Balancing Performance, Energy, and Quality in Pervasive Computing

Jason Flinn[†‡], SoYoung Park[†], and M. Satyanarayanan[†‡]
[†]Carnegie Mellon University, [‡]Intel Research Pittsburgh

Abstract

We describe Spectra, a remote execution system for battery-powered clients used in pervasive computing. Spectra enables applications to combine the mobility of small devices with the greater processing power of static compute servers. Spectra is self-tuning: it monitors both application resource usage and the availability of resources in the environment, and dynamically determines how and where to execute application components. In making this determination, Spectra balances the competing goals of performance, energy conservation, and application quality. We have validated Spectra's approach on the Compaq Itsy v2.2 and IBM ThinkPad 560X using a speech recognizer, a document preparation system, and a natural language translator. Our results confirm that Spectra almost always selects the best execution plan, and that its few suboptimal choices are very close to optimal.

1 Introduction

Creating applications for pervasive computing environments is a challenging task. Such environments are saturated with computing and communication capability, yet need to be gracefully integrated with human users [17]. The need for graceful integration leads to smaller, less obtrusive computing devices. Since people are naturally mobile, these devices are often battery powered, and their size limits the amount of energy that can be stored. Device size also constrains compute power and storage capacity. Remote execution using wireless networks to access compute servers fills a natural role in pervasive computing; it allows applications to combine the mobility of small devices with the greater processing power of large servers in the fixed infrastructure.

We envision a future in which support for mobile clients varies widely with location. Some well-conditioned environments may provide plentiful wireless bandwidth and powerful compute servers. Other locations may be resource-impoverished, with poor connectivity and little infrastructure support. Pervasive applications must thus adapt to a changing environment. When little infrastructure exists, they must execute most functionality on the mobile client. However, when the environment is well-conditioned, they should discover and use the resources offered.

We also envision a broad range of resource-hungry applications such as speech recognition, natural language translation, and augmented reality becoming important in pervasive computing. These applications may be executed on a wide range of mobile hardware, from powerful laptops to wristwatch-sized devices. Variation in the environment and in client capability makes it quite difficult for developers to statically determine which components of an application should execute remotely. Instead, we propose that location decisions be made *dynamically* when applications execute.

In this paper, we describe Spectra, a remote execution system that simplifies the task of developing applications for pervasive computing. Applications statically specify which code components might *possibly* profit from remote execution. When applications execute, Spectra continually monitors resource supply and demand to advise them how and where they should execute the specified components. Spectra's advice lets applications adapt to changes in resource availability without requiring them to explicitly specify their resource requirements. Instead, Spectra monitors application behavior, models resource usage, and predicts future resource needs.

2 Design considerations

2.1 Competing goals for functionality placement

In pervasive computing environments, a remote execution system must reconcile multiple, possibly contradictory goals. Performance is not the sole consideration; it is also vital to conserve energy so as to prolong battery lifetime. Quality must also be considered explicitly, since a resource-poor mobile device may only be able to provide a low fidelity version of a data object or computation [14], while a stationary machine may be able to generate a better version. These goals will often conflict—for example, executing a code component on a remote server might reduce client energy usage at the cost of increasing execution time.

Spectra explicitly considers these competing goals when advising applications where to place functionality. For each alternative placement, it predicts application performance, energy use, and quality. It then balances competing goals when selecting from the alternatives—for example, it would prefer an alternative that offers significant performance improvement at the cost of slightly increased energy use.

2.2 Variation in resource availability

Pervasive computing is characterized by tremendous variation in resource availability. 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, a remote execution system must continually monitor resource availability and adapt to changes in the environment.

Spectra includes a set of *resource monitors* that measure local and remote resource availability. Each monitor measures a single resource or set of related resources—for example, the network monitor reports available bandwidth and latency to remote servers. The monitors are implemented in a modular framework, which enables us to easily add measurement capability for new resources.

2.3 Self-tuning operation

To predict the time and energy needed to execute a code component, the remote execution system must match resource availability with the resource demands of the component. For example, the time needed to transfer data over a network can be roughly predicted by dividing the amount of data that the application will transfer by the available network bandwidth. Thus, a remote execution system must have a model of application resource demand in order to make correct placement decisions.

One approach would be to require applications to explicitly specify resource requirements. However, this seems infeasible for most applications. Many resources are important in pervasive computing, and an application would have to specify models for each. Also, models for some resources are platform specific—application energy usage, for instance, depends upon hardware characteristics. The burden of specifying such models seems too high—we believe that few developers would be willing to invest the effort.

Spectra takes an alternate approach, which we call *self-tuning*. It observes applications execute, measures their resource usage, and generates models of resource consumption. It then uses the models to predict future demand.

2.4 Modification to application source code

Systems such as Coign [7] support remote execution without source-code modification by exploiting externally visible object interfaces. This approach is attractive because it supports closed-source applications and requires little development effort. However, we believe that a little application-specific knowledge can go a long way.

Spectra asks developers to specify possible methods of partitioning applications. Often there will be only a few useful partitions. While developers will often have an intuitive notion of which partitions are useful, it may prove difficult to automatically extract these partitions from source code. First, the best partitions may not occur along object boundaries. Second, since the number of potential partitions is exponential with the number of objects, selecting partitions for large applications may be computationally intractable. Finally, it will be impossible to support legacy applications which do not provide externally visible object interfaces.

