Tactics-Based Remote Execution for Mobile Computing

Rajesh Krishna Balan[†], Mahadev Satyanarayanan[†‡], SoYoung Park[†], Tadashi Okoshi[†]
[†]Carnegie Mellon University and [‡]Intel Research Pittsburgh
{rajesh,satya,seraphin,slash}@cs.cmu.edu

Abstract

Remote execution can transform the puniest mobile device into a computing giant able to run resource-intensive applications such as natural language translation, speech recognition, face recognition, and augmented reality. However, easily partitioning these applications for remote execution while retaining application-specific information has proven to be a difficult challenge. In this paper, we show that automated dynamic re-partitioning of mobile applications can be reconciled with the need to exploit application-specific knowledge. We show that the useful knowledge about an application relevant to remote execution can be captured in a compact declarative form called **tactics***. Tactics capture the full range of meaningful partitions of an application and are very small relative to code size. We present the design of a tactics-based remote execution system, Chroma, that performs comparably to a runtime system that makes perfect partitioning decisions. Furthermore, we show that Chroma can automatically use extra resources in an over-provisioned environment to improve application performance.*

1 Introduction

Remote execution can transform the puniest mobile device into a computing giant. This would enable resource-intensive applications such as natural language translation, speech recognition, face recognition, and augmented reality to be run on tiny handheld, wearable or body-implanted platforms. Nearby compute servers, connected through a low-latency wireless LAN, can provide the CPU cycles, memory, and energy needed for such applications.

Unfortunately, two annoying facts cloud this rosy future. First, the optimal partitioning of an interactive application into local and remote components is highly application-specific and platform-specific. Since mobile hardware evolves rapidly, this optimal partitioning changes on the timescale of months rather than years. Suboptimal partitioning can result in sluggish and intolerable interactive response. Hence, a tight and ongoing coupling between application developers and hardware platform developers appears inevitable. Second, matters are made worse by the fact that mobile environments exhibit highly variable resource availability. Bandwidth, energy and presence of compute servers can change on the timescale of minutes or hours, as a user moves to different locations. Re-partitioning an application for changed operating conditions at this timescale is therefore essential. These considerations suggest that an automated approach to partitioning applications for remote execution is necessary. However, partitioning an application without taking into consideration its unique characteristics may result in sub-optimal partitions.

Can automated dynamic re-partitioning be reconciled with the need to exploit application-specific knowledge? In this paper, we show that this is indeed possible. Our key insight is that *the knowledge about an application relevant to remote execution can be captured in compact declarative form that is very small relative to code size.* More specifically, the full range of meaningful partitions of an application can be described in a compact external description called *remote execution tactics* or just *tactics* for brevity. Thus, the tactics for an application constitute the limited and controlled exposure of application-specific knowledge necessary for making effective partitioning and placement decisions for that application in a mobile computing environment.

In this paper, we examine three applications of the genre mentioned earlier (natural language translation, speech recognition, and face recognition) and show that the tactics for each is much less than one percent of total code size. We present the design of *Chroma*, a tactics-based remote execution system, and show that sound partitioning and placement of these applications using tactics is possible. We show that Chroma is able to achieve application performance that is comparable to execution on an ideal runtime system.

In addition, we show that Chroma can opportunistically utilize extra resources in an over-provisioned environment. This allows us to achieve lower latencies for the three applications mentioned above.

The rest of this paper is organized as follows: Section 2 presents the assumptions and goals of this work while Section 3 presents the design of Chroma. We present our experimental setup in Section 4. Sections 5 and 6 present Chroma's performance relative to an ideal runtime system. In Section 7, we show how tactics can improve application performance in the presence of extra resources. Section 8 presents related work and Section 9 concludes the paper.

2 Design Rationale

2.1 Assumptions

In this work, we assume that all code necessary for remote execution is already present on all the clients and servers. We do not perform any code migration and use coarse-grained remote execution on the order of seconds. This granularity is appropriate for the class of applications being targeted. This is in contrast to other remote execution systems, like Java RMI [21], that perform fine-grained remote execution on the order of microseconds. We assume that the individual remote calls that make up the remote execution are self-contained and do not produce side effects.

Since Chroma is meant to be used on mobile devices, we assume a highly variable resource environment. Network characteristics and remote infrastructure available for hosting computation vary with location. File cache state and CPU load on local and remote machines significantly impact application performance. Application energy consumption varies depending upon the specific platform on which an application executes. Variation in any resource can significantly change the best placement of functionality. Thus, Chroma must continually monitor resource availability and adapt to changes in the environment.

The class of applications that we are targeting are computationally intensive interactive applications. Examples include speech recognition, natural language translation and augmented reality applications. These are the kinds of applications that have been envisioned as being key mobile applications in the near future [18, 23].

