To appear in Ninth IEEE Symposium on High Performance Distributed Computing (HPDC), August 2000.

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

Increased platform heterogeneity and varying resource availability in distributed systems motivates the design of resource-aware *applications, which ensure a desired performance level by continuously adapting their behavior to changing resource characteristics. In this paper, we describe an application-independent adaptation framework that simplifies the design of resource-aware applications. This framework eliminates the need for adaptation decisions to be explicitly programmed into the application by relying on two novel components: (1) a* tunability interface, *which exposes adaptation choices in the form of alternate application configurations while encapsulating core application functionality; and (2) a* virtual execution environment, *which emulates application execution under diverse resource availability enabling off-line collection of information about resulting behavior. Together, these components permit automatic run-time decisions on* when *to adapt by continuously monitoring resource conditions and application progress, and* how *to adapt by dynamically choosing an application configuration most appropriate for the prescribed user preference.*

We evaluate the framework using an interactive distributed image visualization application. The framework permits automatic adaptation to changes in CPU load and network bandwidth by choosing a different compression algorithm or controlling the image transmission sequence so as to satisfy user preferences of visualization quality and timeliness.

1. Introduction

Current day distributed applications are written to 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. However, a consequence is that applications often exhibit diverse and unpredictable performance both because of platform heterogeneity and varying resource availability. Fortunately, the availability of operating system mechanisms for increased control over system utilization (e.g., fair-share CPU scheduling [2, 10, 14, 18] and QoS-aware network protocols [4, 23]) suggests a way for providing predictable application performance. *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 can respond to a reduction in available network bandwidth by compressing the stream or selectively dropping frames.

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 general-purpose resource-aware applications. Several researchers have begun to address this shortcoming [9, 15, 16, 19, 21]; however, most such efforts place a substantial 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 alleviates the burden on the programmer by relying on two novel techniques: (1) a *tunability interface*, which separates specification of potential adaptation behavior from core application functionality; and (2) a *virtual execution environment*, which emulates diverse run-time resource availability in a controlled fashion on top of a static distributed system. The tunability interface exposes choices in application execution paths allowing enumeration of alternate application configurations, and the virtual execution environment permits automatic and off-line generation of profiles about how each configuration behaves under different resource conditions. Together, the two mechanisms enable the development of a run-time adaptation system, which continuously monitors resource conditions and application progress (in terms of user preferences of QoS metrics), and automatically determines both *when* adaptation should be performed and *how* the application must be modified (i.e., which of its configurations must be chosen) based on application profiles. The automation of the adaptation procedure ensures that the application achieves a desired level of performance, while minimizing the programmer's involvement in performance-related considerations.

This framework has been implemented in the context of distributed applications running on Windows NT platforms. The tunability interface is exposed using language-level annotations that allow a preprocessor to generate monitoring and steering agents as well as control messages, permitting external access to and control of the application execution. The virtual execution environment is implemented using a novel technique called API interception that emulates different availability of system resources by continuously monitoring and controlling application requests for system resources. Run-time adaptation takes advantage of the steering and monitoring agents.

We describe evaluation of this framework using a distributed interactive image visualization application. Our experiments demonstrate that starting from a natural specification of alternate application behaviors, it is feasible to model application behavior using the virtual execution environment in sufficient detail to automatically adapt the application at run time to changes in CPU load and network bandwidth. The application adapts by choosing a configuration that employs a different compression method or one where the image transmission sequence is modified to satisfy user preferences of image quality and timeliness.

The rest of this paper is organized as follows. Section 2 introduces the Active Visualization application which serves as a running example. Sections 3 and 4 provide details about the tunability interface and virtual execution environment, and Section 5 describes their use for adaptation in the active visualization application. Related work is discussed in Section 6 and we conclude in Section 7.

2. Active Visualization Application

The active visualization application [5, 6] is a client-server application for interactively viewing, at the client side, large images stored in the server. Figure 1 shows the overall structure of the client and server processes. The application uses several multi-resolution and progressive transmission techniques to improve performance. First, images are stored at the server as wavelet coefficients [5], enabling the construction of images at different levels of resolution. Second, the server employs progressive transmission to improve response 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 (focus of interest), 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 required resolution. Finally, image data is compressed to reduce bandwidth requirements.

