, examine how the subjects spent their time, and examine the differences in applying the solution to the range of applications. Since our solution is already fast, we focus on improving the solution quality.

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.



Only the largest time values (top row) and self-reported difficulty scores (bottom row) 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 7: Time and Difficulty of Each Individual Subtask

In order to eliminate these errors, we need to determine whether the programmers were unable to understand the task or simply forgot to complete all necessary steps. If the latter, straightforward improvements in the instructions may be sufficient to eliminate all observed errors. An examination of the evidence summarized in Figure 7 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 8 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 7 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 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, we 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 our solution.

11 Related Work

Our work spans mobile computing, software engineering and HCI. At their juncture lies the problem of rapid modification of resource-intensive applications for cyber foraging. To the best of our knowledge, we are the first to recognize the importance of this problem and propose a solution. Our solution and its validation build upon work in three areas: little languages, adaptive systems, and HCI evaluation methods.

The power of little languages was first shown by early versions of the Unix programming environment. *Make* [11] is perhaps the best-known example of a little language. As Bentley explains [2], the power of a little language comes from the fact that its abstractional power is closely matched to a task domain. Our use of tactics and the design of Vivendi apply this concept to cyber foraging.

Chroma's approach to adaptation builds on ideas first proposed in Odyssey [32]. Its use of history-based prediction follows the lead of Narayanan et al. [29] and Gurun et al [18]. The use of remote execution to overcome resource limitations has been explored by many researchers, including Rudenko [36] and Flinn [12].

We used well-established techniques from HCI to conduct our user-centric evaluation. Nielsen [31] gives a good overview of these techniques. Ko et al. [27] and Klemmer et al. [26] show how these techniques could be applied to the evaluation of programming tools.

From a broader perspective, our work overlaps with automatic re-targeting systems such as IBM's WebSphere [6] and Microsoft's Visual Studio [52]. These systems allow developers to quickly port applications to new target systems. Unfortunately, they use a language-specific approach, which runs counter to our design considerations.

Finally, dynamic partitioning of applications has a long and rich history in distributed systems and parallel computing. In the space available, we cannot fully attribute this large body of prior work. A sampling of relevant work includes Mentat [16], Jade [35], Nesl [5], Abacus [1] and Coign [21]. None of these efforts focus specifically on mobile computing.

12 Conclusion

Mobile computing is at a crossroads today. A decade of sustained effort by many researchers 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 [44, 53], but their impact has been negligible outside niche markets.

An entirely different world, sketched in the first paragraph of this paper, 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.

Our work aims to break this deadlock. We lower the cost of creating resource-intensive mobile applications by reusing existing software that was created for desktop environments. Using our approach, relatively unskilled programmers can do a credible job of rapidly porting such software to new mobile devices. Even if the result is not optimal in terms of performance, it is typically good enough for real use. We have validated our approach on seven applications. The next step is, of course, to enlarge the suite of applications. This will help us broaden the validity of our approach, and improve it along the lines discussed in Section 10. If these efforts meet with continued success, we are confident that our work can help stimulate the transformation of mobile computing.

References

[1] Amiri, K., Petrou, D., Ganger, G., Gibson, G. Dynamic Function Placement for Data-Intensive Cluster Computing. *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, CA, June 2000.

[2] Bentley, J. Little Languages. *Communications of the ACM*, 29(8):711–721, 1986.

[3] Birrell, A.D., Nelson, B.J. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

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

[5] Blelloch, G.E., Chatterjee, S., Hardwick, J.C., Sipelstein, J., Zagha, M. Implementation of a Portable Nested Data-Parallel Language. *Proceedings of the fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, San Diego, CA, May 1993.

[6] Budinsky, F., DeCandio, G., Earle, R., Francis, T., Jones, J., Li, J., Nally, M., Nelin, C., Popescu, V., Rich, S., Ryman, A., Wilson, T. WebSphere Studio Overview. *IBM Systems Journal*, 43(2):384–419, May 2004.

[7] Chen, B., Morris, R. Certifying Program Execution with Secure Processors. *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HOTOS)*, Lihue, HI, May 2003.

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

[9] Eisenstein, J., Vanderdonckt, J., Puerta, A. Applying Model-Based Techniques to the Development of UIs for Mobile Computers. *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*, Santa Fe, NM, January 2001.

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

[11] Feldman, S.I. Make-A Program for Maintaining Computer Programs. *Software - Practice and Experience*, 9(4):255–265, 1979.

[12] Flinn, J., Satyanarayanan, M. Energy-Aware Adaptation for Mobile Applications. *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[13] Forman, G., Zahorjan, J. The Challenges of Mobile Computing. *IEEE Computer*, 27(4):38–47, April 1994.

[14] Fox, A., Gribble, S.D., Brewer, E.A., Amir, E. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. *Proceedings of the Seventh International ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, October 1996.

[15] Frederking, R., Brown, R.D. The Pangloss-Lite Machine Translation System. *Expanding MT Horizons: Proceedings of the Second Conference of the Association for Machine Translation in the Americas*, Montreal, Canada, October 1996.

[16] Grimshaw, A.S., Liu, J.W. MENTAT: An Object-Oriented Data-Flow System. *Proceedings of the second ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Orlando, FL, October 1987.

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

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

[19] Haartsen, J. The Bluetooth Radio System. *IEEE Personal Communications*, 7(1):28–36, February 2000.