We assume that Chroma will not require applications to be developed from scratch. Instead Chroma will use existing applications that have been slightly modified to work with Chroma. This is a realistic assumption because building new applications from scratch requires huge amounts of effort. This is likely to be unprofitable when application development time becomes comparable to the useful lifetime of the wearable and/or handheld hardware being targeted. In this paper, we do not address the security and admission control issues involved in using remote servers.

2.2 Goals

Chroma was designed to achieve three major goals. These are:

- *Seamless from user perspective*: The user should be oblivious to the decisions being made by Chroma and the actual execution of those decisions.
- *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.
- *Minimal burden on application writers*: We want Chroma to be an easy system for application writers to use.

2.3 Solution Strategy

2.3.1 Seamless from user perspective

We achieved this goal by making Chroma completely automatic from the perspective of the application user. Chroma was designed to work with interactive applications which demand user attention due to their interactive nature. As such, Chroma was designed to require minimal additional user attention. The user specifies high-level preferences in advance to Chroma to guide its decision making process. 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.

2.3.2 Effectiveness

To achieve the best possible performance, Chroma should use the optimal strategy for remote execution for any particular resource condition. But how do we determine what that optimal strategy is? In theory, it is possible, for every resource condition, to test every single way of splitting an application for remote execution and then picking the best strategy. However, this is intractable in practice. Another method is just to pick one possible way of splitting up the application and using it all the time. However, this static method will be ineffective when resources change. The key insight that allows us to achieve our performance goal while keeping the search space small is this:

For every application, the number of useful ways of splitting the application for remote execution is small.

We call these useful ways of splitting the application the *tactics* of the application. Tactics are specified by the application developer and are high level descriptions of meaningful module-level partitions of an application. Our experience with modifying applications in the course of this work suggests that it is easy for an application developer to provide the tactics for an application.

An application is made up of *operations*. An operation is an application-specific notion of work. Tactics enumerate the various ways that an operation can be usefully executed. For example, an operation for a speech recognition application would be `recognize_utterance` while an operation for a graphics application would be

`render`. For each operation, the application developer specifies one or more tactics. These different tactics may differ in the amount of resources they use and their *fidelity* [15]. Fidelity refers to an application specific metric of quality. For example, speech recognition has higher fidelity when using a large vocabulary rather than a small vocabulary. Fidelity ranges from 0 to 1, with 1 being the best quality and 0 the worst.

2.3.3 Minimal Burden on Application Writers

Developing mobile computing applications is especially difficult because they have to be adaptive [6, 10]. The resource constraints of mobile devices, the uncertainty of wireless communication quality, the concern for battery life, and the lowered trust typical of mobile environments all combine to complicate the design of mobile applications. Only through dynamic adaptation in response to varying runtime conditions can applications provide a satisfactory user experience. Unfortunately, the complexity of writing and debugging adaptive code adds to the application software development time. Hence, instead of building the mechanisms to detect resource availability and trigger adaptation directly into each application, we created a runtime system that provides this functionality. However, the question still remains: How do we easily modify existing applications to use the adaptation features provided by our runtime?

Our approach to achieving this goal can be summarized as follows: First, we provide a lightweight semi-automatic process for customizing the API used by the application. Such customization is targeted to the specific needs of the application. Second, we provide tools for automatic generation of code stubs that map the customized API to the generic API used by Chroma. Finally, this generic API is supported by Chroma, which monitors resource levels and triggers application adaptation. Chroma support also helps ensure that the adaptations of multiple concurrently executing applications do not interfere with each other. Further details about the software engineering aspects of Chroma can be found elsewhere [2].

3 Chroma Design

In this section, we present the design of our tactics-based remote execution system, Chroma, that satisfies the goals described in Section 2. Building Chroma required two main components:

- A way of describing tactics.
- A method for selecting a tactic at runtime.

3.1 Describing Tactics

Figure 1 shows the tactics for Pangloss-Lite, a natural language translator. Pangloss-Lite uses up to three translation engines (*dictionary*, *ebmt* and *glossary*) to translate a sentence. The tactics specify the different ways of combining these engines and are composed of two distinct portions. Using more than one engine results in a better translation but at the cost of using more resources.

The first portion of the description (denoted by the keyword *RPC*) details the remote calls that can be used for this application. The second portion (denoted by the keyword *DEFINE_TACTIC*) defines the specific sequence of remote calls that make up a particular tactic. An "&" separator between remote calls denotes that the remote calls must be performed in sequential order while remote calls within brackets ((`server_gloss`, `server_ebmt`)) tell the remote execution system that those calls can be executed in parallel.

Each tactic fully describes one way of combining RPCs to complete an operation. The data dependencies between RPCs are visible because the prototypes of the remote calls are specified in the tactics description. Each of the individual remote calls that make up a particular tactic can be run either locally or on any remote server. This decision is made at runtime. Even though the tactics may differ in their resource usage and fidelity, each tactic is guaranteed to produce a proper result for the given operation if the remote calls are performed in the order specified by the tactic (we assume no side effects as mentioned in Section 2.1). Since the data dependencies and ordering between remote calls is fully specified by the tactic description, Chroma is able to parallelize the execution of these remote stages whenever possible. This aspect of Chroma is explained further in Section 3.3.