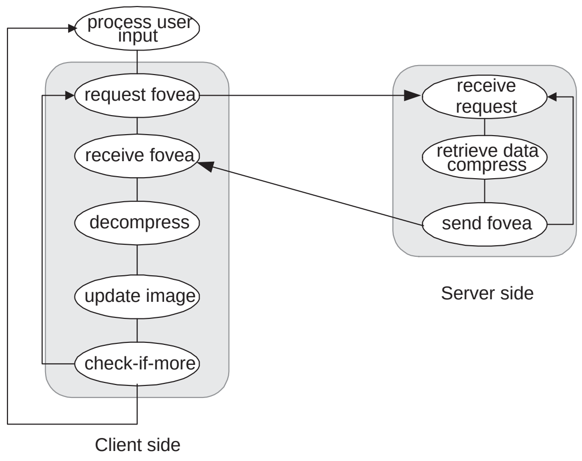


Figure 1. Structure of Active Visualization Application. The client requests a foveal region of the specific resolution level from the server, decompresses the data, and updates the local image on display.

This application permits adaptation by permitting association of different values with parameters such as preferred resolution level, size of the foveal region, and choice of the compression method. The resultant executions trade off resource requirements for application performance: a low resolution implies less resource requirements; different compression methods require different CPU resources; and a small fovea size leads to a quicker response for getting the foveal region, but a longer time to transmit the whole image.

3. Application Structuring

Adaptation requires the existence of multiple ways of executing an application, or *alternate configurations*, which exhibit different resource utilization profiles. These multiple profiles provide run-time flexibility, permitting choice of an alternate configuration better suited to matching user preferences given available system resources. The alternate configurations can be built out of different application modules or with the same module exhibiting different behaviors under the influence of some control parameters. For instance, a video compression component can trade off compression ratio against image quality either within the context of a single algorithm, or conceivably by switching between different algorithms to allow a bigger tradeoff range.

Although transitioning among different configurations can be achieved completely at the application level, it is better controlled at the system level without extensive programmer involvement. However, this requires some application-independent way of enumerating what the different configurations are as well as mechanisms for effecting transitions among the various alternatives. We meet these requirements by relying on a structuring concept called the application *tunability interface*, which was introduced in earlier work done in the context of the Calypso parallel programming language [8]. The tunability interface provides language support to express the availability of multiple execution paths for the application, as well as providing means by which application progress can be monitored and influenced (by switching to a different path).

In the context of distributed applications of the kind considered in this paper, the tunability interface takes the following form. Applications are assumed to be built up from multiple components linked together using standard control-flow mechanisms. Different application configurations correspond to different behaviors for these components, which are exported in terms of the tunability interface. In general, the tunability interface of a component provides five kinds of information:

1. *Control parameters* and their value ranges, which provide the “knobs” using which component behavior can be influenced. Setting different values to the control parameters results in the component following a different execution path. The control parameters associated with the component name are evaluated as name-value pairs when the component is instantiated at run time, and serves as a handle for referring to a specific application configuration.

2. *Execution environment*, which specifies the system entities (hosts and network links) on which the application component executes. Each system entity in turn encapsulates several resources that affect application behavior. For instance, a host is characterized by its CPU, memory, and network resources.

3. *QoS metrics* and expressions for evaluating them, which specify the component metrics of interest as well as application-provided means of computing them. We require that different values of the same QoS metric can be compared with each other.

4. *Tunable modules* and inter-module control flow, which set up the alternate execution paths. Each module is specified by a `task` construct. Component execution paths are specified by associating guard expressions of control parameters with each task and specifying inter-task control flow (using sequencing, conditional, and looping constructs).

5. *Transition functions*, which encapsulate application-specific actions (e.g., updates to local variables and/or control messages to other components) required upon a change in the component configuration.

To implement the above interface, we currently rely upon unverified source-level annotations, which are processed by a simple macro preprocessor. The control parameters, QoS metrics, and execution environment definitions are converted into data structures in the source language (e.g., structure declarations in C/C++), accessible to other annotation constructs. For instance, references to two copies of the control parameter structure are passed to each transition function: a copy containing the current values and a copy containing values these control parameters need to be set to after the transition function is executed. These two copies correspond to the different application configurations bridged by the transition function. Different execution paths are achieved by expanding the tunable module constructs into conditional statements involving the associated guard expressions. Changes in component configurations are detected by inline checking code.

