Automatic Configuration and Run-time Adaptation of Distributed Applications

Fangzhe Chang and Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University, New York, NY 10012
{*fangzhe,vijayk*}@*cs.nyu.edu*

Abstract

Current technology trends point towards both an increased heterogeneity in hardware platforms and an increase in the mechanisms available to applications for controlling how these platforms are utilized. These trends motivate the design of *resource-aware* distributed applications, which proactively monitor and control utilization of the underlying platform, ensuring a desired performance level by adapting their behavior to changing resource characteristics.

This paper describes a general framework for enabling application adaptation on distributed platforms. The framework combines programmer specification of alternate execution behaviors (configurations) with automatic support for deciding *when* and *how* to adapt, relying extensively on two components: (1) *profile-based modeling* of application behavior, automatically generated by measuring application performance in a virtual execution environment with controllable resource consumption, and (2) *application-specific continuous monitoring* of current resource characteristics. The latter detects when application configurations need to change while the former guides the selection of a new configuration.

We evaluate these framework components using an interactive image visualization application. Our results demonstrate that starting from a natural specification of alternate application behaviors and an automatically generated performance database, our framework permits the application to both configure itself in diverse distributed environments and adapt itself to run-time changes in resource characteristics so as to satisfy user preferences of output quality.

1 Introduction

Current day general-purpose applications can execute on a wide range of platforms ranging from fast desktop computers to mobile laptops all the way to hand-held PDAs, spanning several orders of magnitude in processing, storage, and communication capabilities. At the same time, operating systems are providing increased control over system utilization through mechanisms such as fair-share CPU scheduling [2, 11, 13, 16]. Similarly, QoS-aware network protocols [4, 20] permit control over bandwidth, delay, and loss rate. The first trend implies that the performance of a distributed application running on diverse heterogeneous platforms varies dramatically, both because of differing capabilities of these platforms and because of competition for resources affecting their dynamic availability. Fortunately, the second trend suggests a way for providing predictable performance despite these variations. *Resource-aware* applications, which proactively monitor and control utilization of the underlying platform, can ensure a desired performance level by adapting themselves to changing resource characteristics. For instance, a distributed application conveying a video stream from a server to a client machine can respond to network bandwidth reduction by compressing the stream or selectively dropping frames. Similarly, it can respond to increased server load by trading off server involvement for increased network bandwidth and client computation.

However, despite a thorough understanding of the need for adaptation at the level of individual components (e.g., the network-congestion induced dynamic resizing of the acknowledgment window in the TCP protocol), little support is available for structuring such globally adaptive applications. Several researchers have begun to address this shortcoming [10, 14, 15, 17, 18]; however, most such efforts place an unreasonable burden on application developers requiring them to provide explicit specification of both *resource-utilization profiles* (which resources are used at which time and in what quantity), and *adaptation behaviors* (how should the application react to changes in resource allocation levels).

This paper describes a general framework for enabling application adaptation on distributed platforms, which reduces the burden on the application developer by automating as much of the adaptation process as feasible. The developer is responsible only for specifying alternate application algorithms and configurations (broadly referred to as *application tunability*). The execution system automatically determines both *when* adaptation should be performed and *how* the application must be modified (i.e., which of its configurations must be chosen). Central to this framework, and the main focus of this paper, are two components: (1) **profile-based modeling** of application behavior, automatically generated by measuring application performance in a virtual execution environment that enables control over resource consumption, and (2) **application-specific continuous monitoring** of current resource characteristics. Run-time adaptation is triggered whenever the second component detects that the currently active application configuration no longer meets user preferences of application quality, and is guided by the first component.

Thus, our framework articulates a different application structuring and execution strategy. The passive form of the application consists of multiple configurations and an accompanying (automatically generated) performance database of their behavior under different resource conditions. Application execution is achieved by selecting one among multiple configurations, with the active configuration dynamically chosen based upon a correlation of observed resource characteristics with the performance database.

This strategy has been implemented in the context of distributed applications running on Windows NT platforms. Application tunability is specified using language-level annotations. The performance database is constructed by sampling the behavior of each configuration in a controlled testbed environment that leverages OS neutral mechanisms to limit CPU, network, and memory consumption of general Win32 processes. The run-time environment consists of a monitoring module that employs similar mechanisms to detect available resource characteristics, a scheduling module that uses the database to select the next active configuration, and a steering module that enforces the change. Experiments with a distributed interactive image visualization application demonstrate that starting from a natural specification of alternate application behaviors, it is possible to automatically generate the performance database. Moreover, using other components of the framework, the application can automatically configure itself in diverse distributed environments as well as adapt itself to run-time changes in resource characteristics so as to satisfy user preferences of output quality.

The rest of this paper is organized as follows. Section 2 introduces the notion of tunability and describes the interactive visualization application used in the remainder of the paper. Section 3 overviews the overall adaptation framework with Sections 4, 5, and 6 providing details about specifying alternate configurations, modeling application behavior using the virtual testbed, and controlling application execution at run time, respectively. Section 7 describes the use of these components for dynamic adaptation of the visualization application. Related work is discussed in Section 8 and we conclude in Section 9.

2 Application Tunability

Application tunability, a notion we initially introduced in [9, 8], refers to an application's ability to trade off resource requirements over several dimensions, including time, quality, and resource type, while still producing an output of adequate quality. Tunable applications are able to compensate for a lower allocation in one stage of the computation by (1) requiring a higher allocation in another stage, or (2) lowering output quality, or (3) raising demand for resources of another type.

Application tunability is a characteristic of several parallel and distributed computations. The key attribute is the existence of *alternate application configurations*, each with a different execution path and resource profile. These multiple resource profiles provide run-time flexibility, permitting choice of an alternate configuration better suited to matching user preferences given available system resources.

2.1 Active Visualization: A Tunable Distributed Application

The active visualization application [6] is a general-purpose client-server application for interactively viewing, at the client side, large images stored in the server. Active visualization exploits several multi-resolution and progressive transmission techniques to reduce client latency. First, images are stored at the server as wavelet coefficients [5], enabling the construction of images at different levels of resolution (including the original one). Second, it uses progressive transmission, making it unnecessary to fetch the entire image at a single time. Based upon an initial specification of the highest resolution required by the client, the server constructs a pyramid of images ranging from the finest to the coarsest resolution. The server uses this pyramid to transmit an area of the image that corresponds to the user's fovea (the user's focus of interest as identified by location of the mouse cursor), starting from the coarsest resolution and progressing up to the user-preferred resolution. If the user's fovea does not change, the client requests the server to send it an incremental region surrounding the fovea, ensuring eventual transmission of the entire image at the highest resolution. To further reduce client latency, the application can optionally compress the data before injecting it into the network, reducing network bandwidth at the expense of requiring decompression at the client.