A key point to note is that the description of the application's tactics is very small compared to the size of the application. As shown in Figure 1, it requires about 14 lines to specify the tactics for Pangloss-Lite. This is in comparison to the roughly 150K lines of code in Pangloss-Lite.

3.2 Tactic Selection

In this section we highlight the system components necessary for Chroma to decide at runtime which tactic to use and where to execute it. For example, if Chroma picks the tactic `gloss_ebmt` (Figure 1) for Pangloss-Lite, it will also have to decide whether to execute the `server_gloss`, `server_ebmt` and `server_lm` remote calls of this tactic locally or remotely. Chroma's goal is thus to decide on a *tactic plan*. A tactic plan is comprised of a tactic number (denoting which tactic to use) along

```
RPC server_gloss (IN string line, OUT string gloss_out);
RPC server_dict  (IN string line, OUT string dict_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);

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

Pangloss-Lite has seven tactics that are listed after the DEFINE_TACTIC keyword. These seven tactics give different ways of combining the remote calls (listed after the keyword RPC) for this application. Each of these calls can be executed locally or at a remote server and this is determined at runtime by Chroma

Figure 1: Tactics for Pangloss-Lite

with a list that specifies the server to use for each RPC in that tactic. Using the local machine avoids network transmission and is unavoidable if the client is disconnected. In contrast, using a remote machine incurs the delay and energy cost of network communication but exploits the CPU and energy resources of a remote server. Chroma enumerates through all possible tactic plans and picks the best one for the given resource availability.

To be able to do this, first, Chroma needs to be able to predict the resource usage of each tactic plan. Second, Chroma has to measure the current resource availability. Third, Chroma requires guidance from the user about the relative importance of each resource. Given these three things, Chroma will be able to decide on the best tactic.

3.2.1 Resource Prediction

For a given operation and tactic plan, Chroma needs to be able to predict the resources the tactic plan will require. This information is provided by resource demand predictors that use history based prediction [14]. The key idea here is that the resource usage of a tactic plan can be predicted from its recent resource usage. The demand prediction mechanisms are initialized by off-line logging. At runtime, these predictors are updated using online monitoring and machine learning to improve accuracy.

3.2.2 Resource Monitoring

Chroma uses multiple resource measurers to determine current resource availability. These resource measurers currently measure memory usage, CPU availability, available bandwidth, latency of operation, file cache state and battery energy remaining. Chroma also has mechanisms to retrieve resource availability information from remote servers.

3.2.3 User Guidance

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 runs off to board their plane? That knowledge is very hard to obtain at the application level as it is user-specific and not application-specific.

We provide Chroma with these user-specific resource tradeoffs in the form of *utility functions*. A utility function is a user-specific function that quantifies the tradeoff between two or more attributes.

In this paper, we use a fixed utility function that states that latency is as important as fidelity and that Chroma should ignore battery lifetimes. Chroma will thus choose the tactic plan that maximizes the *latency-fidelity metric* (expressed mathematically as maximizing the quantity $\frac{fidelity}{latency}$). In our future work, we plan to develop methods that will allow us to capture different utility functions from the user using a graphical user interface. These different utility functions will allow us to optimize the tactic selection for other user specified metrics like conserving battery power or minimizing network bandwidth.

3.2.4 Selection Process

Figure 2 shows how all the components work together. Chroma determines expected resource demand for each

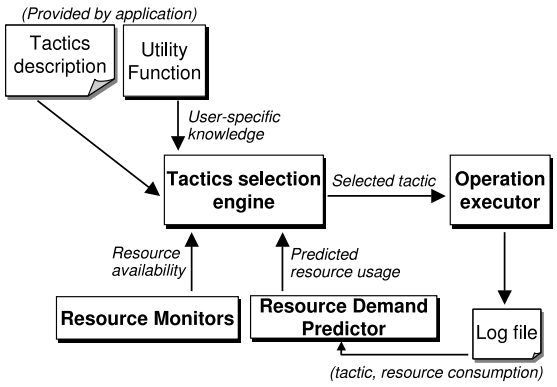
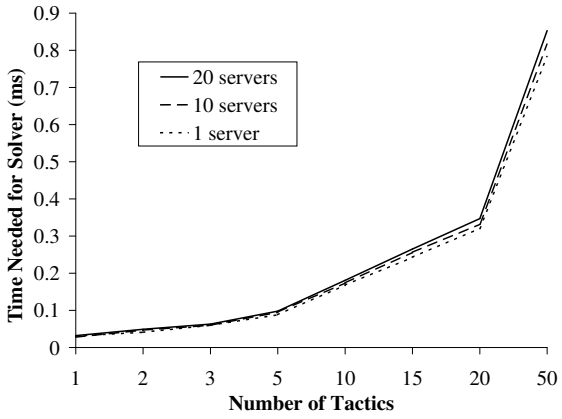


Figure 2: Choosing a Tactic



This figure shows the overhead incurred (in milliseconds) by the solver in deciding which tactic to choose. The overhead shown is only for the computational aspect of the solver and does not include the time needed by other parts of Chroma such as the resource estimators and resource demand predictors. To obtain these results, we extracted the core solver from Chroma and supplied it with synthetic inputs. This allowed us to measure just the overhead of the solver. The total measured overhead of Chroma is shown in Section 6.

Figure 3: Overhead of Choosing a Tactic