In addition to the tunable versions of each component, the preprocessor also generates *monitoring* and *steering* agents that monitor resource availability and control application execution respectively (see Section 5 for details). These agents interact with the application components by accessing the tunability data structures using shared memory. Finally, the preprocessor sets up information about the different application configurations as a file that serves as input to both the virtual execution environment (see Section 4 for details) and the external resource scheduler. This configuration file identifies the application components, and their control parameters, execution environments, and QoS metrics. Availability of this information allows an external source, either a driver program or a system-level scheduler, to flexibly interact with a running application for performing a variety of tasks ranging from passive measurement (e.g., recording the performance achieved by a particular configuration under a given resource condition) to more active control (e.g., changing the behavior of application components).

3.1. Tunability in Active Visualization

Figure 2 shows the adaptation structure of the client side of the Active Visualization application as exposed using the tunability interface. The progressive refinement component

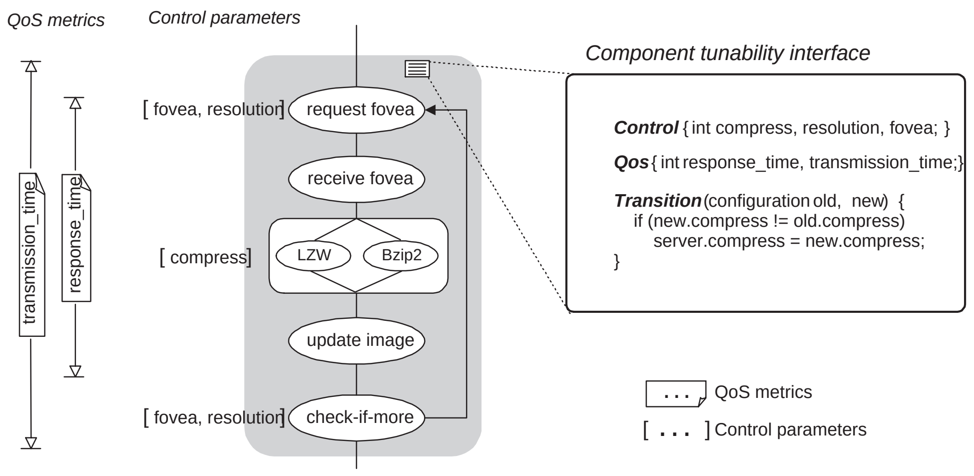

QoS metrics
Control parameters
Component tunability interface
[fovea, resolution]
request fovea
receive fovea
[compress]
LZW
Bzip2
update image
[fovea, resolution]
check-if-more
transmission_time
response_time
Control { int compress, resolution, fovea; }
Qos { int response_time, transmission_time;}
Transition (configuration old, new) {
if (new.compress != old.compress)
server.compress = new.compress;
}
... QoS metrics
[...] Control parameters

Figure 2. Exporting component tunability interface for client slide of Active Visualization. Tunability interface of a component includes control parameters, QoS metrics, and transition functions called upon configuration changes.

is controlled using three control variables—fovea, resolution, and compress—which influence the foveal region size, the image resolution, and the compression method affecting the behavior of the request, decompress, and check-if-more components. Two QoS metrics—response time and transmission time—measure performance of this component. Finally, the transition function updates control parameters in the server instance of the application whenever there is a change in the compression method. This update takes the form of a control message sent to the server. The execution environment (not shown) here includes CPU speed and network bandwidth at both the client and the server machines.

The component tunability interface enables the system to learn about the application's state and control its execution from outside without application specific knowledge. Together with the virtual execution environment described next, this enables the system to automatically decide when adaptation is needed by monitoring resource conditions and application state, and decide on the configuration that should be executed next.

4. Modeling Application Behavior

Explicit specifications of application tunability enable the development of a model of behavior for each of the application configurations, expressed as the mapping from control parameters (application input can be viewed as an additional control parameter) and resource conditions to application-specific quality metrics. Due to the difficulty in obtaining an analytic expression, profile-based modeling is used to approximate this mapping.