Active visualization is a tunable application in that its behavior is affected by values associated with parameters such as preferred resolution level and size of the foveal region. In addition to reflecting different user quality preferences, these parameters influence the resource utilization profile of the application. A lower preferred resolution value implies a smaller amount of data processing and transmission time, and consequently a lower utilization of processing and network resources. The foveal region size parameter affects responsiveness and total image transmission time. The bigger the value of this parameter, the longer the round-trip response time, but the shorter it takes to receive the entire image. In addition, the value of the optional compression parameter, which controls choice of the compression algorithm, impacts resource utilization at the client and server nodes as well as of the network connection between them.

3 Application Structuring and Adaptation Framework

Figure 1 shows the general framework for enabling application adaptation on distributed platforms. The framework consists of several integrated components that support three functions: (1) *specification of application configurations*, (2) *modeling of application behavior*, and (3) *run-time application adaptation*. A detailed description of the components is deferred to Sections 4–6; here, we focus on providing an understanding of how these components work together to enable adaptation.

Specifying application configurations. Alternate execution behaviors are specified using language **annotations** which identify the control parameters that determine alternate paths, execution environments where these paths execute, transitions between paths, and application-specific quality metrics. This annotated program is converted by a **preprocessor** into an executable form of the application; the latter includes application modules that make up the different paths as well as steering and monitoring agents that are used for run-time adaptation. In addition, the preprocessor generates performance database templates, used during the modeling of application behavior.

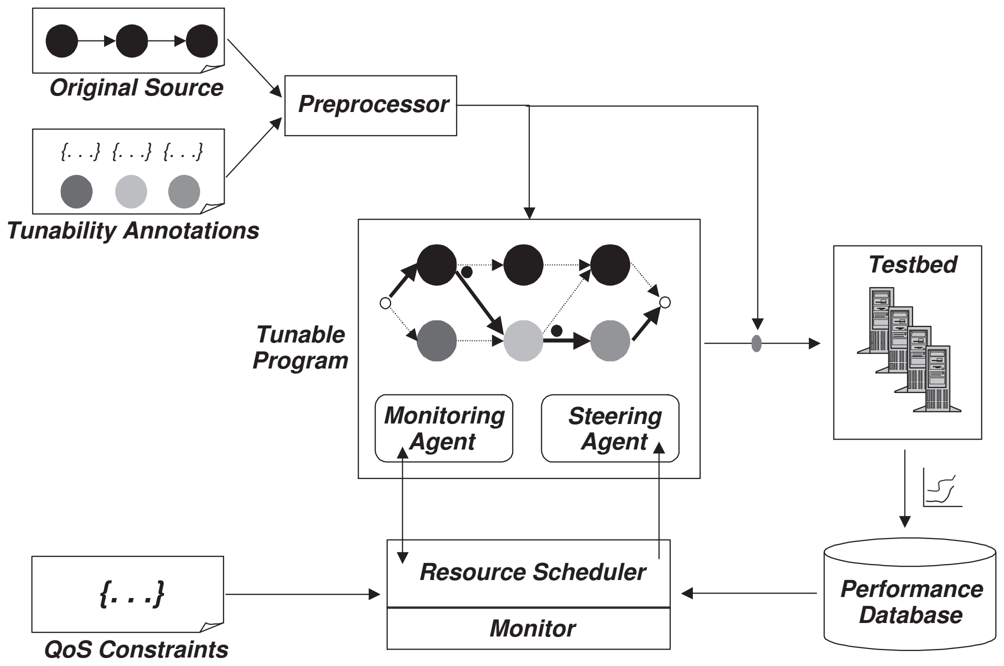


Figure 1: Generation and Execution of Tunable Applications.

Modeling application behavior. Run-time adaptation is guided by models of the behavior of different application configurations under a variety of resource conditions. These models are automatically generated by executing each configuration in a virtual **testbed** environment that allows control over the amount of resources (CPU, memory, network) available to the application. The testbed enables the execution behavior of each configuration to be sampled at different points in a multidimensional resource space without actually requiring the application to be executed in each of these different environments. The outcome of these controlled executions is a performance model that maps, for each configuration, the application input and resource availability to application output quality. These models are stored in a **performance database** used during run-time adaptation.

Run-time Application Adaptation. At run time, an appropriate application configuration is chosen to satisfy user preference constraints (i.e., **QoS constraints**) given available resource characteristics. The **monitoring agent**, combined with the system-wide monitor, keeps track of the actual portion of system resources available to application processes. Upon detecting that user preferences of application quality can no longer be satisfied using the currently active configuration, the monitoring agent invokes the **resource scheduler** which decides on an alternate configuration by correlating available resource characteristics with the models of configuration behavior stored in the performance database. Actual adaptation is realized by a control message sent from the resource scheduler to the **steering agent**, causing the latter to switch application configurations by appropriately updating the control parameters.

4 Specifying Application Configurations

Although several applications allow flexible execution, this flexibility is often hidden deep inside the specific expression of application functionality. For automatic adaptation, we need to identify the control parameters ("knobs") that determine execution behavior and explicitly specify different execution paths. To facilitate this, our approach relies upon language-level annotations to the original application source code, which specify the execution environment, control parameters, application qualities and their measurement mechanism, tunable application modules, and the special points where configuration and reconfiguration can take effect. A detailed description of the syntax and semantics of these annotations appears elsewhere [9, 8]; here, we illustrate their usage using the active visualization application.

4.1 Tunability in Active Visualization

Figure 2 shows the specification of alternate configurations in the active visualization application. Although the application has both a server-side and a client-side component, in the rest of this paper, we restrict our attention to the client-side component, treating the server-side component as a black-box whose behavior is entirely determined by the control messages sent to it from the client.

The top part of Figure 2 shows a simplified version of the original client-side code, while the bottom part shows this code annotated with tunability specifications. The client first establishes a connection with the server and notifies the latter regarding choice of compression method. The client then retrieves the image in steps. In each step, it requests a square area denoted by the foveal center (x, y) and size r up to an image resolution level l. The client decompresses the data upon reception, and then updates the display. It then checks for any user interaction, which has the effect of changing the location and size of the foveal region.