Application-specific knowledge improves models of resource demand. Often resource usage will depend upon a few application-specific parameters—for example, the amount of CPU cycles required to translate a sentence from Spanish to English often depends upon the length of the sentence to be translated. Spectra allows applications to specify the important parameters that affect the complexity of the operations they perform. Spectra derives more accurate predictions by anticipating the effect of these parameters.

We have tried to situate Spectra in a sweet spot of the design space. In exchange for a minimal amount of application modification, Spectra provides significantly better advice than could be provided without modification.

2.5 Granularity of remote execution

Fine-grained remote execution yields increased flexibility since it creates more opportunities to locate functionality on remote servers. However, coarse-grained remote execution may lead to better performance by amortizing overhead over a larger unit of execution.

Since Spectra is designed to make intelligent placement decisions that balance competing goals and adjust to changes in resource availability, its decision overhead is non-negligible. Therefore, Spectra targets applications that perform relatively coarse-grained operations—currently a second or more in duration. Examples of such applications include speech recognition and language translation. Applications that perform shorter operations, i.e. a few milliseconds in duration, will not benefit from Spectra since system overhead will negate the performance and energy benefits achieved by making correct location decisions.

2.6 Support for remote file access

Many of Spectra's target applications access a large amount of file data. For remote execution to yield correct results, file operations performed on remote servers must produce the same results that would be produced if the operation were performed on the client. Systems such as Butler [13] solve this problem by using a distributed file system that presents a consistent file system image across all machines on which functionality may be executed.

Unfortunately, file consistency comes at significant cost in pervasive computing environments. Network connections to file servers often exhibit high latency, low bandwidth, or intermittent connectivity. When an application modifies files, it will block for long periods of time waiting for data to be reintegrated to file servers.

`register_fidelity`	Describes an application operation.
`begin_fidelity_op`	Signals operation start.
`do_local_op`	Makes a RPC to the local Spectra server.
`do_remote_op`	Makes a RPC to a remote Spectra server.
`end_fidelity_op`	Signals operation end.

Figure 1. Spectra API

This performance consideration led us to adopt the Coda file system [9]. Coda provides strong consistency when network conditions are good. Under low bandwidth conditions, Coda buffers file modifications on the client to improve performance. Buffered modifications are reintegrated to file servers in the background. Until a modification is reintegrated, it is not visible on other machines.

Spectra interacts with Coda to provide the requisite consistency for remote execution. Before it executes functionality remotely, Spectra predicts which files will be accessed by the application. If any such files have buffered modifications on the client, Spectra triggers the reintegration of modified file data to servers before executing the operation.

3 Implementation

Spectra consists of a client, which executes on the same machine as the application, and a server, which executes on machines that may perform work on behalf of clients. It is common for a single machine to run both client and server. The client is tightly integrated with Odyssey [14], a platform for applications that vary *fidelity*. Fidelity is an application-specific metric of quality. For example, fidelities for a video player are lossy compression and frame rate; for a speech recognizer, vocabulary size is a fidelity.

3.1 Application interface

Figure 1 shows the Spectra API. An application statically identifies *operations*: code components that may benefit from remote execution. When the application starts executing, it registers its operations using `register_fidelity`. For each operation, it specifies a set of possible *execution plans*, which represent different methods of partitioning functionality between local and remote machines. The application also specifies the possible fidelities at which the operation may be performed, as well as *input parameters*, variables that significantly affect operation complexity.

For example, we have modified a speech recognition application to use Spectra. It has one operation, recognition of a spoken utterance, with three possible execution plans: local, remote, and hybrid. The local plan performs computation entirely on the client, the remote performs computation entirely on a server, and the hybrid plan represents a specific split of computation between local and remote machines. The operation has one fidelity, the number of words in the vocabulary used for recognition, and one input parameter, the length of the utterance to be recognized.

```
service_init (&argc, &argv);
while (1) {
    service_getop (&optype, &opid, path,
                   &indata, &inlen);
    rc = do_operation (indata, inlen,
                       &outdata, &outlen);
    service_retop (opid, 0, outdata, outlen);
}
```

Figure 2. Sample service implementation

Applications call `begin_fidelity_op` to determine how and where each operation should execute. The application specifies the values of any input parameters, for example, the length of the utterance to be recognized. Odyssey chooses the fidelity at which the operation will execute, and Spectra chooses which execution plan will be used. When the execution plan involves a remote server, Spectra also chooses a particular server to use.

Applications use the `do_local_op` and `do_remote_op` calls to make remote procedure calls (RPCs) to local and remote Spectra servers. While applications could invoke remote functionality directly, these calls let Spectra measure the amount of local and remote resources used by the operation. Additionally, Spectra acts as a central repository for information about server status.

Operations often consist of multiple RPCs. For example, the speech hybrid plan uses both `do_local_op` and `do_remote_op` calls to perform the operation. Applications signal the end of operation execution by calling `end_fidelity_op`. Marking the start and end of operation execution allows Spectra to precisely measure resource use.

3.2 Invoking remote functionality

Spectra clients maintain a database of servers willing to host computation. The database stores snapshots of recent status (availability, CPU load, file cache state, etc.). Currently, potential servers are statically specified in a configuration file. We have designed Spectra so that it could also use a service discovery protocol [1, 20] to dynamically locate additional servers, but this feature is not yet supported.