In principle it is possible to obtain these profiles by executing each configuration in different distributed environments corresponding to every possible run-time resource availability situation. However, practical considerations rule against this approach because of the difficulty of configuring such a large variety of distributed systems. Instead, we obtain the profiles using a technique for creating *virtual execution environments*. These environments run on top of a static distributed system, and can be configured to accurately emulate a variety of resource availability scenarios. Our implementation of these environments (described in additional detail below) relies on standard mechanisms available in most current-day operating systems and a novel technique called API interception, and supports the constrained execution of unmodified applications.

Given such environments, obtaining profile-based models of configuration behavior is straightforward. A driver script executes each configuration repeatedly setting up the virtual execution environment to sample different resource conditions. A separate tool analyzes output quality measures to determine configurations and regions of the resource space that require additional samples. The output of this modeling step is a *performance database* that records information about a maximal subset of the configurations representing the resource profile of this application.[1] These measurements are interpolated to get performance curves that summarize configuration performance.

4.1. Implementing the Virtual Environment

We implement a virtual execution environment by effectively creating a "sandbox" around an application, which constrains application utilization of system resources such

[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.

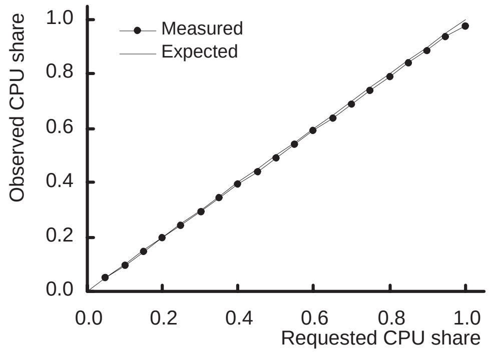
as the CPU, memory, disk, and network. Our specific implementation on the Windows NT platform relies on the ability to inject arbitrary code into a running application, using a technique known as API interception [1, 12]. This code continually monitors application requests for operating system resources and estimates a "progress" metric (e.g., what fraction of the CPU share has the application been receiving). Upon detecting that the application has received either more or less than the share configured for the virtual environment, the injected code actively *controls* future application execution, relying on generally available OS mechanisms, to ensure compliance with specified limits. For instance, the code can intercept application's API calls to access disk resources and delay the operations if the application has exceeded its disk bandwidth limit. This technique allows accurate control over resource availability to applications on commodity OSes without modification to either OS or application source code. A detailed description of the techniques used for constructing the virtual execution environment and its capabilities n both Windows NT and Linuxis reported elsewhere [7].

Figure 3 demonstrates the overheads and accuracy of the virtual execution environment. Figure 3(a) compares the execution time of a tight loop on the virtual execution environment (realized on a Pentium II 450MHz machine) and the expected execution time (normalized with the requested share) as CPU share varies from 5% to 100%. The application's execution time under the virtual execution environment is very close to what is expected (except when 100% CPU share is requested) indicating that the environment incurs low overhead. Figure 3(b) shows the accuracy of the virtual execution environment by comparing execution times for the active visualization application on different real client machines (a Pentium Pro 200MHz, a Pentium II 333MHz, and a Pentium II 450MHz) with that obtained under the virtual environment when the latter is configured to provide a CPU share based on the WinBench 99 v1.1 scores for the different machines.[2] The execution times under the virtual environment are within 3% of that on the physical machines. Note that similar accuracy cannot be obtained by simply scaling the execution time in proportion to the CPU shares. This estimate, shown by the "Linear scale" bar, turns out to have a big difference from real executions (primarily because of idle time behavior), indicating the necessity to measure application behavior under different resource conditions.

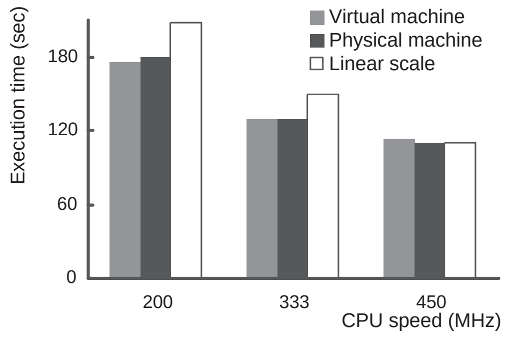
4.2. Profiles for Active Visualization

A driver script repeatedly executes different configurations of the active visualization application in the virtual execution environment, obtaining a mapping from the control parameter values to the output qualities for a wide range of resource conditions. Figures 4 and 5 discusses a small subset of these measurements and the resulting mappings. The measurements reported here are obtained on a Pentium 450MHz machine andrely upon manual determination of appropriate resource settings for the testbed since the sensitivity analysis tool is currently under development.