tactic of the current operation by querying the resource prediction component. At the same time, Chroma determines the available resources via the resource monitoring component. These resource monitors also query any available remote servers to determine the resource availability on those servers. This information is necessary as the latency of the tactic is determined by where each individual remote call in that tactic is being executed. Determining resource availability on demand can be a very time consuming operation. Hence, to improve performance at the cost of accuracy, the resource monitors perform these queries periodically in the background and cache the results.

Chroma iterates through every possible tactic plan and picks the best tactic plan to use for this operation. It does this by picking the tactic plan that maximizes the latency-fidelity utility function metric. The tactic plan is then executed and its resource usage is logged to refine future demand prediction. This brute force method works well for a small number of tactics as shown in Figure 3. From the results in Section 6, we see that the contribution of the solver to the total Chroma overhead is minimal. We claim that, in practice, the number of useful tactics for computationally intensive interactive applications is small enough to allow this brute force tactic selection mechanism. We are currently verifying this claim and also looking at using other solvers that are both less computationally demanding and provably correct [11].

3.3 Over-Provisioned Environments

Our discussion so far has focused on environments that are resource constrained. However, environments such as smart rooms, may be *over-provisioned*. Over-provisioned environments are characterized as having more computing resources than are actually needed for normal operation. We would like to have a system that works well if resources are scarce but is able to immediately make use of over-provisioning if it becomes available. Our goal is to exploit idle resources to improve user experience.

Tactics help by providing the knowledge of the remotes calls needed by a given operation and the data dependencies between them. Chroma can use the knowledge in tactics opportunistically to improve user experience in three different ways.

First, Chroma can make multiple remote execution calls (for the same operation) to remote servers and use the fastest result. For example, Chroma can execute the glossary engine of Pangloss-Lite at 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 the glossary engine is a stand-alone operation and does not require any previous results or state. We call this optimization method "fastest result".

Second, Chroma can split the work necessary for an operation among multiple servers. It does this by decomposing operation data into smaller chunks and shipping each chunk to a different remote server. Chroma uses hints from the application to determine the proper method of splitting operation data into smaller chunks. We call this optimization "data decomposition".

Third, 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 the `ebmt` engine of Pangloss-Lite in parallel at separate servers (all with different fidelities) and use the highest fidelity re-

sult that has returned before a specified amount of time. We call this optimization method "best fidelity".

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 we have explored so far and we present performance results for them in Section 7.

4 Validation Approach

4.1 Applications

To validate the design of Chroma, we have used three applications that are representative of the needs of a future mobile user. These applications are all computationally intensive interactive applications that are currently being actively developed for mobile environments. These applications are

- Pangloss-Lite [7] : A natural language translator written in C++ for translating sentences in one language to another. This kind of application is important for the modern mobile user who is moving from country to country.
- Janus [22] : A speech to text conversion program written in C that can be used to convert voice input into text. This kind of application is at the core of any voice recognition system that is used to control mobile devices.
- Face [20] : A program written in Ada that detects faces in images and is representative of image processing applications. Surveillance personnel, with wearable computers, that use images to detect suspicious features in the environment are likely to require this kind of application.

4.2 Experimental Platform

We used HP Omnibook 6000 notebooks with 256 MB of memory, a 20 GB hard disk and a 1 GHz Mobile Pentium 3 processor as our remote servers.

We 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. A deployed version of Chroma would use a wireless LAN such as 802.11a (55 Mb/s). We used the Coda [19] distributed file system to share application code between the clients and servers.

4.3 Success Criteria