Application code components executed on Spectra servers are referred to as *services*. Each service executes as a separate process to protect against malicious or faulty application code. Spectra provides an application library that simplifies service implementation. Figure 2 shows the main loop of a simple service. The `service_init` function parses the command line and extracts Spectra-specific information. In the main loop, `service_getop` blocks until a request is received. The function returns the type of operation requested, a unique identifier associated with the request, and application-specific input data. The sample service has only one request type—services that handle more than one type of request multiplex on `optype`. When processing is complete, the service calls `service_retop`, passing in the request identifier and application-specific output data.

3.3 Resource monitors

Spectra measures supply and demand for many different resources in order to make correct location decisions. Its measurement functionality is implemented as a set of *resource monitors*, code components that measure a single resource or a set of related resources. The monitors are contained within a modular framework shared by Spectra clients and servers. The modular design makes it easy to add new measurement capability. Further, it allows us to implement several different methods of measuring a particular resource and choose the appropriate method for each execution environment.

Currently, Spectra has six types of monitors: CPU, network, battery, file cache state, remote CPU, and remote file cache state. Each monitor provides a common set of functions. Prior to executing an operation, Spectra generates a *resource snapshot* that provides a consistent view of the local and remote resources available for execution. Spectra first generates a list of servers that could possibly execute some portion of the operation. For each server in the list, Spectra iterates through the set of resource monitors, calling `predict_avail`. Each monitor returns predicted resource availability—for example, the network monitor returns predicted network bandwidth and latency for communicating with the specified server. The predictions in the snapshot are used to decide how and where to execute the operation.

During operation execution, the resource monitors observe application resource usage. Before executing an operation, Spectra calls `start_op`, which alerts each monitor to begin observation. After the operation completes, Spectra calls `stop_op`, which terminates measurement and returns the amount of resources consumed. The `add_usage` function accounts for resource usage on Spectra servers—the monitor adds reported resource consumption to the total resource usage of the operation.

Spectra logs resource usage and creates models that predict future demand. Thus, the more an operation is executed, the more accurately its resource usage is predicted.

3.3.1 The CPU monitor

The CPU monitor predicts availability using a smoothed estimate of recent load. The monitor first determines the amount of competition for the CPU by measuring the percentage of cycles recently used by other processes. It then calculates the percentage of cycles available for operation execution by assuming that background load will remain unchanged and that the operation will get a fair share of the CPU. It multiplies this value by the processor speed to predict the cycles per second the operation will receive. Narayanan et al. [12] describe this algorithm in more detail.

The monitor observes CPU usage by associating an operation with the identifier of the executing process. This distinguishes the CPU usage of concurrent operations. Before and after execution, the monitor observes CPU statistics for the executing process and its children using Linux's `/proc` file system. It returns total cycles used by the operation.

3.3.2 The network monitor

The network monitor predicts bandwidth and latency using an algorithm first developed for Odyssey [14]. Predictions are based upon passive observation of communication. The RPC package logs the sizes and elapsed times of short exchanges and bulk transfers. The short, small RPCs give an approximation of round trip time, while the long, large bulk transfers approximate throughput. The network monitor periodically examines recent transmission logs and determines the instantaneous bandwidth available to the entire machine. It then estimates how much of that bandwidth is likely to be available for communicating with each server, assuming that the first hop is the bottleneck link.

Observing network usage is trivial since all client-server communication passes through Spectra. For each operation, the network monitor records the number of bytes sent and received, as well as the number of RPCs performed.

3.3.3 The battery monitor

When asked to predict availability, the battery monitor returns the amount of energy remaining in the client's battery. It also returns an estimate of the current importance of energy conservation, which is determined by goal-directed adaptation [4]. Using this technique, the user estimates how long the mobile computer will need to operate on battery power. The system monitors energy supply and demand, and adjusts a global feedback parameter that represents the importance of energy conservation.

During operation execution, the monitor measures energy consumption. Since it is difficult to distinguish the energy usage of concurrent operations, Spectra ignores data gathered from concurrently executing operations when modeling demand and predicting future energy needs.

Spectra obtains energy measurements from two sources: the Advanced Configuration and Power Interface [8] and SmartBattery [18] device drivers. Each source is supported by a separate resource monitor—this modular design makes it easy to select the appropriate measurement methodology when compiling for different hardware platforms.

3.3.4 The file cache state monitor

The Coda file system hides server access latency by caching files on clients. If data is cached locally, operations can take significantly less time and energy to execute. To predict this effect, the file cache state monitor asks Coda which files are in its cache. Although it is possible that cache state will change slightly during operation execution, the changes are unlikely to be significant. The monitor also obtains an estimate of the rate at which uncached data will be fetched.

During operation execution, the monitor observes Coda file accesses and returns the names and sizes of files accessed. Section 3.5 describes how this data is used to predict which files will be accessed during future operations.

3.3.5 The remote proxy monitors

Resource monitors on Spectra servers measure CPU and file cache state. They communicate this information to *remote proxy monitors* running on Spectra clients. Each client periodically polls servers to obtain a snapshot of resource availability. It then calls the `update_preds` function of each remote proxy monitor to update server status.

When Spectra executes a RPC, server monitors observe resource usage and report the total resource consumption as part of the RPC response. The Spectra client passes this data to proxy monitors by calling the `add_usage` function. The proxy monitors accumulate server resource consumption and report the total when the operation completes.