[2]WinBench is a Windows 98/NT benchmark that measures the aggregate impact of CPU speed and memory configuration.



(a)

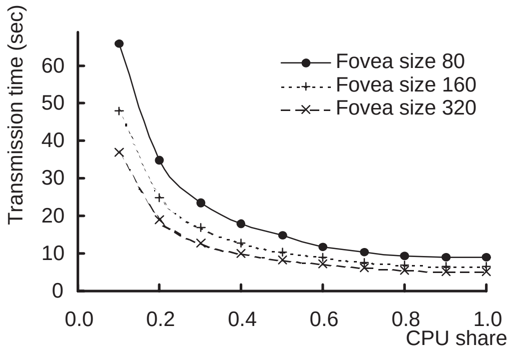


(b)

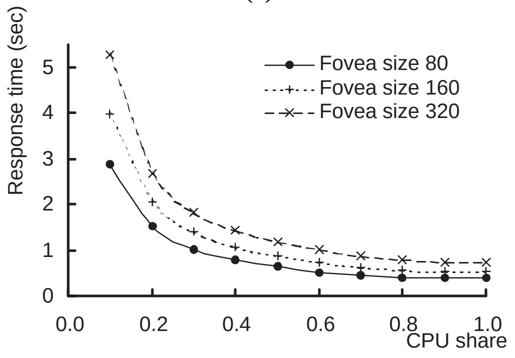
Figure 3. (a) Overheads and accuracy of the virtual execution environment with a tight loop. (b) Comparison of application execution times using the virtual execution environment and physical machines for the active visualization application.

Figure 4 shows the image transmission time and average response time of user interactions for different fovea sizes as the client-side CPU share varies (this corresponds to running the application on client machines with different CPU capabilities). In general, an increase in CPU resources reduces both transmission time and response time. However, they show opposite trends (seen in the order of the curves) with the increase of fovea size: the larger the fovea size, the smaller the total transmission time, but the larger the response time.

Figure 5(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



(a)



(b)

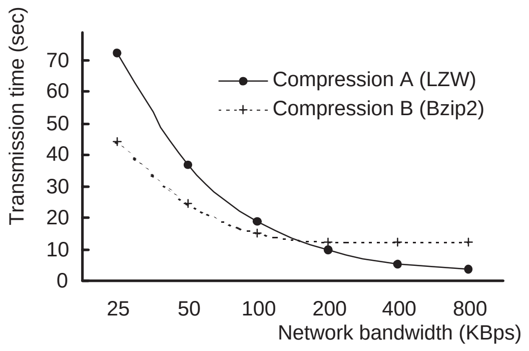
Figure 4. Image transmission time (a) and response time (b), for different fovea sizes as CPU share varies.

two curves in the figure correspond to two different compression methods in the application: compression B (Bzip2) trades off additional CPU resources to achieve a better compression ratio than compression A (LZW). 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 compression A performs better when the network bandwidth is high because the CPU becomes the bottleneck then.

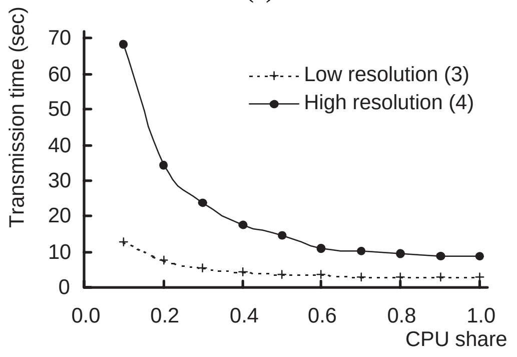
Figure 5(b) shows the transmission time for images of different resolutions as the CPU resource varies (keeping other resources at a fixed level). In general, additional CPU resources lead to a shorter transmission time. However, transmission time can also be lowered by lowering the image resolution.