[20] Hoeim, D., Ke, Y., Sukthankar, R. SOLAR: Sound Object Localization and Retrieval in Complex Audio Environments. *Proceedings of the 30th IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Philadelpha, PA, March 2005.

[21] Hunt, G.C., Scott, M.L. The Coign Automatic Distributed Partitioning System. *Proceedings of the 3rd Symposium on Operating System Design and Implementation (OSDI)*, New Orleans, LA, Feb. 1999.

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

[23] Jul, E., Levy, H., Hutchinson, N., Black, A. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

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

[25] Katz, R.H. Adaptation and Mobility in Wireless Information Systems. *IEEE Personal Communications*, 1(1):6–17, 1994.

[26] Klemmer, S.R., Li, J., Lin, J., Landay, J.A. Papier-Mache: Toolkit Support for Tangible Input. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*, Vienna, Austria, April 2004.

[27] Ko, A.J., Aung, H.H., Myers, B.A. Eliciting Design Requirements for Maintenance-oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. *Proceeding of the 27th International Conference on Software Engineering (ICSE). To Appear*, St. Louis, MO, May 2005.

[28] Lakshminarayanan, K., Padmanabhan, V.N., Padhye, J. Bandwidth Estimation in Broadband Access Networks. *Proceedings of the 4th ACM/USENIX Internet Measurement Conference (IMC)*, Taormina, Sicily, Italy, October 2004.

[29] Narayanan, D., Satyanarayanan, M. Predictive Resource Management for Wearable Computing. *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, CA, May 2003.

[30] Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., Pignol, M. Generating Remote Control Interfaces for Complex Appliances. *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology (UIST)*, October 2002.

[31] Nielsen, J. *Usability Engineering*. Academic Press, San Diego, CA, 1993.

[32] Noble, B.D., Satyanarayanan, M., Narayanan, D., Tilton, J.E., Flinn, J., Walker, K.R. Agile Application-Aware Adaptation for Mobility. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[33] Pane, J.F., Myers, B.A. Usability Issues in the Design of Novice Programming Systems. Technical Report CMU-HCII-96-101, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 1996.

[34] Richard III, G.G. *Service and Device Discovery: Protocols and Programming*. McGraw-Hill Professional, 2002.

[35] Rinard, M.C., Lam, M. S. The Design, Implementation, and Evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, May.

[36] Rudenko, A., Reiher, P., Popek, G.J., Kuenning, G.H. Saving Portable Computer Battery Power through Remote Process Execution. *Mobile Computing and Communications Review*, 2(1):19–26, January 1998.

[37] Sailer, R., van Doorn, L., Ward, J.P. The Role of TPM in Enterprise Security. Technical Report RC23363(W0410-029), IBM Research, October 2004.

[38] Sailer, R., Zhang, X., Jaeger, T., van Doorn, L. Design and Implementation of a TCG-based Integrity Measurement Architecture. *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.

[39] Salonidis, T., Bhagwat, P., Tassiulas, L. Proximity Awareness and Fast Connection Establishment in Bluetooth. *Proceedings of the 1st ACM International Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc)*, Boston, MA, 2000.

[40] Satyanarayanan, M. Fundamental Challenges in Mobile Computing. *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing (PODC)*, Philadelphia, PA, May 1996.

[41] Schneiderman, H., Kanade, T. A Statistical Approach to 3D Object Detection Applied to Faces and Cars. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, Hilton Head Island, South Carolina, June 2000.

[42] Schulenburg, J. GOCR source code and online documentation. `http://jocr.sourceforge.net/`, Feb. 2004. (Version 0.39).

[43] Shneiderman, B. Empirical Studies of Programmers: The Territory, Paths, and Destinations. *Proceedings of First Workshop on Empirical Studies of Programmers*, Alexandria, VA, Jan 1996.

[44] Smailagic, A., Siewiorek, D. Application Design for Wearable and Context-Aware Computers. *IEEE Pervasive Computing*, 1(4), October-December 2002.

[45] TechSmith Corporation. *Camtasia Studio*. `http://www.techsmith.com/`, June 2004.

[46] The Walkthru Project. GLVU *source code and online documentation*. `http://www.cs.unc.edu/~walk/software/glvu/`, Feb. 2002. (Accessed on July 23 2002).

[47] Trusted Computing Group. *Trusted Platform Module Main Specification, Version 1.2, Part 1: Design Principles, Part 2: TPM Structures, Part 3: Commands*, October 2003. https://www.trustedcomputinggroup.org.

[48] Waibel, A. Interactive Translation of Conversational Speech. *IEEE Computer*, 29(7):41–48, July 1996.

[49] Waldo, J. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 42(7):76–82, 1999.

[50] Walko, J. *Handset sales reach new high in 2004*. EE Times. `http://www.eetimes.com/showArticle.jhtml;?articleID=59100009`, January 2005.

[51] Willmott, A.J. Radiator source code and online documentation. `http://www.cs.cmu.edu/~ajw/software/`, Oct. 1999. (Accessed on July 23 2002).

[52] Yao, P., Durant, D. Microsoft Mobile Internet Toolkit Lets Your Web Application Target Any Device Anywhere. *MSDN Magazine*, 17(11), November 2001.

[53] Zieniewicz, M.J., Johnson, D.C., Wong, D.C., Flatt, J.D. The Evolution of Army Wearable Computers. *IEEE Pervasive Computing*, 1(4), October-December 2002.