The tunability specification includes five kinds of annotations:

1. *Control parameters:* The bottom part of Figure 2 identifies `dR`, `c`, and `l` as parameters, which determine the execution path taken by the application and the behavior along this path.
2. *Execution environment:* The execution environment specifies the system components (hosts and network links) on which the application executes. Each system component encapsulates several resources that affect application behavior. For instance, a host is characterized by its CPU, memory, and network resources, while a network link is characterized by its bandwidth. In our example, the execution environment consists of two hosts, the client and the server. The network link between them is not explicitly included in the execution environment because in this application, link resource constraints can be captured in terms of constraints on host network resources.
3. *Quality metrics:* The `QoS_metric` construct specifies the application output metrics of interest. In our example, these metrics include the total data transmission time (`transmit_time`), average response time (`response_time`, and image resolution level (`resolution`). We require that different values of the same quality metric can be compared with each other. As shown in Figure 2, updates to these metrics are handled by code segments contained within the `QoS_monitor` construct.
4. *Tunable modules:* The abstract model of a tunable application is that of a family of DAGs built up from individual modules. Each module is specified by the `task` construct, which specifies the control parameters affecting module execution, the environment components and resource types utilized by the module, and application-specific quality metrics that denote task outputs. Application execution paths are specified by associating guard expressions of control parameters (not shown in the Figure) with each task and specifying inter-task control flow. In this example, the entire data transmission task is treated as a single module named `module{l}{dR}{c}`. The control parameters in the task name are evaluated as name-value pairs when the task construct is instantiated at run time, and serve as a handle for referring to a specific task configuration.
5. *Configuration Transitions:* When resource availability changes, the system reconfigures the application. The new configuration (indicated by `new_control`) takes effect when a `transition` construct or `task` construct is encountered. The `transition` construct contains code for application-specific actions. As shown in Figure 2, reconfiguration may require setting some local variable or

Original program :

```
establish_connection();
notify_server_compression_type(c);
while (r < image_size(l)) {
    r += dR;
    send_request(x, y, r, l);
    recv_reply(&data);
    decompress(c, &data);
    update_display(x, y, r, l, data);
    check_for_user_interaction(&x, &y, &r, &dR);
}
...
close_connection();
```

Tunable program :

```
control_parameters {
    int dR;                  // incremental fovea size
    int c;                   // compression type
    int l;                   // level of image resolution
} control;
execution_env {
    host client;             // local host
    host server;
} env;
QoS_metric {
    int transmit_time;       // total image transmission time
    int response_time;       // maximum response time of a single round
    int resolution;          // the resolution of the image
} QoS;
 establish_connection();
 notify_server_compression_type(control.c);
task module{l}{dR}{c} [l, dR, c] [client.CPU, client.network]
       [QoS.transmit_time, QoS.response_time, QoS.resolution]
        while (r < image_size(control.l)) {
        QoS_monitor {t0 = clock();}
        r += control.dR;
        send_request(x, y, r, control.l);
        recv_reply(&data);
        decompress(control.c, &data);
        update_display(x, y, r, control.l, data);
        check_for_user_interaction(&x, &y, &r, &control.dR);
        QoS_monitor {
            t1 = clock();
            QoS.response_time = avg(QoS.response_time, t1-t0);
            QoS.transmit_time += (t1-t0);
            QoS.resolution = control.l;
        }
        transition (new_control) {
            if (new_control.c != control.c) notify(env.server, new_control.c);
        }
    }
taskend

 ...
 close_connection();
```

Figure 2: Expression of application tunability in Active Visualization.

notifying a remote host. Similar to tasks, the transition construct can also have associated guard expressions; these determine whether or not transitions from/to a specific task configuration are possible.

5 Modeling Application Behavior

An application usually exhibits different behaviors, most noticeably in terms of execution duration, when its input or the resource conditions in its execution environment change. Explicit specifications of application tunability enable us to develop a model of behavior for each of the application configurations and at run-time choose the appropriate configuration for desired behavior. We model the behavior of a tunable application as the mapping from control parameters (application input can be viewed as an additional control parameter) and resource conditions to application-specific quality metrics, where each of them constitutes a multi-dimensional domain. In practice and for general-purpose applications, the interaction of these factors is so complex that it is difficult to obtain an analytical expression capturing their relationship. Consequently, we use profile-based modeling to approximate this mapping: for each application configuration, we measure the achieved quality metrics for a sampling of different resource conditions, and interpolate these measurements to get performance curves that summarize configuration performance.

As described in Section 4, language annotations specify application-specific control parameters, the underlying execution environment, and quality metrics. Starting from these annotations, the preprocessor generates configuration files that allows dynamic selection of individual configurations. A driver program executes each configuration repeatedly in a virtual execution environment for different levels of allocated resources. As mentioned earlier, this has the effect of sampling configuration behavior at different points in a multidimensional resource domain. For each such run, application-specific output quality metrics are measured and stored in a database. A separate tool analyzes this performance data, performs sensitivity analysis to determine configurations and regions of the resource space that require additional samples. The output of the modeling step is a performance database that records information about a maximal subset of the configurations representing the resource profile of this application.[1] Interpolation of these data gives reasonal predication of application performance under different run-time conditions.

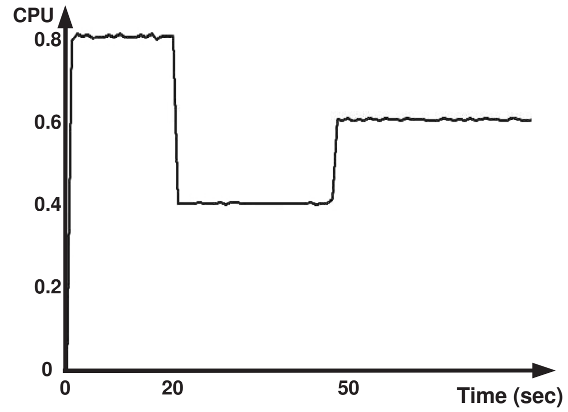
In the rest of this section, we first describe the realization of the virtual "testbed" execution environment that allows profile-based modeling of application behavior, and then present an example of such modeling in the context of the active visualization application.

5.1 Virtual Execution Environment

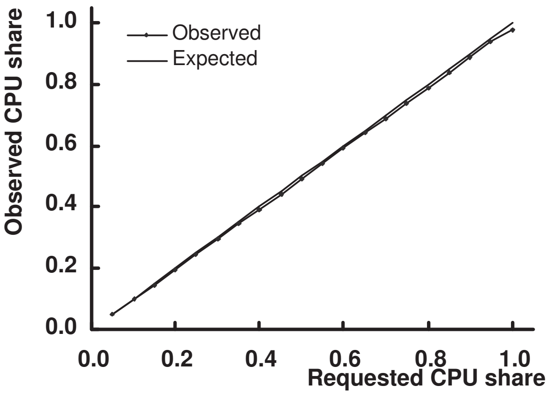
Our virtual execution environment effectively creates a sandbox [7] around an application, which constrains application utilization (in terms of capacity) of system resources such as the CPU, memory, disk, and network. For the kinds of applications that we are interested in, it is sufficient for the virtual execution environment to ensure that average utilization of a resource over a time period is below specified limits.

The sandbox is realized at the user level for general Win32 applications running on the Windows NT operating system using a combination of novel techniques. The basic idea is to inject some functionality into the application at run-time, using a technique known as API interception [1, 12]. This code continually monitors application requests for operating system resources and estimates a "progress" metric (e.g., what fraction of the CPU share has the application been receiving). Upon detecting that the application has

[1]These can informally be defined as configurations that outperform other configurations under at least one resource situation. Additionally, configurations that exhibit similar execution behavior can be merged (with only one of them being stored) in the performance database.