3.4 Predicting resource demand

Spectra builds on previous work in history-based resource prediction [12] by using measurements of application resource usage to generate models that predict future demand. It assumes that the resources usage of an operation will be similar to the amount used by recent operations of similar type.

Spectra provides default *predictors* that model resource demand. For numerical data such as CPU and energy usage, the default predictor generates simple linear models of application behavior. The default file predictor is somewhat more complicated, and is described in the next section. We believe that the default predictors will usually prove sufficient; in fact, we use them for all current applications. However, Spectra also provides an interface through which application-specific predictors may be specified.

When an application calls `register_fidelity`, Spectra creates predictors for each resource type. Each predictor reads the logged resource usage data and generates a parameterized model of demand that is stored in memory. When subsequent operations are performed, Spectra updates the in-memory model in addition to logging resource usage.

Each model predicts resource demand as a function of application fidelity and operation input parameters. Fidelities and input parameters may be either discrete or continuous. The default predictor uses binning to model discrete variables: it maintains a separate prediction for each possible discrete value. The default predictor also maintains a generic prediction that is independent of any discrete variable—this prediction is used whenever a specific combination of discrete variables has not yet been encountered. The default predictor uses linear regression to model continuous variables. It adjusts for changes in application behavior over time by giving more recent samples a greater weight in its predictions.

For some applications, resource usage depends heavily upon the specific data on which an operation is performed. For example, the input document to the Latex document preparation system will significantly affect resource usage: a 100 page document consumes more CPU cycles and battery energy than a 2 page document. Spectra's default predictor anticipates this relationship with data-specific models of resource usage. Applications such as Latex associate each operation with the name of a data object. The default predictor maintains a LRU cache of the most recent data objects. When asked to predict future demand, the predictor uses a data-specific model to predict resource usage if it finds such a model in its cache. Otherwise, it uses the more general, data-independent model.

3.5 Ensuring data consistency

Since Coda relaxes data consistency under poor network conditions to achieve acceptable performance, Spectra must interact with Coda to ensure that remote operations read the same data that they would read if they were executed locally. This is vital for applications such as compilers and Latex that read files commonly modified on clients.

Prior to executing an operation, Spectra predicts which files are likely to be accessed. Spectra provides a default file predictor that builds upon the numerical predictor described in the previous section. The file access predictor maintains a numerical prediction of access likelihood for each file that may be accessed. When updating each file's model, the predictor assigns the value of 1 to a file access, and the value of 0 when a file is not accessed. Each resulting prediction thus represents the likelihood that a given file will be accessed.

Spectra uses the file predictor to estimate the cost of servicing cache misses. It compares the list of files that may be accessed by an operation to the list of cached files. For each uncached file, it estimates the number of bytes of data that must be fetched from file servers by multiplying the file size by the predicted access likelihood. Summing this value over all files yields a prediction for the total number of bytes that must be fetched to perform an operation. Spectra divides this prediction by the rate at which Coda will fetch data from servers to estimate time spent servicing cache misses.

Spectra uses the file predictor to maintain data consistency. Before executing an operation remotely, Spectra ensures that all modifications to files with non-zero access likelihood have been reintegrated to file servers. Spectra also ensures that modifications made during the remote execution of an operation are immediately visible to the client. Since Coda performs file reintegration at volume-level granularity, Spectra triggers the reintegration of all modifications for a volume that includes at least one modified file.

If a large amount of data must be reintegrated in poor network conditions, then data consistency significantly increases remote operation execution time. Spectra estimates the added execution time by multiplying the size of the modifications by the predicted bandwidth available to the file server. If the predicted reintegration costs are too high, Spectra avoids them by executing the operation locally.

3.6 Selecting the best option

When applications call `begin_fidelity_op`, Spectra selects a location and fidelity at which to perform the operation. Spectra first determines which servers could possibly execute part or all of the operation. It then polls the resource monitors to obtain a snapshot of resource availability. Finally, it uses a heuristic solver [12] to search the space of possible servers, execution plans, and fidelities. The solver selects the alternative that maximizes an input utility function. Because it uses heuristic techniques, it is not guaranteed to select the optimal alternative—however, as shown in Section 4, it usually selects a very good option.

Spectra provides a default utility function that has so far proven sufficient for all applications. However, applications may override the default with an application-specific implementation. The default utility function evaluates execution alternatives by their impact on *user metrics*. User metrics measure performance or quality perceptible to the end-user—they are thus distinct from resources, which are not directly observable by the user. Spectra currently considers three metrics: execution time, energy usage, and fidelity.

As the solver searches the space of possible alternatives, it calls the utility function with specific input parameters, fidelities, execution plans, and server choices. The default utility function first predicts a context-independent value for each metric: total execution time, total energy usage, and a vector representing fidelity. It then weights each value by its current importance to the user and returns the product of the weighted values as the utility of the alternative.

The default utility function predicts execution time to be the sum of local and remote CPU time, network transmission time, time to service cache misses, and time to ensure data consistency. This simple model reflects Spectra's current implementation, which does not allow computation and network transmission to overlap.

The utility function uses the models of operation resource usage to predict resource demand. It matches demand to the availability predictions in the resource snapshot. It calculates local and remote CPU time by dividing the predicted cycles needed for execution by the predicted cycles per second available on a machine. Network transmission time is calculated by predicting the number of bytes to be transmitted and dividing by the available bandwidth. The effect of latency is predicted by multiplying the estimated number of RPCs by the round-trip time. The calculations for time to service file cache misses and ensure data consistency are described in the previous section.