To successfully validate Chroma, we need to show the following things:

- Chroma is able to correctly pick the best tactic plan for a particular application and resource availability. We demonstrate this by showing that Chroma picks the tactic plan that maximizes (or comes close to maximizing) the latency-fidelity metric.
- The overhead of Chroma's decision making process is not too large and does not add substantially to the total latency of the application.
- Chroma is able to use tactics to automatically improve application performance in the presence of additional server resources.

The validation of these three parts will justify our claim that tactics are a valuable. Sections 5, 6 and 7 present our results relative to the above points.

5 Results: Tactic Selection

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. We 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.

To demonstrate this, we compare the decision making of Chroma with that of an ideal runtime system. This ideal runtime system is achieved by manually testing every possible tactic plan for a given experiment and then choosing the best one. Chroma, on the other hand, has to figure out the best tactic plan dynamically at runtime. We define the best tactic plan as being the one that maximizes the latency-fidelity metric. We show that Chroma chooses a tactic plan that either maximizes the latency-fidelity metric or comes very close to it.

Each experiment was repeated five times and our results are shown with 90% confidence intervals where applicable. Since Chroma uses history-based demand prediction, we 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.

Sentence Length (No. of Words)	Ideal Runtime		Chroma		Ratio
	chosen tactic	metric	chosen tactic	metric	
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	0.70	dict_ebmt	0.70	1.00
59	dict_ebmt	0.70	dict_ebmt	0.70	1.00

(a) Fast Client

Sentence Length (No. of Words)	Ideal Runtime		Chroma		Ratio
	chosen tactic	metric	chosen tactic	metric	
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

This table shows the tactic plan 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. We also show the value of the latency-fidelity metric for the tactic plans chosen by the two systems. The ratio ($\frac{Chroma}{Ideal}$) between the ideal system's metric and Chroma's is shown in the Ratio column.

Figure 4: Comparison Between the Ideal Runtime and Chroma for Pangloss-Lite

5.1 Pangloss-Lite

5.1.1 Description

As mentioned in Section 3.1, Pangloss-Lite 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.

Pangloss-Lite's fidelity increases with the number of engines used for translation. We assign the EBMT engine a fidelity of 0.5. The glossary and dictionary engines produce subjectively worse translations—we assign them fidelity levels of 0.3 and 0.2, respectively. When multiple engines are used, we add 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, we assign a fidelity of 0.8. The seven possible combinations of the engines are captured by the seven tactics (shown in Fig 1).

We use the latency-fidelity utility function to determine the tactic to use for Pangloss-Lite. However, to model the preferences of an interactive user, we specify that all latencies of one second or lower are equally good and that all latencies larger than five seconds are impossibly bad. Thus if the latency is greater than five seconds, we set the latency to a really large number (thus making the utility value really small) and if the latency is one second or lower, we set the latency value to one. All other latency values are left unchanged.

All three engines and the language modeler may be executed remotely. While execution of each engine is optional, the language modeler must always execute. Thus, there are at least 52 tactic plans from which Chroma may choose when at least one remote server is available.

We 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.

5.1.2 Results

Figures 4 displays the decisions made by Chroma compared with the decisions made by the ideal runtime for each sentence 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

Utterance	Ideal Runtime		Chroma		Ratio
	chosen tactic	metric	chosen tactic	metric	
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
	chosen tactic	metric	chosen tactic	metric	
1	reduced	0.50	reduced	0.50	1.00
2	reduced	0.50	reduced	0.48	0.96
3	reduced	0.50	reduced	0.50	1.00
4	reduced	0.50	reduced	0.50	1.00
5	reduced	0.50	reduced	0.49	0.98
6	reduced	0.50	reduced	0.50	1.00
7	reduced	0.50	reduced	0.50	1.00
8	reduced	0.42	reduced	0.41	0.98
9	reduced	0.50	reduced	0.50	1.00
10	reduced	0.50	reduced	0.48	0.96

(b) Slow Client

This table shows the tactic plan 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. We also show the value of the latency-fidelity metric for the tactic plans chosen by the two systems. The ratio ($\frac{Chroma}{Ideal}$) between the ideal system's metric and Chroma's is shown in the Ratio column.

Figure 5: Comparison Between the Ideal Runtime and Chroma for Janus