(a)



(b)

Figure 3: (a) Dynamic control of CPU share, (b) Application execution with controlled CPU share.

received either more or less than its share, the injected code actively *controls* future application execution, relying on generally available OS mechanisms, to ensure compliance with specified limits.

The sandbox has been used to allot a specific share of the CPU, limit the amount of physical memory usage, and adjust the network bandwidth on request. In particular, for these resources, application control involves dynamically manipulating application priority (every few milliseconds) to ensure a proper CPU share, switching protection bits of mapped pages to ensure physical memory usage limits, and delaying sending and receiving of messages to ensure that the application sees the desired bandwidth in its network interactions. These controls incur very low overhead and interference with other system applications; in fact, we can run several virtual machines on the same physical host, without them interfering with each other. When the system is under-loaded, the virtual execution environment guarantees that the application will get the exact amount of resources as requested. Therefore, it can serve as a testbed for examining the application behavior as if the latter runs in an environment with the specified resource characteristics.

A detailed description of the techniques used for constructing the virtual execution environment and its capabilities is reported elsewhere [7]. Here we are interested in understanding whether the virtual execution environment can serve as a useful tool for modeling application performance in a (different) physical environment. To do this, we measure the behavior of applications on the controlled testbed, comparing it with that on a variety of physical machines. The experiments were conducted on four PCs: two Pentium II (450 MHz), one Pentium II (333 MHz), and one Pentium Pro (200 MHz), all with 128 MB memory and running Windows NT 4.0 (service pack 5). They are connected by 100Mbps Ethernet, with 10/100 MB auto-sensing network cards from different manufactures. The testbed runs on a Pentium II (450 MHz) machine to emulate less powerful machines. Although our experiments involved multiple resource types and their combinations, we only describe the results for one kind of resource type, the CPU.

Figure 3(a) shows a snapshot (with added axes) of CPU usage of a simple toy application from the NT Performance Monitor. At startup, the testbed is configured to provide the application with 80% of the CPU share, which changes to 40% after 20 seconds, and to 60% CPU after an additional 30 seconds. This figure shows that the application does get the CPU share as specified and that its resource usage is stable over time, thus demonstrating the testbed's ability to create an execution environment where dynamic resource availability is controlled as desired.

To understand how application behavior under the testbed differs from that on a slower physical machine, we measure the execution time of this simple application on a Pentium II (450 MHz) machine and compare it with the execution times measured for the application when running under the testbed configured for various CPU shares ranging from 10% to 100%. Figure 3(b) plots the measured times (on the testbed) and the expected times (the execution time measured on the physical machine, normalized with the requested share). As the figure shows, the application's execution time under the testbed is very close to what we expect, with negligible differences except for the case when the request is for 100% CPU share.[2]

To see if this result also holds across different physical machines, we execute the application on a Pentium II (333 MHz) and a Pentium Pro (200 MHz), comparing the measured times against those measured under the testbed running on a Pentium II (450 MHz). The testbed is configured to provide a CPU share that corresponds to the ratio of processor speeds (such simple modeling of the physical machine is sufficient because the application is a tight loop running out of registers). Figure 4(a) shows that the execution times on the testbed for this simple application are about the same as that on the physical machines.

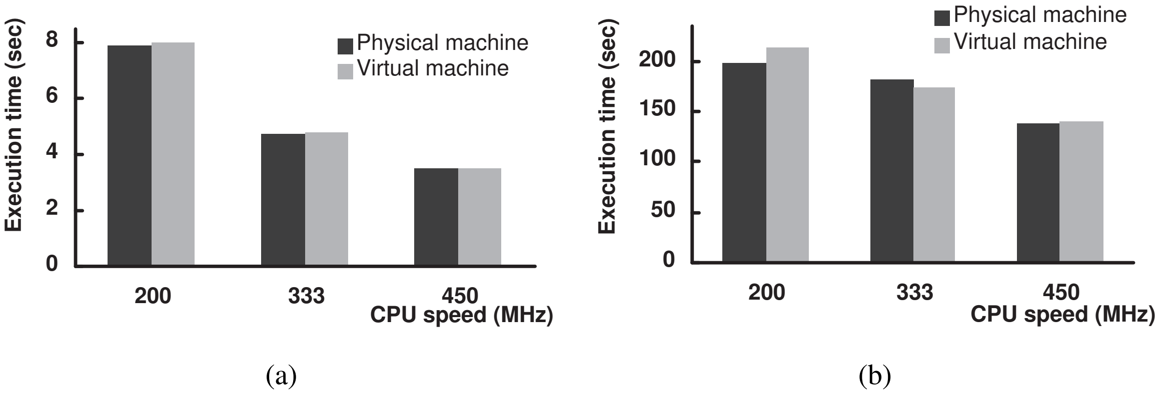


Figure 4: Application execution on testbed and physical machines: (a) a simple application (b) active visualization application

We next examine if the same behavior is true for more general applications that include memory, disk, and network effects. Figure 4(b) shows the measured performance of the active visualization application with the client side running under the testbed on the Pentium II (450 MHz) machine, and compares it against measured performance when the client is running on the other two physical machines (Pentium II (333 MHz) and Pentium Pro (200 MHz)). In each case, the server runs on a Pentium II (450 MHz) machine under a testbed which limits its network bandwidth to 1 MBps. In this case, the CPU shares on the testbed are based on ratios of the Spec Int95 index for these machines. We observe that the execution time on both the physical machine and on the testbed is far less than that obtained by stretching the Pentium II (450 MHz) execution time in proportion to the CPU share. This is expected since application waiting times (e.g., for message reception) are not affected by the node CPU performance; our testbed models such effects as accurately as possible. As the figure shows, the results are very similar for the emulation of the Pentium II (333 MHz) machine. While still good, we observe a bigger difference (up to 8%) for the emulation of the Pentium Pro (200 MHz). Several factors contribute to this: heuristics are used to estimate application progress, the CPU share for which the testbed is configured may not be just right for this application on this machine, and the underlying hardware may exhibit different behaviors (e.g., different network cards). Better emulation can be achieved by factoring these effects into the testbed configuration. However, such accuracy may not be required because for our purposes, it suffices to have the testbed accurately capture relative performance of the different configurations for a particular level of resource availability.