The importance of execution time is application-specific. Therefore, Spectra requires each application to provide a function that expresses the desirability of different latency values. Many of our applications use the simple expression, $1/T$, where T is the predicted execution time. This has the nice property that an operation that takes twice as long to execute is only half as desirable to the user.

Energy usage is calculated using the model of operation energy demand. Spectra weights the predicted energy usage by the importance of energy conservation. As described in Section 3.3.3, the importance of energy conservation is represented by a parameter, c, that ranges between 0 and 1. The weighted energy component of utility is calculated as $(1/E)^{kc}$, where E is predicted energy usage and k is a constant (currently 10). Thus, when c is 0, energy does not impact utility at all; when c is 1, energy has a large impact.

Fidelity is a multidimensional metric of application-specific quality. Since fidelity is fixed as an input to the utility function, no prediction is necessary. However, an application must provide a function that specifies the desirability of each fidelity as a numerical value.

3.7 Applications

In order to validate the effectiveness of our system, we have modified three applications to use Spectra: the Janus speech recognizer [21], the Latex document preparation system, and the Pangloss-Lite language translator [5].

3.7.1 Speech recognition

Janus performs speech-to-text translation of spoken phrases. It may use one of the three execution plans described in Section 3.1. 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. For execution time, we assign utility to be the inverse of the expected latency.

3.7.2 Document preparation

Latex generates a DVI file from multiple input files. It has only one fidelity, but supports two possible execution plans: local, in which all work is done on the client, and remote, in which all work is done on a server. As with Janus, we assign utility to be the inverse of the expected latency.

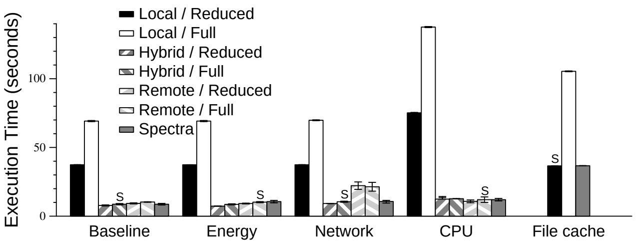


Figure 3. Speech recognition execution time

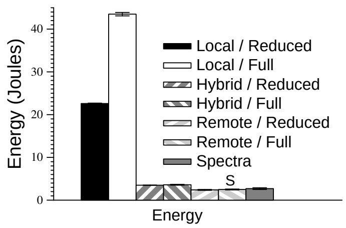


Figure 4. Speech recognition energy usage

We modified Latex to use Spectra by creating a front-end and a Spectra service. The front-end calls Spectra to select an execution location, specifying the name of the top-level input file so that Spectra can parameterize its predictions by document. The service runs Latex as a child process. Data consistency is a significant consideration since input files are often modified on the client and must be reintegrated before processing is done remotely.

3.7.3 Natural language translation

Pangloss-Lite translates text from Spanish to English. 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.

We assign the EBMT engine a fidelity of 0.5 The glossary and dictionary engines produce subjectively worse translations—we assign them fidelities 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 all engines are used, fidelity is 1.0. If a translation takes longer than 5 seconds, we assign it a utility of 0. Conversely, all translations that take less than 0.5 seconds have a utility of 1. Translations that take time, T, between 0.5 and 5 seconds are assigned utility $(T-0.5)/(5-0.5)$.

4 Validation

Our validation of Spectra measures how well it adapts to changes in resource availability. We executed Janus, Latex, and Pangloss-Lite under a variety of scenarios in which we varied resource availability. For each scenario, we measured application latency and energy usage for each possible combination of fidelity, execution plan, and remote server. We also asked Spectra to choose one of the possible alternatives for application execution. The next three sections discuss the results; Section 4.4 evaluates Spectra's overhead.

4.1 Speech recognition

To evaluate Spectra's support for Janus, we limited execution to two machines. The client, an Itsy v2.2 pocket computer [6], represents the small, mobile devices used for pervasive computing. The Itsy was developed by Compaq's Palo Alto Research Labs and includes a 206 MHz StrongArm-1100 processor and a Smart Battery. An IBM T20 laptop with a 700 MHz Pentium III processor was a possible remote server. Since the Itsy lacks a PCMCIA slot, we connected the two machines with a serial link.

We first recognized 15 phrases so that Spectra could learn the application's resource requirements. We then measured how well Spectra performed when recognizing a new phrase in five different resource scenarios.

Figure 3 shows measured execution time for each scenario. The first data set shows results for the baseline scenario, in which both computers are unloaded and connected to wall power. In all figures in this section, each bar shows the mean of five trials—the error bars are 90% confidence intervals. The first six bars show execution time for the alternatives available to Spectra. The local execution plan is clearly inferior to the hybrid and remote plans, taking 3–9 times as long to execute. The large disparity is caused by Janus using floating-point instructions that are emulated in software on the Itsy's SA-1100 processor. However, using the hybrid plan and performing some computation locally takes less time than using the remote execution plan.

The "S" label in each scenario shows the alternative chosen by Spectra. In the baseline scenario, Spectra correctly chooses the hybrid plan and the full vocabulary. This combination executes faster than all but one alternative; that alternative is only slightly faster but its utility is only half as desirable. The last bar in each data set shows the execution time when Spectra chooses an alternative—comparing this bar to the one indicated with an "S" shows Spectra's overhead. In all scenarios, the overhead is minimal—the difference in height is within the 90% confidence intervals.