```
RPC do_full_recognition
                (IN string utterance,
                OUT string translation);

RPC do_reduced_recognition
                (IN string utterance,
                OUT string translation);

DEFINE_TACTIC full_recognition =
                 do_full_recognition;

DEFINE_TACTIC reduced_recognition =
                 do_reduced_recognition;
```

The tactics declaration for Janus contains two remote calls (do_full_recognition and do_reduced_recognition) that can be run either locally or remotely.

Figure 6: Tactics for Janus

for Chroma to predict as it was not a simple function of the length of the input sentence. 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 metric were below 1 second for both Chroma and the ideal runtime system in all the cases where the metrics differed.

5.2 Janus

5.2.1 Description

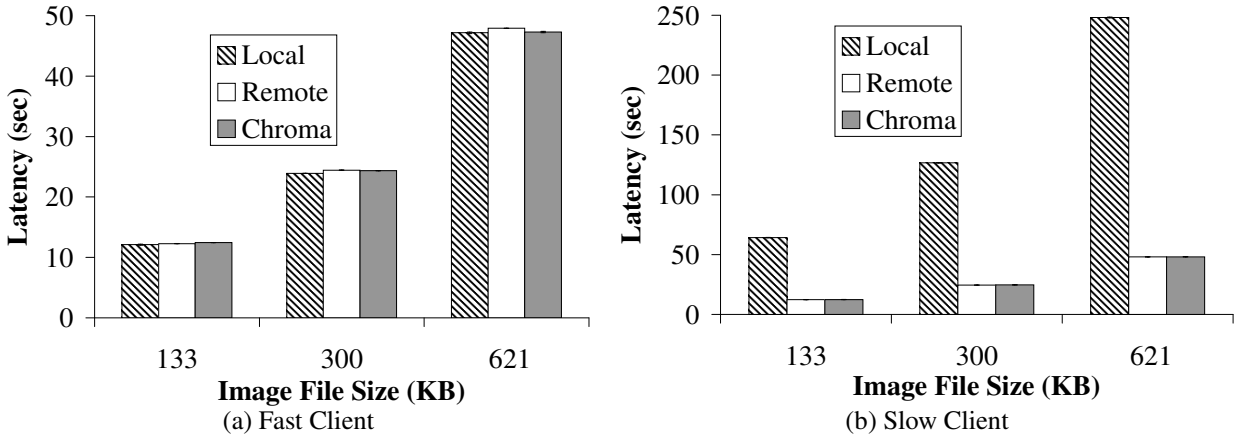
Janus performs speech-to-text translation of spoken phrases. Recognition can be performed at either full or reduced fidelity. The reduced fidelity uses a smaller, more task-specific vocabulary that limits the number of phrases that can be successfully recognized but requires less time to recognize a phrase. We assign the reduced fidelity a utility of 0.5 and the full fidelity a utility of 1.0 to reflect this behavior. Similar to Pangloss-Lite, we model an interactive user by making all latencies less than or equal to one second equally good (we set the latency value to one) and all latencies greater than five seconds horribly bad (we set the latency to a really large number). All other latency values are left unchanged.

Janus has two remote calls that can be executed either locally or remotely. These two possible ways of executing Janus are captured by Janus's tactics, as shown in Figure 6. The tactic `full_recognition` uses the full fidelity vocabulary to do the recognition while the tactic `reduced_recognition` uses the reduced fidelity vocabulary to do the recognition. Describing Janus's tactics requires 4 lines of code in our declarative language. This is significantly smaller than Janus itself which is ≈120K lines of C code.

We 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.

5.2.2 Results

Figure 5 shows the decisions made by the ideal runtime and Chroma. 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 latency-fidelity metric 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 metric were due to experimental errors in the latency measurements.



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 latency-fidelity metric as the fidelity was constant in all cases.

Figure 7: Relative Latency for Face

```
RPC detect_face (IN file in_image_name,
                 OUT file out_image_name);