Thus, the testbed provides a good emulation platform for predicting the behavior of an application configuration on a variety of physical machines. We next describe its use for populating the performance

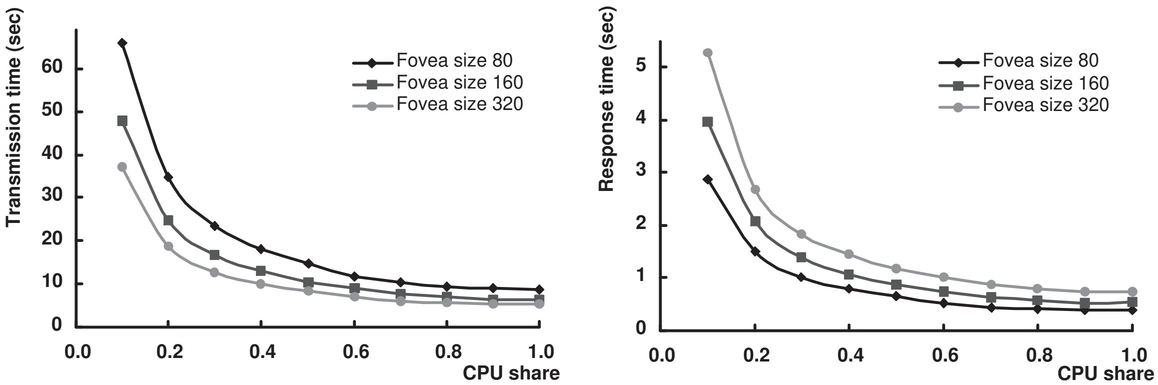
[2]This is because of daemons and other uncontrollable OS activity.

database of the active visualization application.

5.2 Performance Database for Active Visualization

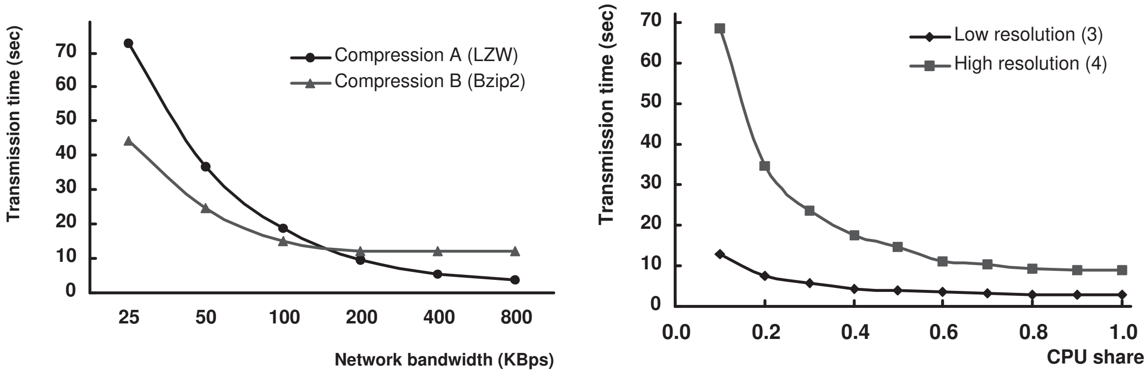
The application performance database stores profiles of application behavior, which are automatically generated by measuring application performance in a virtual execution environment. Each record in the database contains control parameter setting, resouce conditions, and the corresponding application quality. These records are interpolated to obtain performance curves.

As described in Section 2, different configurations of the active visualization application can be executed by appropriately manipulating its control parameters. We use a driver program to repeatedly execute each of the configurations in the testbed, obtaining a mapping from the control parameter values to the output qualities for a wide range of resource conditions. Here, we discuss a small subset of these measurements and the resulting mappings (see Figures 5 and 6).



(a) (b)

Figure 5: Image transmission time (a) and response time (b), for different fovea sizes as CPU share varies.



(a) (b)

Figure 6: Image transmission time, for (a) different compression methods as network bandwidth varies, and for (b) images of different resolutions as CPU share varies.

Figure 5 shows the image transmission time and average response time of user interactions for different fovea sizes as the CPU share varies (this corresponds to running the application on physical machines with different CPU capabilities). In general, an increase in CPU resources reduces both transmission time and response time. Ideally, a user would like these two parameters to be as small as possible. However, they show opposite trends with fovea size. Comparing curves for different fovea sizes, we observe that the larger the fovea size, the smaller the total transmission time, but the larger the response time. This implies that simultaneously satisfying user preferences of transmission time and response time require selecting a different fovea size for different levels of CPU resources.

Figure 6(a) shows image transmission time for different compression methods as the network bandwidth varies (keeping other resources such as CPU at a fixed level). The two curves in the figure correspond to two different compression methods used in the active visualization application: compression B (Bzip2) trades off additional CPU resources to achieve a better compression ratio than compression A (LZW). Although image transmission times decrease with increased network bandwidth for both compression methods, the crossover between the two curves indicates that there exist resource conditions where one compression method should be preferred over another. Compression B outperforms compression A when the network bandwidth is low because less data is transmitted. However it is not good when the network bandwidth is high because the CPU becomes the bottleneck then. In this case, compression A should be used. Therefore, adaptation to varying network conditions and CPU capabilities requires choosing an appropriate compression method.

Figure 6(b) shows the transmission time for images of different resolutions as the CPU resource varies (keeping other resources at a fixed level). In general, additional CPU resources lead to a shorter transmission time and a low level of CPU resources imply a longer transmission time. However, transmission time can be lowered by lowering the image resolution, implying that satisfying user preferences for image transmission time requires choosing an image resolution level that will deliver the required transmission time given available CPU resources.

Section 7 shows how these performance data are used for automatic configuration of this application.

6 Run-time Application Adaptation

Given a tunable application, an accompanying performance database, and user preference constraints, the run-time application adaptation subsystem is responsible for selecting an application configuration most suitable for meeting the constraints given available resources, and dynamically updating this selection as and when resource availability changes. Such configuration and adaptation are realized by interactions between three components: (1) an *application-specific monitoring agent* that monitors resource characteristics of interest to the application, (2) a *resource scheduler* that correlates observed resource characteristics and user preferences with performance models stored in the performance database, and (3) a *steering agent* that performs the actual reconfiguration.

Each user preference constraint is expressed as value ranges on a subset of output quality metrics and is accompanied with an objective function to be optimized.[3] These constraints when considered together with measured resource characteristics, restrict the suitable set of application configurations. Of these, the scheduler picks the one that best satisfies the objective function. Multiple user preference constraints can be specified. The system examines them in decreasing order of preference; in the case that one request cannot be satisfied due to inadequate resources, the system attempts fulfill the next preferred constraint.

The rest of this section describes the monitoring, the scheduling, and steering components in more detail.

[3]For simplicity, we assume a relatively restricted form of this function: maximizing or minimizing a single quality metric.

6.1 Monitoring Agent

The monitoring agent is automatically generated by preprocessing the source code tunability annotations. The monitoring agent, running as an application module, continually observes application activities and estimates the fraction of resources that are available for use by the application. The motivation for such continuous monitoring is that a static view of resource characteristics is insufficient in shared environments comprising heterogeneous resources with different degrees of control over allocation of their resources.