Each remaining scenario varies the availability of a single resource. In the energy scenario, the client is battery-powered with an ambitious battery lifetime goal of 10 hours. The second data set in Figure 3 shows latency results, and Figure 4 shows energy use. Since energy is critical, Spectra chooses the remote execution plan and the full vocabulary. Although hybrid execution takes less time, it consumes more energy because a portion of the computation is done on the client. Spectra correctly chooses to avoid the

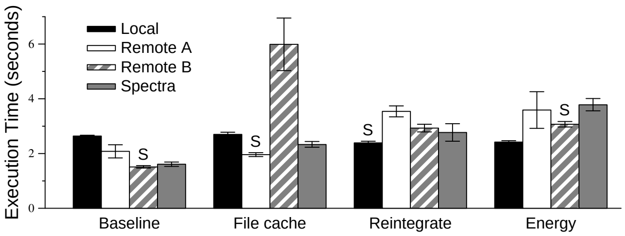


Figure 5. Small document execution time

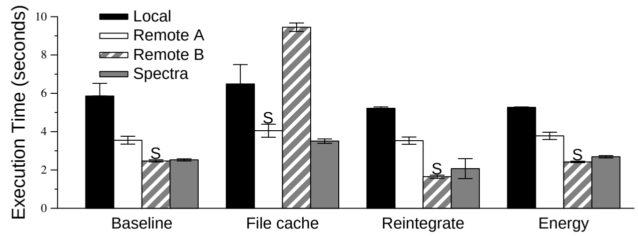
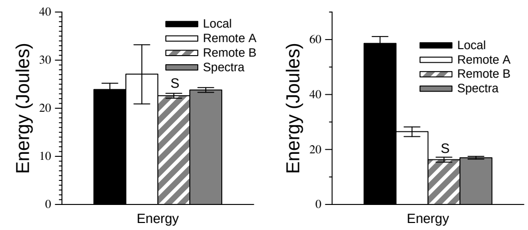


Figure 6. Large document execution time



(a) Small document (b) Large document

Figure 7. Latex energy usage

reduced vocabulary—the small energy and latency benefits do not outweigh the decrease in fidelity.

The network scenario halves the bandwidth between the client and server. This makes remote execution undesirable, and Spectra correctly chooses to use the hybrid plan and full vocabulary. The CPU scenario loads the client processor by executing a CPU-intensive background job. The cost of local computation increases, making the remote execution plan more attractive than the hybrid plan.

The file cache scenario simulates a network partition in which the Spectra server is unavailable and the file servers remain accessible. Prior to execution, the 277 KB language model for the full vocabulary is flushed from the client's cache. This does not affect reduced-quality speech recognition. However, the execution time of full-quality recognition increases significantly because the language model must be refetched from a file server. Spectra anticipates the cache miss and chooses to use reduced-quality recognition. Since full-quality recognition would be approximately 3 times slower, the decrease in fidelity is acceptable.

4.2 Document preparation

We next evaluated Spectra's support for Latex. Latex's resource needs differ from those of Janus; it performs less computation but accesses more files. Since input files may be modified on the client, data consistency is important.

We executed the Latex front-end on an IBM 560X laptop with a 233 MHz Pentium processor. Since the 560X has no energy management support, we used a digital multimeter to measure energy. We ran Spectra servers on the laptop and two remote servers; server A had a 400 MHz Pentium II processor and server B had a 933 MHz Pentium III processor. The network link was a shared 2 Mb/s wireless network.

We used two documents for evaluation: the smaller was 14 pages in length and the larger was 123 pages. We first executed Latex 20 times to allow Spectra to learn application resource requirements. Figures 5 and 6 show execution time for both documents. The first data set in each shows the baseline scenario, in which all computers are unloaded and connected to wall power, and data files are cached on every machine. Since little network communication is needed, CPU speed is the primary consideration. Spectra correctly chooses to use the faster server B for both documents.

In the file cache scenario, server B does not have any input files cached. Spectra correctly anticipates that file access time will increase the time needed to execute Latex on server B and switches execution to server A.

In the reintegrate scenario, a 70 KB input file for the smaller document is modified on the client. Before executing Latex on a remote server, Spectra must ensure that the modified file is reintegrated to the file servers. Reintegration over the wireless network significantly increases execution time for remote execution. However, the speed of local execution is unaffected. Spectra therefore chooses local execution for the smaller document. For the larger document, Spectra correctly predicts that the modified file will not be needed and does not force reintegration. It chooses the fastest plan: execution on server B.

The energy scenario is identical to the reintegrate scenario, except that the client is battery powered and a very aggressive goal for battery lifetime is specified. For the smaller document, Spectra chooses to use server B, even though this takes more time to execute. Figure 7(a) shows the reason: server B uses slightly less energy than other options. Because energy is of paramount concern, Spectra opts for energy savings over faster execution time. The choice for the larger document is much clearer, since execution on server B saves both time and energy.

4.3 Natural language translation

We evaluated Pangloss-Lite using an experimental setup identical to that used for Latex. We first translated a set of 129 sentences. We then asked Spectra to choose the best option for translating five additional sentences.