5. Run-time Adaptation

The program annotated with tunability interface specifications is converted by a preprocessor into an executable form; the latter includes application modules that make up the different paths as well as steering and monitoring agents



(a)



(b)

Figure 5. Image transmission time, for (a) different compression methods as network bandwidth varies, and for (b) images of different resolutions as CPU share varies.

used for run-time adaptation.

5.1. Run-time Monitoring and Steering

At run time, an appropriate application configuration is chosen to satisfy user preference constraints, and this selection is dynamically updated as resource availability changes. Such configuration and adaptation are realized by interactions between three components (see Figure 6): (1) an *application-specific monitoring agent* that monitors resource characteristics of interest to the application as well as application progress, (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

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

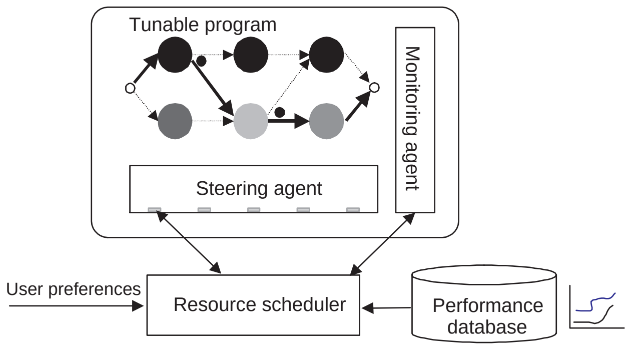


Figure 6. Runtime components (e.g., monitoring agent, steering agent, and resource scheduler) cooperate to enable application-independent adaptation to satisfy user preferences.

configurations. Of these, the *resource 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 to fulfill the next preferred constraint.

The *monitoring agent* continually observes application progress and estimates the fraction of resources of interest-that are available for use by the application. To do this, the agent relies on generic progress metrics similar to those described in Section 4 (e.g., comparing allotted CPU time with the wall clock time after factoring in periods where the application is waiting) and a system database providing information about maximum capacities of system resources (such as CPU speed and the number of physical memory pages). Our experiments show that such monitoring incurrs negligible overhead to application execution even when performed in very fine-grained time slots.

The *steering agent* listens to control messages and is responsible for switching application configurations. Control messages specify new values for control parameters as well as the resource conditions under which these new settings are valid (because of user preference constraints). Upon receiving them, the steering agent sets up the new configuration, which would cause the transition function to execute at the activation of the corresponding component.

5.2. Adaptation in Active Visualization

Because of the interactive nature of the application, the output quality metrics of most interest to the user are image resolution and timeliness, although their relative importance varies from situation to situation. To capture a wide range of usage scenarios, we describe three experiments below, demonstrating that our framework permits successful automatic adaptation to different patterns of change in resource availability. Figure 7(a)–(d) shows the quality metrics achieved by the adaptable form of the application and contrasts it with the non-adaptive versions. 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.

The experiments emulate the client downloading ten images from the server and correspond to the server and client components running on two Pentium II 450MHz machines connected by 100 Mbps Ethernet. In this paper, we focus on adaptation to variations in CPU and network resources, and on only the client side of the application since the latter is more likely to be concerned with output quality metrics such as image resolution and response time. To test whether the application can adapt to run-time variations in resource conditions, we vary one of the resources (either CPU share or network bandwidth) after a fixed time in the experiment. As a final note, the experiments reported here rely on manually generated steering code for switching between different application configurations. Note that this step can be automated and we are currently developing a preprocessor that will generate the adaptable version of the application components using annotations for the tunability interface described in Section 3.

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). As Figure 5(a) shows, for a bandwidth of 500 KBps, compression method A (LZW) outperforms compression method B (Bzip2). 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, which notifies the resource scheduler. The latter (based on the correlation in performance database) suggests the application to switch to compression method B (Bzip2), which the application-specific transition function realizes by sending the corresponding control message to the server (since compression is performed at the server side). As Figure 5(a) shows, compression method B yields better performance than compression method A when the