The monitoring agent attempts to estimate the shortfall between the level of resources requested by the application from the system and what it actually obtained. For instance, changes in available network bandwidth can be detected by observing that a message send incurs more delay than would be expected given a prior measurement of the same. The monitoring agent employs the same mechanisms previously described in the context of the virtual execution environment in Section 5, relying on a system-wide monitor to provide information about maximum capacities of system resources (CPU speed, physical memory pages, network bandwidth, etc.). The monitor computes available resource capacities by comparing alloted CPU time with the wall clock time (factoring in periods where the application is waiting), comparing physical memory usage with virtual memory size, and keeping track of aggregate network traffic in a particular duration.

The monitoring agent runs periodically (every 10 ms) and processes raw data within a history window. The result is an estimate about resource availability, which is supplied to the resource scheduler and other monitoring agents in remote instances of this application. Our experience shows that this data collection adds negligible overhead to application execution even when performed in very fine-grained time slots. The behavior of the monitoring agent is customized to the currently active configuration, affecting both which resources are monitored, and when the information is communicated to the scheduler (only when resource availability falls out of a range) and other monitors.

6.2 Scheduling

Scheduling distributed applications requires placing a set of competing applications, each with multiple distributed instances, on a collection of interconnected machines with the purpose of optimizing application and system performance [17]. Scheduling tunable applications adds another dimension to this traditional problem. The scheduler needs to decide which alternate execution path (correspondingly application configuration) to choose for a tunable application. While tunability does not reduce the complexity of the scheduling itself, the availability of multiple application configurations increases the likelihood that application user preference constraints will be satisfied over a range of resource situations [8, 9].

The choice of which configuration to select given the application input, current resource conditions, and user preferences is guided by the resource profiles of each configuration stored in the performance database. Given our focus on mechanisms for configuring tunable applications, we adopt a relatively straightforward heuristic to choose the configuration. The measured resource characteristics and required user preferences (expressed as allowable value ranges on application quality metrics) are used to prune candidate configurations. Of the configurations that remain, a simple multidimensional optimization approach is used to pick the one that best satisfies the user-specified objective criterion. When resource conditions do not fit the records in the performance database, interpolation (or even extrapolation) of the representative data is used to predict the application performance (i.e., values for its quality metrics). If no candidate configurations exist, the next preferred user constraint is examined.

Two issues must be addressed for realizing the selected schedule: (1) system must have enough resources, and (2) applications must not be allowed to use more than their share of resources. The first can be solved by admission control and reservation. The second issue requires policing the applications, constraining their resource usage so that overuse does not happen. Both of these issues can take advantage of our resource-constrained execution environment, which was previously used as a testbed for automatically generating the performance database. Since multiple such execution environments can operate on the same physical machine with negligible overhead, we can reserve a specific CPU share (as well as network bandwidth and amount of physical memory) with simple admission control. For example, the application can be admitted if the total request for CPU share across all applications is less than a certain threshold. Once admitted, the resource-constrained execution environment monitors and controls application progress, assuring applications the required resource capacity and sandboxing them so that they do not overuse resources.

6.3 Steering Agent

The steering agent is responsible for ultimately switching application configurations, executing any clean-up code as appropriate. The steering agent receives control messages either from the resource scheduler or from other distributed instances of the application. These messages specify new values for control parameters as well as the resource conditions under which these new settings are valid. Upon receiving them, the steering agent executes corresponding handlers, which sets up the new configuration. The new setting only takes effect at the beginning of a task boundary, or at the transition points specified by the language annotation (see Section 4). At these points, the steering agent sends an acknowledgement to the resource scheduler; because of guard associated with these transitions, additional negotiation may be required between the steering agent and resource scheduler to find an appropriate setting. This new setting remains valid as long as the resource availability satisfies the resource conditions for the chosen configuration. When the monitoring agent finds out that the resource conditions have changed, it triggers the resource scheduler to find a new configuration.

7 Adaptation in the Visualization Application

In this section, we describe how our framework enables the active visualization application adapt at run time to changes in available resource characteristics.

The interactive nature of the application implies that both output quality and timeliness are of interest. The first implies a higher resolution level for the output image, and the latter implies some combination of a low response time and low overall image transmission time. However, the importance of these quality metrics varies from situation to situation. To capture a wide range of usage scenarios, we describe three experiments, each with unique user objectives. The first experiment asks for high image quality and minimized image transmission time. The second experiment emphasizes timeliness, requiring that each image transmission be finished by a rigid deadline while trying to get images of the highest resolution. In this case, the underlying system attempts to maximize the resolution that can be delivered without violating timing constraints. The third experiment emphasizes responsiveness, requiring that individual image components be delivered within some bounded time while preferring to get the entire image as soon as possible. As we shall see, the application needs to continually monitor resource availability and switch configurations at run time in order to satisfy these objectives in the presence of run-time variations in resource characteristics.

The rest of this section first describes the implementation status of the general framework and then shows how the framework helps the active visualization application achieve the above objectives.

7.1 Implementation Status and Experiment Setting

At the current time, the virtual execution testbed, the monitoring infrastructure, and early versions of the performance database and resource scheduler have been fully implemented. The preprocessor that generates the tunable version of the application components is under development, so the experiments reported here rely on manually generated steering code for switching between different application configurations. The performance database is populated by running each application configuration under the virtual execution testbed using a simple driver loop that looks up a configuration file listing the various application configurations, their control parameters, and a collection of manually determined resource settings. The latter compensates for the current lack of a sensitivity analysis tool that can automatically drive the collection of performance data in the most relevant regions of a multidimensional resource space. An additional limitation of the framework as it currently exists is that the resource scheduler does not do any interpolation on the performance profiles; a new configuration is selected by examining discrete points in the performance database that provide the best match to the measured resource condition.

The server and client components of the active visualization application are run on two Pentium II (450 MHz) machines connected by 100 Mbps Ethernet. The performance database is generated as described earlier in Section 5; a subset of the performance profiles used in the experiments reported here are shown in Figures 5 and 6 in Section 5. For simplicity, we restrict our attention in these experiments to variations in CPU and network resources, keeping memory resources at a fixed level.

Each experiment emulates the downloading of ten images from the server. To test the capability of the application to adapt to run-time variations in resource conditions, we vary one of the resources (either CPU share or network bandwidth) after a fixed time into the experiment. For simplicity, we restrict our attention to only the client side of the application and vary only CPU and network resources. Resource variations are enforced by changing the parameters of the virtual execution environment within which the client executes forcing the latter to adapt by switching to a different configuration. Changes in application configurations are communicated to the server component by sending control messages.

Figure 7 shows different application quality metrics versus time as resource availability at the client is varied. In each plot, the thick line represents the performance of the tunable application and the two thinner lines represent the (non-adaptive) performance of the two configurations that the adaptive application switches amongst.