Pangloss-Lite has many options for execution; there are

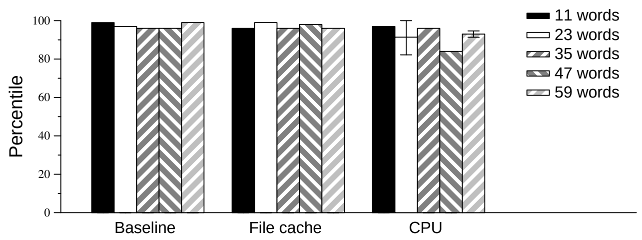


Figure 8. Accuracy for Pangloss-Lite

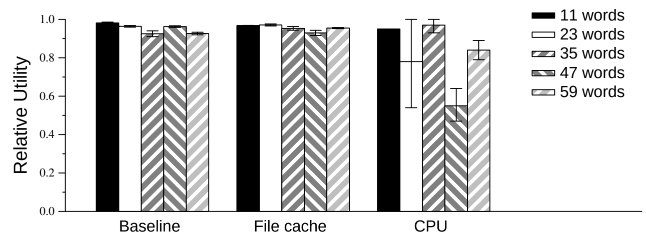


Figure 9. Relative utility for Pangloss-Lite

100 different combinations of location and fidelity. Due to the large number of options, we present results in a different format. We rank the alternatives by the utility they achieved. Each bar in Figure 8 shows the percentile into which Spectra's chosen alternative falls; a value of 99 indicates that Spectra has made the best choice. Each bar in Figure 9 compares the utility achieved by Spectra with the utility that would be achieved by an oracle with no overhead.

In the baseline scenario, all computers are unloaded and wall-powered, and data files are cached on all machines. For the three smallest sentences, Spectra uses all engines; for the two larger sentences, it does not use the glossary engine. These choices are best for all sentences. This scenario shows the importance of modeling input parameters—Spectra correctly predicts that execution time will increase with sentence size and switches to a lower fidelity to achieve acceptable performance for larger sentences.

Spectra runs the dictionary engine locally for the four smallest sentences and executes all other components on server B. Remote execution yields significant performance improvements for the glossary and EBMT engines because they have large CPU requirements. The location of the dictionary engine and the language modeler does not affect performance much because their processing requirements are small. Thus, although Spectra's location choice for these two components is not optimal for all sentences, the performance penalty is small (less than 0.07 seconds).

This illustrates an important property: when alternatives significantly differ in utility, Spectra almost always makes a correct decision. If it does choose an inferior option, that option's utility is usually close to that of the best option. For the baseline scenario, the utility of Spectra's choices are all within 2% of the best option. Even adding in the overhead of picking the correct alternative, the utility of Spectra's choices are all within 7% of the best choice.

Figures 8 and 9 show results from two additional scenarios. In the file cache scenario, we evicted a 12 MB file needed by the EBMT engine from server B's cache. The CPU scenario is identical to the file cache scenario, except that we execute two CPU-intensive processes on server A. In general, Spectra did an excellent job for Pangloss-Lite, achieving on average 91% of the best utility. However, Spectra is limited by its execution model which currently supports only sequential execution. We plan to explore execution plans that support parallel execution. For Pangloss-Lite, this would yield considerable benefit: the three engines could be executed in parallel on different servers.

Activity	Duration (ms.)		
	No Servers	1 Server	5 Servers
register_fidelity	1.2	1.5	1.2
begin_fidelity_op	8.3	13.1	65.5
file cache prediction*	5.2	8.5	8.5
choosing alternative	0.4	1.0	43.4
other activity	2.7	3.6	13.6
do_local_op	5.9	4.7	5.1
operation execution	4.9	4.0	4.0
other activity	1.0	0.7	1.1
end_fidelity_op	2.1	2.1	2.2
total	18.4	21.4	74.0

Figure 10. Spectra overhead

4.4 Overhead

We measured Spectra's overhead by performing a null operation that returns immediately after being invoked. Figure 10 shows that with no remote servers available, the null operation takes 18 ms. to execute. File cache prediction takes 5.2 ms. with a relatively empty cache; however, it can take as long as 359.6 ms. when the cache is full. This excessive overhead is due to an inefficient interface in which Coda writes the entire cache state to a temporary file. We plan to replace this interface with a more efficient implementation. Overhead increases with the number of potential servers, primarily due to additional time spent choosing the best alternative. With 5 servers, overhead is only 74 ms., which is very reasonable for our targeted applications that perform operations of a second or more in duration.

5 Related work

Remote execution is a well-established field in systems research. While most remote execution systems target only performance benefits, a few recent systems have explored how remote execution can reduce application energy use. Rudenko et al. [15] compare the energy cost of executing several tasks both locally and remotely. Their RPF framework [16] adaptively decides where a task should be executed based upon a history of past power consumption. Kunz's toolkit [11] uses similar considerations to locate mobile code. Although both systems monitor execution time and RPF also monitors battery use, neither monitors individual resources such as network and cache state, limiting their ability to cope with resource variation. In addition, neither exploits the full potential of the energy-performance tradeoff—they use remote execution only when both energy usage and performance are not adversely affected.

Kremer et al. [10] propose using the compiler to select tasks that might be executed remotely to save energy. At present, this analysis is static, and thus cannot adapt to changing resource conditions. Such compiler techniques are complementary to Spectra—they could automatically select operations and insert Spectra calls into executables.