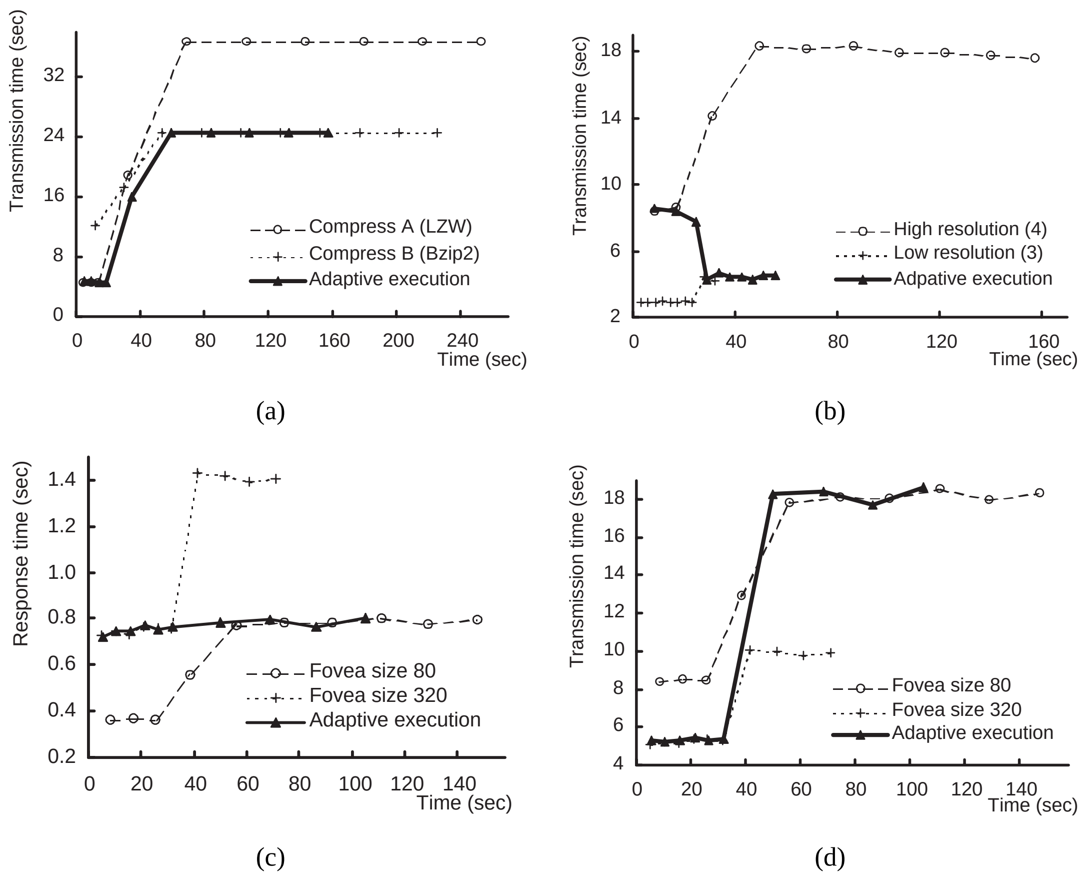


Figure 7. Application adapts to resource variations by: (a) switching compression methods when network bandwidth drops, (b) degrading image resolution as CPU share decreases, (c) and (d) changing fovea size as CPU share changes.

bandwidth is 50 KBps. The switch takes effect in the middle of transmitting the fifth image, which takes 16 seconds to complete. Subsequent image transmissions use compression B and complete in 24 seconds apiece.

Experiment 2: Adapting image resolution to CPU conditions Figure 7(b) shows the application adaptation in response to changes in CPU conditions. 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 to 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 when resource availability drops. These choices are consistent with the performance profile shown in Figure 5(b). When the CPU share is 90%, resolution level 4 results in image transmission times within the user-requested range (i.e., less than 10 seconds). However, with a CPU share of 40%, image transmission times at this resolution level would violate user constraints. Degrading the resolution to level 3 permits each image to be transmitted in about 4 seconds, satisfying user requirements.

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. In this case, the user preference is to minimize image transmission time while keeping average response time of user interactions below 1 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 35 seconds. The two figures show response time (for retrieving the fovea) and transmission time (for retrieving the entire image) of executions under this resource availability pattern. The resource scheduler initially selects a fovea size of 320 (pixels), and switching down to a fovea size of 80 due the change in resource availability. These selections can be understood with the performance profiles shown in Figure 4. A fovea size of 320 satisfies the user preference for response times being below 1 second, while achieving the shortest image transmission time. 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, the scheduler switches to a fovea size of 80, ensuring response times of below 1 second for the remainder of the experiment. Compared with the configuration that satisfies this constraint (i.e., selecting fovea size 80), the adaptive execution achieves much shorter image transmission time.