7.2 Experiment 1: Adapting Compression Method to Network Conditions

Figure 7(a) shows the adaptation of the active visualization application in response to changes in the network bandwidth available between server and client. The user preference is to minimize image transmission time.

The network bandwidth is varied as follows: at the start of the experiment, the virtual execution environment provides a bandwidth of 500 KBps, which is changed to 50 KBps after 25 seconds. The resource scheduler responds to this pattern of resource availability as below:

- At startup, it configures the application to use compression method A (LZW), leading to faster transmission times but requiring additional network bandwidth. This decision is based on the resource availability and the application behavior described in performance database. As Figure 6(a) shows, for a bandwidth of 500 KBps, compression method A (LZW) outperforms compression method B (Bzip2); the former delivers a transmission time of around 5 seconds, while the latter requires more than 12 seconds. This choice allows the application to download four images before the available bandwidth changes at time 25 seconds.
- The change in bandwidth is detected by the application monitoring agent before the end of the fifth image transmission. Aware that current resource availability is well below the resource requirement of the original configuration, it informs the resource scheduler, which reconfigures the application to switch to compression B (Bzip2). Again, this selection is based upon a correlation of resource availability and the application behavior described in the performance database. As Figure 6(a) shows, compression method B yields better performance than compression method A when the bandwidth is 50 KBps. The switch takes effect in the middle of transmitting the fifth image, which takes 16 seconds

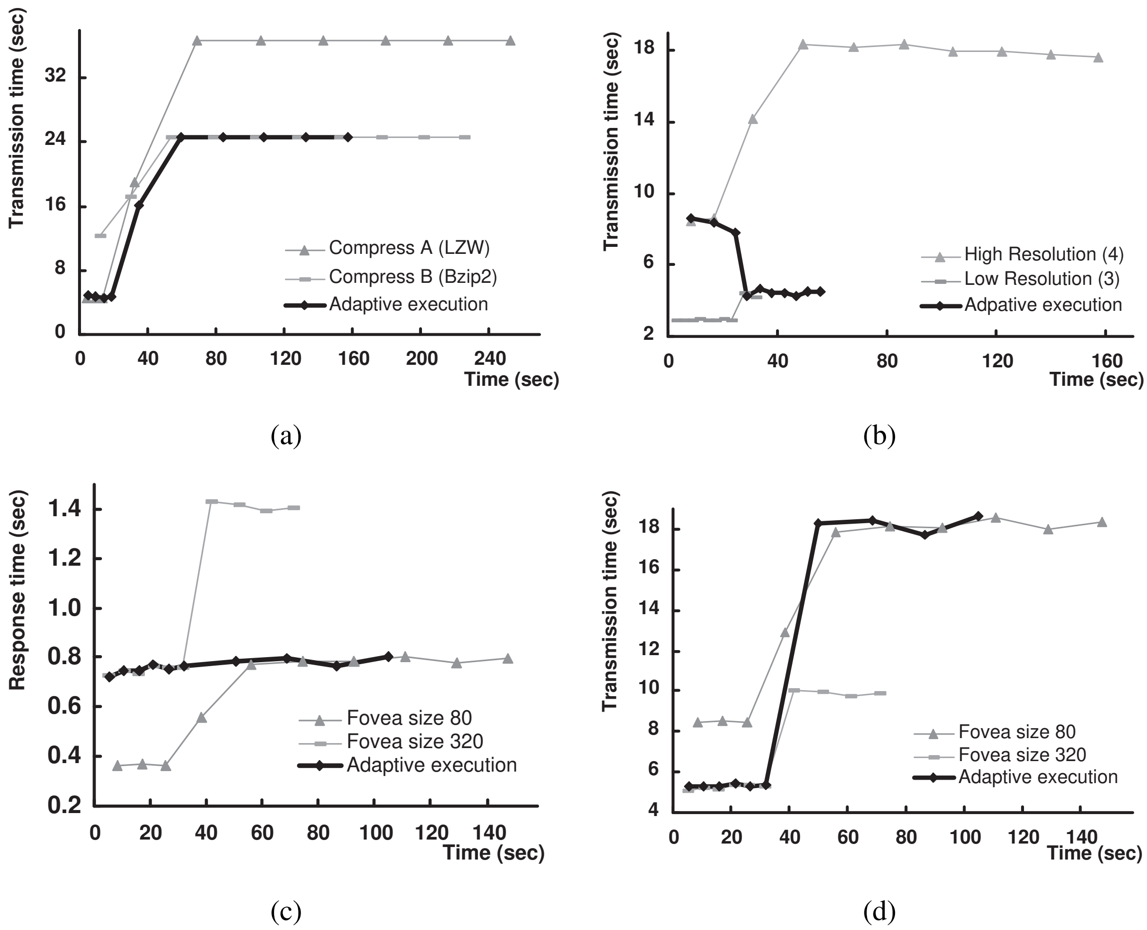


Figure 7: Application adapts to resource variations by: (a) switching compression methods when network bandwidth drops, (b) degrading image resolution as CPU share decreases, (c) and (d) changing fovea size as CPU share changes.

to complete. Subsequent image transmissions use compression method B and complete in 24 seconds apiece.

Thus, our framework allows the application to correctly configure itself and adapt its behavior at run-time to respond to changes in resource conditions (in this case, network bandwidth). As Figure 7(a) shows, the behavior of the adaptive application closely tracks the behavior of the appropriate (non-adaptive) configurations during periods of steady resource availability (compression method A before time 25 seconds, and compression method B afterwards). Just selecting the right initial configuration is not adequate: if the application had continued with compression method A, the total transmission time would have been 260 seconds as opposed to the 160 seconds it ends up taking.

7.3 Experiment 2: Adapting Image Resolution to CPU Conditions

Figure 7(b) shows the application adaptation in response to changes in CPU conditions (at the client). In this case, the user preference requires that image transmission time not exceed 10 seconds and that image quality be maximized. For simplicity, we constrain image resolution to be one of two levels (referred hereafter as level 3 and level 4).

The CPU conditions at the client are varied as follows: at startup, the CPU share available to the client is set at 90%, which changes to 40% at time 30 seconds. The resource scheduler responds to this pattern of resource availability by starting off with a configuration that sets the image resolution to be level 4, but then degrades image quality to level 3 upon detecting a reduction in CPU resources. These choices are consistent with the performance profile shown in Figure 6(b). When the CPU share is 90%, resolution level 4 results in image transmission times within the user requested range (less than 10 seconds). However, with a CPU share of 40%, image transmission times at this level of resolution consume take about 18 seconds apiece, which would violate user constraints. Degrading the resolution to level 3 permits each image to be transmitted in about 4 seconds, satisfying user requirements.