Vahdat et al. [19] note issues considered in Spectra's design: the need for application-specific knowledge and the difficulty of monitoring remote resources. Butler [13] uses AFS for consistency between local and remote machines. Our use of Coda is similar, but reflects a different target environment: Coda's support for disconnected and weakly-connected operation is vital in pervasive computing.

Several systems analyze application behavior to locate functionality. Coign [7] statically partitions objects in a distributed system by logging and predicting communication and execution costs. Abacus [2] monitors network and CPU usage to migrate functionality in a storage-area network, and Condor monitors goodput [3] to migrate processes in a computing cluster. Because these systems are not designed for pervasive computing, they do not monitor the range of resources considered by Spectra. In addition, they do not support adaptive applications that modify their fidelity.

6 Conclusion

Remote execution is an important capability in pervasive computing because it combines the mobility of small devices with the greater processing power of large compute servers. Spectra helps applications realize the benefit of remote execution by matching resource supply and demand in order to advise applications how and where they should execute functionality. Our evaluation shows that Spectra usually chooses the best alternative for execution despite wide variation in resource availability. When Spectra does not make the best decision, its choice is usually very good. Given these encouraging results, we believe that Spectra will prove to be a valuable platform for future research.

Acknowledgments

We wish to thank Ralf Brown and Rob Frederking for assistance with Pangloss-Lite, and Jie Yang for help with Janus. We thank Jan Harkes and Shafeeq Sinnamohideen for their invaluable knowledge of the Coda file system. Lastly, we thank Dushyanth Narayanan for numerous useful discussions throughout this project.

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, IBM, Nokia, Intel, Hewlett-Packard, and Compaq. 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, IBM, Nokia, Intel, HP, Compaq, nor the U.S. government.

References

[1] Adjie-Winoto, W., Schartz, E., Balakrishnan, H., and Lilley, J. The design and implementation of an intentional naming system. In *Proc. of the 17th ACM Symp. on Op. Syst. Princ.*, pages 202–16, Kiawah Island, SC, Dec. 1999.

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

[3] Basney, J. and Livny, M. Improving goodput by co-scheduling CPU and network capacity. *Intl. Journal of High Perf. Comp. App.*, 13(3), Fall 1999.

[4] Flinn, J. and Satyanarayanan, M. Energy-aware adaptation for mobile applications. In *Proc. 17th Symp. on Op. Syst. Princ.*, Kiawah Is., SC, Dec. 1999.

[5] Frederking, R. and Brown, R. D. The Pangloss-Lite machine translation system. In *Proc. of the 2nd Conf. of the Assoc. for Mach. Trans. in the Americas*, pages 268–272, Montreal, Canada, 1996.

[6] Hamburgen, W. R., Wallach, D. A., Viredaz, M. A., Brakmo, L. S., Waldspurger, C. A., Bartlett, J. F., Mann, T., and Farkas, K. I. Itsy: Stretching the Bounds of Mobile Computing. *IEEE Computer*, 13(3):28–35, Apr. 2001.

[7] Hunt, G. and Scott, M. The Coign automatic distributed partitioning system. In *Proc. 3rd Symp. Op. Syst. Design and Imp.*, New Orleans, LA, Feb. 1999.

[8] Intel, Microsoft, and Toshiba. *Advanced Configuration and Power Interface Specification*, February 1998. http://www.teleport.com/˜acpi/.

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

[10] Kremer, U., Hicks, J., and Rehg, J. M. Compiler-directed remote task execution for power management. In *Proc. of the Workshop on Compilers and Operating Systems for Low Power*, Philadelphia, PA, Oct. 2000.

[11] Kunz, T. and Omar, S. A mobile code toolkit for adaptive mobile applications. In *Proc. of the 3rd IEEE Workshop on Mobile Comp. Syst. and App.*, pages 51–59, Monterey, CA, Dec. 2000.

[12] Narayanan, D., Flinn, J., and Satyanarayanan, M. Using history to improve mobile application adaptation. In *Proc. of the 2nd IEEE Workshop on Mobile Comp. Syst. and App.*, pages 30–41, Monterey, CA, Dec. 2000.

[13] Nichols, D. Using idle workstations in a shared computing environment. In *Proc. of the 11th ACM Symp. on Op. Syst. Princ.*, Austin, TX, Nov. 1987.

[14] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. Agile application-aware adaptation for mobility. In *Proc. of the 16th ACM Symp. on Op. Syst. Princ.*, Saint-Malo, France, Oct. 1997.

[15] Rudenko, A., Reiher, P., Popek, G. J., and Kuenning, G. H. Saving portable computer battery power through remote process execution. *Mobile Computing and Communications Review*, 2(1):19–26, Jan. 1998.

[16] Rudenko, A., Reiher, P., Popek, G. J., and Kuenning, G. H. The Remote Processing Framework for portable computer power saving. In *Proc. of the ACM Symposium on Applied Computing*, San Antonio, TX, Feb. 1999.

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

[18] SBS Implementers Forum, http://www.sbs-forum.org/. *Smart Battery Data Specification, Revision 1.1*, Dec. 1998.

[19] Vahdat, A., Lebeck, A. R., and Ellis, C. S. Every Joule is precious: A case for revisiting operating system design for energy efficiency. In *Proc. of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, Sept. 2000.

[20] Viezades, J., Guttman, E., Perkins, C., and Kaplan, S. *Service Location Protocol*. IETF RFC 2165, June 1997.

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