6. Related Work

Our work is most closely related to a few recently-started projects that are looking into the problem of adapting application behavior in response to varying availability of system resources.

The Quality Object (QuO) framework [24, 20] supports the development of distributed applications with QoS requirements. It permits applications' execution to switch between different contract regions and their remote method invocations to be dispatched to alternate remote objects based on dynamic system conditions. Switching between contract regions and dispatching on alternate objects are similar to our notion of different application execution paths. The Darwin project [19] permits a flow-centric 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 can be mapped to physical service nodes and links in alternate ways to permit optimization of available resources. In addition, application-specific "delegates" can be associated with flows for detecting and handling changes to flow metrics. Both the virtual mesh and delegate notions play similar roles in their specific application domain that is closely related to our notion of alternate execution paths exposed using a tunability interface. The Remos system [17, 11] presents a structure for collecting resource information and a set of APIs for application to query resource information. It provides similar functionality to that of our monitoring agents. Apertos [13] proposes building adaptive operating system using the reflection mechanism. Meta-level objects (reflectors) are used to hold the information about system components and act as an interface to manipulate the components' execution environment. Our work can be used in an Apertos-like system for building components that can reason about themselves to adapt to changes in the environment. The ActiveHarmony [15] and AppLeS [3, 22] projects provide application-level mechanisms and resource monitoring tools to enable general applications to adapt to changing resource characteristics, albeit with a fair degree of user involvement. EPIQ [21] and ERDoS [9] projects are closer to our approach in that they expose quality aspects of an application, automatically trading off output quality against resource requirements. However, they focus on the negotiation aspects of configuration and an analytic specification of the benefit/utility function.

Our approach differs from the above approaches principally in the division of responsibility between application developers and the execution system. Application developers are required only to expose the adaptation structure of the application using the tunability interface. The execution system takes responsibility for obtaining application behavior profiles using the virtual execution environment, and incorporating these profiles into adaptation decisions at run time. The tunability interface insulates the resource scheduler from application-specific knowledge. As far as we know, our strategy is unique in its use of a virtual execution environment for automatically modeling application performance under diverse loads.

7. Conclusion

This paper describes a general framework for enabling application configuration and adaptation based on resource characteristics and user preferences. The framework relies on the application developer exporting an application-independent tunability interface that allows external access to and control of application execution. It models the behavior of alternate execution configurations in a virtual execution environment with controllable resource availability. These together permit run-time mechanisms to automatically decide when and how to adapt the application in reaction to changes in resource conditions. This paper demonstrates using a case study of a distributed image visualization application how such structuring of distributed applications can help achieve automatic application configuration and adaptation for prescribed user preferences. Our future work will include studying different policies for selecting appropriate configurations and analyzing application behavior for further automating the profile modeling process.

Acknowledgments

This research was sponsored by the Defense Advanced Research Projects Agency under agreement numbers F30602-96-1-0320, F30602-99-1-0157, and N66001-00-1-8920; 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, SPAWAR SYSCEN, 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. Chang and C. Yap. A wavelet approach to foveating images. In *Proc. 13th ACM Symp. on Computational Geometry*, 1997.

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

[7] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level resource-constrained sandboxing. In *Proc. Fourth USENIX Windows System Symposium*, Aug. 2000.

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

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

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

[11] T. Gross, P. Steenkiste, and J. Subhlok. Adaptive distributed applications on heterogeneous networks. In *Proc. 8th Heterogeneous Computing Workshop (HCW '99)*, Apr. 1999.

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

[13] J. Itoh, R. Lea, and Y. Yokote. Using meta-objects to support optimisation in the Apertos operating system. In *Proc. USENIX Conference on Object-Oriented Technologies (COOTS'95)*, Jun. 1995.

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

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

[16] 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.

[17] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *Proc. 7th IEEE Symposium on High-Performance Distributed Computing*, Jul. 1998.

[18] 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.

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

[20] P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier. Using QDL to specify QoS aware distributed (QuO) application configuration. In *Proc. Third IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC 2000)*, Mar. 2000.

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

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

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

[24] J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, Jan 1997.