DEFINE_TACTIC detect = detect_face;
```

Face has only one remote call (detect_face) that can be run either locally or remotely. This is captured by its single tactic.

Figure 8: Tactics for Face

5.3 Face

5.3.1 Description

Face 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 can be run either entirely locally or entirely remotely. In both cases, it runs the exact same remote procedure and it has no other modes of operation. It thus has only one tactic and this is shown in Figure 8. Even though Face has only one tactic, this does not mean that it cannot benefit from tactics. We show in Section 7.2 how Chroma can use this single tactic to improve the performance of Face by using extra resources in the environment. Face is written in Ada and has ≈20K lines of code while the description of its tactics requires just 2 lines.

We 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.

5.3.2 Results

Figure 7 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 latency-fidelity metric would require Chroma to pick the option that minimized the latency. We see that in all cases, Chroma chose the option that maximized the latency-fidelity metric by picking the tactic plan that minimized the latency.

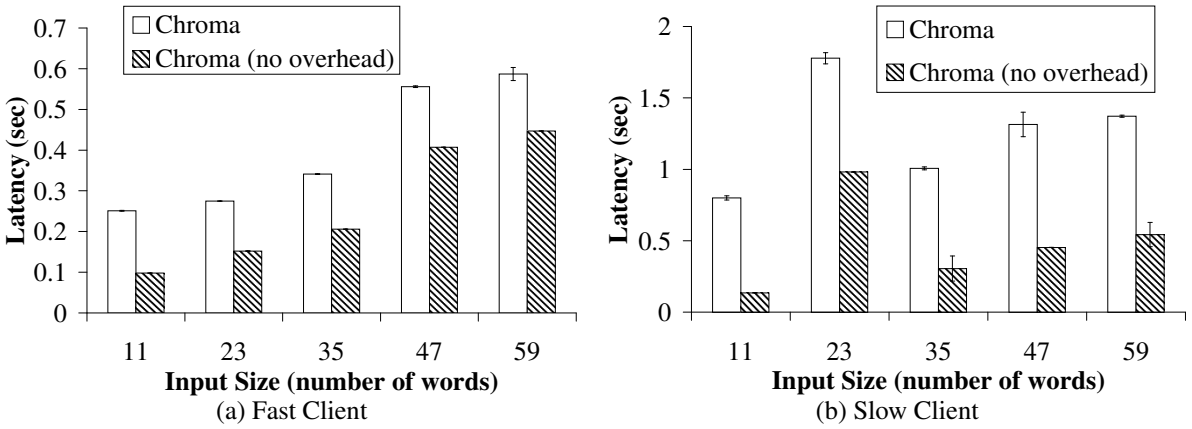
The graphs show that Face has extremely high latencies; on the order of tens of seconds per image. We will show how tactics allow us to reduce this latency without sacrificing fidelity in Section 7.2.

5.4 Summary

Sections 5.1, 5.2 and 5.3 described the performance of Chroma relative to an ideal runtime system for Pangloss-Lite, Janus and Face respectively. We see that while Chroma is not perfect, its performance is still comparable to an ideal runtime system. We believe that the results indicate that it is viable to build a tactics-based remote execution system that provides good application performance.

6 Results: Chroma's Overhead

In this section, we present the CPU overhead of Chroma's decision making using Pangloss-Lite as the example application. Pangloss-Lite has the largest number of tactic plans among all the applications used in this paper and required Chroma to do the most decision making. As such, we do not present the overhead results for the other applications as they were strictly less than the



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 4 for the different inputs. The results are the average of 5 runs and are shown with 90% confidence intervals.

Figure 9: Overhead of Decision Making for Pangloss-Lite

overhead incurred for Pangloss-Lite.

Figure 9 shows the overhead of Chroma's decision making. This overhead represents the time that Chroma needs to determine the tactic plan to use. Chroma currently does not take its own overhead into account when making placement decisions and thus can achieve longer latencies than it 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 less than 0.5 seconds. This overhead, while somewhat high, was still acceptable for the class of applications being targeted. We are currently improving the internal algorithms used in Chroma to reduce this overhead.

7 Results: Over Provisioning

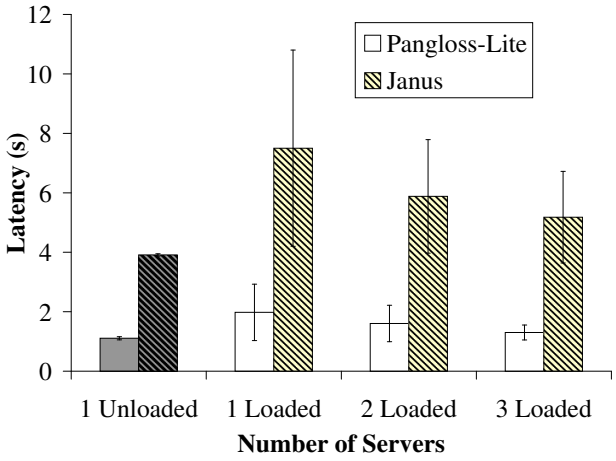
In this section, we show the performance improvements that Chroma achieves by opportunistically using extra resources in the environment. These extra resources take the form of extra available servers that can be used for remotely executing application components. We used the slow client for these experiments.

To show the benefits of this approach, we 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. We 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.

The overall scenario we 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. We 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. These extra servers can be used in the three ways detailed in Section 3.3 to allow us to:

- Hedge against load spikes at the remote servers: the same operation can be run on multiple servers using the "fastest result" method.
- 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.
- 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.

It should be noted again that all these methods can be used automatically at runtime by Chroma without the ap-



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

Figure 10: Using Extra Loaded Servers to Improve Latency

plication being aware of them. This is one of the key benefits of using a tactics-based remote execution system.

7.1 Hedging Against Load Spikes

7.1.1 Description

This experiment shows how opportunistically using extra servers in the environments provides protection against random load spikes at any particular remote server. In this experiment, Chroma decides to execute the glossary engine of Pangloss-Lite remotely to translate a sentence containing 35 words. We ran the translation of this sentence 100 times using a different number of remote servers in parallel and noted the average latency achieved and the standard deviation.

7.1.2 Results

Figure 10 shows the results we 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 10 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 the best possible results.

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. Our 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. We are currently studying various resource management algorithms to ensure fair usage of extra servers. We are also looking at mechanisms to allow the user to explicitly specify (if necessary) which extra servers can be used and which should not.

7.2 Reducing Latency by Decomposition

7.2.1 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. As shown in Figure 7, 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.

Here, we assume that the application has previously provided Chroma with the methods for splitting and recombining the image files. Given these methods, at run-

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.

Figure 11: Improvement in Face Latency by Decomposition

time, Chroma is able to automatically split the input images to improve application performance when extra servers become available.

7.2.2 Results

Figure 11 shows the results obtained by using this method. We 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 the operation) results in a substantial latency improvement (up to 60% reduction) over the original latency.

7.3 Meeting Latency Constraints

7.3.1 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 Pangloss-Lite, 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.

7.3.2 Results

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

Figure 12 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.

7.4 Summary

We have presented three different 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 7.2), 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.

8 Related Work