7.4 Experiment 3: Adapting Fovea Size to CPU conditions

Figures 7(c) and 7(d) show application adaptation by changing fovea size in response to changes in CPU conditions (at the client). In this case, the user preference is to minimize image transmission time while keeping average response time of user interactions below one second.

The CPU conditions at the client are varied as follows: initially, the CPU share is set to 90%, but decreases to 40% at time 40 seconds. The two figures show response time and transmission time of executions under this resource availability pattern. The resource scheduler responds to this pattern by initially selecting a fovea size of 320, and switching down to a fovea size of 80 upon detecting a change in resource availability. As before, these selections are consistent with the performance profiles shown in Figure 5. A fovea size of 320 satisfies the user preference of response times below one second, while achieving the fastest image transmission time of candidate configurations. However, when the amount of available CPU share drops, this configuration results in response times of about 1.4 seconds, which falls outside the user-requested range. Consequently, from the seventh transmission onwards, the scheduler switches to a fovea size of 80, ensuring response times of below one second for the remainder of the experiment.

7.5 Summary

These three experiments clearly demonstrate the ability of our adaptation framework as applied on the active visualization application to appropriately configure itself and adapt itself at run-time in response to changing resource availability patterns. The monitoring agent integrated with the application detects changes in resource availability and triggers the resource scheduler, which in turn suggests an alternate configuration that satisfies user preferences of application quality by relying upon a performance database of application resource profiles.

We should note that although our simple resource scheduler and performance database were sufficient to appropriately adapt the active visualization application in the above experiments, one of the main reasons for this is relatively large variations in resource availability. Smaller variations would require better algorithms that take into consideration the sensitivity of application configurations to resource variations so as to not degrade overall performance by unnecessary adaptations.

8 Related Work

Our work is most closely related to a few recently-started projects [3, 10, 14, 17, 18, 19] that are looking into the problem of adapting application behavior in response to variations of system resources. The Darwin project [17] permits an application to specify its resource requests in the form of a virtual mesh of nodes (representing desired services) and edges (denoting communication flows). The virtual mesh in its specific application domain is similar to our notion of alternate execution paths. ActiveHarmony [14] and

AppLeS [3, 19] projects provide application-level mechanisms and resource monitoring tools to enable an application to adapt itself to changing resource characteristics. EPIQ [18] and ErDos [10] projects are closer to our approach in that they look into the quality aspects of an application, trading off output quality against resource requirements.

Our approach differs from them in the division of responsibility between application developer and execution system. Application developers are concerned with annotating the original source code to specify application tunability. Execution system provides virtual environment for obtaining application behavior profiles. In addition, the resource scheduler is insulated from application-specific knowledge when configuring application. As far as we know, our strategy is unique in its using of a virtual execution environment for measuring application performance and in monitoring application-specific resource availability instead of the total capacity of resources.

9 Conclusion

This paper studies the language constructs for specifying application tunability. It proposes a way to structure tunable applications so that they could be automatically configured to take the appropriate execution path for better or more predictable performance. It validates the usage of software virtual machine as the testbed to study application behavior. It demonstrates automatic configuration of applications with a case study. Our results to date is encouraging: it is convenient to specify application tunability with language-level annotations. The virtual execution environment approximates the real execution platform and can be used to automate application behavior modeling. Significant performance improvement can be gained by dynamically adapting application execution to resource characteristics. Our framework provides a substrate for exploiting the tunability inherent in many applications, equipping them to "self-adapt" themselves in a variety of resource situations. More work is needed to automate analysis of application behavior and to study scheduling policies in terms of its sensitivity in response to resource variations.

Acknowledgments

We thank Ayal Itzkovitz for his contribution to the development of distributed resource-constrained sandbox. This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-96-1-0320 and F30602-99-1-0517; by the National Science Foundation under CAREER award number CCR-9876128; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

[1] R. Balzer and N. Goldman. Mediating connectors. In *Proc. 1999 ICDCS Workshop on Electronic Commerce and Web-based Applications*, Jun. 1999.

[2] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. 3rd USENIX Symp. on Operating Systems Design and Implementation*, 1999.

[3] F. Berman and R. Wolski. Scheduling from the perspective of the application. In *Proc. 5th IEEE Intl. Symp. on High Performance Distributed Computing*, 1996.

[4] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated service. IETF Network Working Group, RFC 2475, Dec. 1998.

[5] E.C. Chang and C. Yap. A wavelet approach to foveating images. In *Proc. 13th ACM Symp. on Computational Geometry*, 1997.

[6] E.C. Chang, C. Yap, and T.-J. Yen. Realtime visualization of large images over a thinwire. In *IEEE Visualization*, 1997.

[7] F. Chang, A. Itzkovitz, V. Karamcheti, and Z. Kedem. Resource-constrained sandbox. In preparation.

[8] F. Chang, V. Karamcheti, and Z. Kedem. Exploiting application tunability for efficient, predictable parallel and distributed systems. Invited submission to Journal of Parallel and Distributed Computing.

[9] F. Chang, V. Karamcheti, and Z. Kedem. Exploiting application tunability for efficient, predictable parallel resource management. In *Proc. 13th Intl. Parallel Processing Symposium*, 1999.

[10] S. Chatterjee. Dynamic application structuring on heterogeneous, distributed systems. In *Proc. IPPS/SPDP'99 Workshop on Parallel and Distributed Real-Time Systems*, 1999.

[11] P. Goyal, X. Guo, and H.M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. Operating System Design and Implementation*, Oct. 1996.

[12] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proc. 3rd USENIX Windows NT Symposium*, Jul. 1999.

[13] M. Jones and J. Regehr. CPU reservations and time constraints: Implementation experience on Windows NT. In *Proc. 3rd USENIX Windows NT Symposium*, Jul. 1999.

[14] P. Keleher, J. Hollingsworth, and D. Perkovic. Exploiting application alternatives. In *Proc. 19th Intl. Conf. on Distributed Computing Systems*, Jun. 1999.

[15] C. Lee, J. Lehoczky, R. Rajkumar, and D. Siewiorek. On quality of service optimization with discrete QoS options. In *Proc. IEEE Real-time Technology and Applications Symposium*, Jun. 1999.

[16] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proc. IEEE Intl. Conf. on Multimedia Computing and Systems*, May 1994.

[17] P. Chandra et al. Darwin: Customizable resource management for value-added network services. In *Proc. Sixth IEEE Intl. Conf. on Network Protocols*, 1998.

[18] M. Shankar, M. DeMiguel, and J. Liu. An end-to-end QoS management architecture. In *Proc. Real-Time Applications Symposium*, Jun. 1999.

[19] N. Spring and R. Wolski. Application level scheduling of gene sequence comparison on metacomputers. In *Proc. 12th ACM Intl. Conf. on Supercomputing*, Australia, 1998.

[20] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource ReSerVation Protocol. *IEEE Network*, Sep. 1993.