There have been a number of application-aware remote execution systems such as Abacus [1], Coign [3] and Condor [8]. They perform well in environments where resource availability does not change between the time the system decides how to remotely execute an application and when it actually performs the remote execution.

However, this assumption comes under fire in mobile environments. These environments are characterized by highly variable resource conditions that change on the order of seconds [5, 6, 17]. Overcoming this uncertainty requires application-specific knowledge on how to remotely partition the application.

An extra benefit of acquiring this knowledge is that it allows us to utilize additional resources in over-provisioned environments such as smart spaces with many idle compute servers. We envision that these environments will become increasingly common in the new future. Our system is designed to opportunistically

	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.

Figure 12: Achieving Latency Constraints for Pangloss-Lite

use these extra resources to improve application performance. We know of no other system that does this.

There have been other systems that have looked at the problem of partitioning applications. These include systems that performed object migration like Emerald [9] and systems that performed process migration [13]. Other systems [16] looked at the problem of service composition or the building of useful applications from components available in the environment. Currently, we have concentrated on the problem of identifying useful remote execution partitions of existing applications and have not performed any form of code migration or service composition.

The declarative language we use to express an application's tactics addresses some of the same issues as 4GLs [12] and 'little "languages" [4]. The latter are task-specific languages that allow developers to express higher level semantics without worrying about low level details. Our language is similar as it allows application developers to specify the remote execution capabilities of their applications at a higher level without needing to worry about low level system integration details. However, our approach is focused towards remote execution systems for mobile computing.

9 Conclusion

In this paper, we introduced the concept of tactics. This abstraction captures application-specific knowledge relevant to remote execution with minimal exposure of the implementation details. This allows the use of computationally intensive applications on handheld and wearable devices even in environments with changing resources. We showed how tactics can be used to build a remote execution system. We also provided experimental results from three applications to confirm the benefits of using tactics.

Currently, we are looking at methods of resource allocation to ensure that servers are used fairly by Chroma clients. We are also looking at various service discovery mechanisms to allow us to easily discover the presence of these servers. Finally, we are developing better software engineering methods to ease application development.

10 Acknowledgments

This research was supported by the National Science Foundation (NSF) under contracts CCR-9901696 and ANI-0081396, the Defense Advanced Projects Research Agency (DARPA) and the U.S. Navy (USN) under contract N660019928918. Rajesh Balan was additionally supported by a USENIX student research grant. We would like to thank Hewlett-Packard for donating the notebooks to be used as the servers and Compaq for donating handhelds used as clients. Finally, we would like to thank Mukesh Agrawal, Jan Harkes, Urs Hengartner, Ningning Hu, Glenn Judd, Mahim Mishra, Bradley Schmerl, Joao Sousa, the anonymous MobiSys reviewers and our shephard Marvin Theimer for their many insightful comments and suggestions related to this work. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, DARPA, USN, HP, USENIX, Compaq, or the U.S. government.

References

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

[2] Balan, R. K., Sousa, J. P., and Satyanarayanan, M. Meeting the software engineering challenges of adaptive mobile applications. Technical Report CMU-CS-03-111, Carnegie Mellon University, Pittsburgh, Pennsylvania, Feb. 2003.

[3] Basney, J. and Livny, M. Improving goodput by coscheduling CPU and network capacity. *Intl. Journal of High Performance Computing Applications*, 13(3), Fall 1999.

[4] Bentley, J. Little languages. *Communications of the ACM*, 29(8):711–21, 1986.

[5] D. Eckhardt, P. S. Measurement and analysis of the error characteristics of an in-building wireless network. *Proceeding of ACM SIGCOMM*, pages 243–254, Stanford, California, October 1996.

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

[7] Frederking, R. and 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, pages 268–272, Montreal, Canada, 1996.

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

[9] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the emerald system. *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, volume 21, pages 105–106, 1987.

[10] Katz, R. H. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1):611–17, 1994.

[11] Lee, C., Lehoczky, J., Siewiorek, D., Rajkumar, R., and Hansen, J. A scalable solution to the multi-resource QoS problem. *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS '99)*, pages 315–326, Phoenix, AZ, Dec. 1999.

[12] Martin, J. *Fourth-Generation Languages*, volume 1: Principles. Prentice-Hall, 1985.

[13] Milojicic, D. S., Douglis, F., Paindaveine, Y., Wheeler, R., and Zhou, S. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[14] Narayanan, D. and Satyanarayan, 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.

[15] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. Agile application-aware adaptation for mobility. *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 276–287, Saint-Malo, France, October 1997.

[16] Raman, B. and Katz, R. An architecture for highly available wide-area service composition. *Computer Communications Journal, special issue on 'Recent Advances in Communication Networking'*, May 2003.

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

[18] Satyanarayanan, M. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, Aug. 2001.

[19] Satyanarayanan, M. The evolution of coda. *ACM Transactions on Computer Systems (TOCS)*, 20(2):85–124, 2002.

[20] Schneiderman, H. and 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)*, pages 746–751, Hilton Head Island, South Carolina, June 2000.

[21] Sun Microsystems Inc. *Remote Method Invocation Specification*.

[22] Waibel, A. Interactive translation of conversational speech. *IEEE Computer*, 29(7):41–48, July 1996.

[23] Weiser, M. The computer for the twenty-first century. *Scientific American*, pages 94–101, September 1991.