

A Framework for Automatic Adaptation of Tunable Distributed Applications

Fangzhe Chang and Vijay Karamcheti

*Department of Computer Science
Courant Institute of Mathematical Sciences
New York University, New York, NY 10012
E-mail: {fangzhe,vijayk}@cs.nyu.edu*

Increased platform heterogeneity and varying resource availability in distributed systems motivate 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 and a parallel image processing application. The framework permits automatic adaptation to changes in execution environment characteristics such as available network bandwidth or data arrival pattern by choosing a different application configuration that satisfies user preferences of output quality and timeliness.

Keywords: Application adaptation, Quality of Service (QoS), Application performance optimization

AMS Subject classification: Distributed Computing, Adaptive Computing

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 monitoring of and control over system utilization (e.g., fair-share CPU scheduling [2,12,17,22] and QoS-aware network protocols [4,27]) 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 [9,19,20,23,25]; 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 and parallel 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 and parallel image processing 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, network bandwidth, data arrival pattern, and other environmental characteristics. Each application adapts by choosing a configuration that employs a different algorithm or one where the execution sequence is modified to satisfy user preferences of output quality (e.g., image resolution) and timeliness.

The rest of this paper is organized as follows. Section 2 introduces the Active Visualization and Junction Detection applications, which serve as running examples. Sections 3 and 4 provide details about the tunability interface and virtual execution environment, and Section 5 describes their use for adaptation in both parallel and distributed applications. Related work is discussed in Section 6 and we conclude in Section 7.

2. Example Applications

Many applications exhibit the flexibility to choose different paths of execution at run time, trading off resource requirements over several dimensions including output qualities, resource types, or computational stages. These applications are *tunable* in the sense that their flexibility permits adaptation by switching over to a different execution path at run time, ensuring user preferred performance levels despite changes in characteristics of the execution environment. In this section, we describe two example applications — a distributed image visualization application and a parallel image processing application — to examine how they permit run-time configuration and adaptation.

2.1. Active Visualization

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 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.

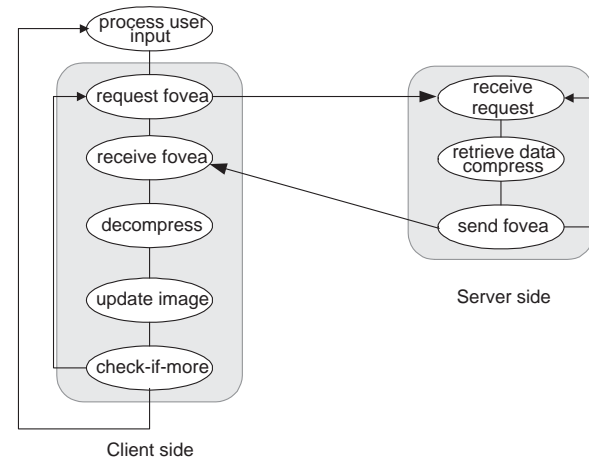


Figure 1. Structure of Active Visualization application. The client requests a foveal region of the specific resolution level from the server, decompresses the data, and updates the local image on display.

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 and network resources; and a small fovea size leads to a quicker response for getting the foveal region, but a larger number of rounds to transmit the whole image.

2.2. Junction Detection

The Junction Detection application [15] is a parallel image processing application running on the Calypso system [11,18], an adaptive parallel processing system which views computations as consisting of several parallel tasks inserted into a sequential program. Each parallel task can

run on a changing set of *worker* machines and is responsible for performing the computationally intensive work, while the sequential code runs on a dedicated *manager* machine and is responsible for the high-level control-flow and I/O. The parallel tasks can be executed multiple times (with possibly some partial executions), with exactly-once semantics. Within a parallel step, Calypso supports CREW (concurrent read, exclusive write) semantics to shared data structures, with updates visible only at the end of the step. Junction

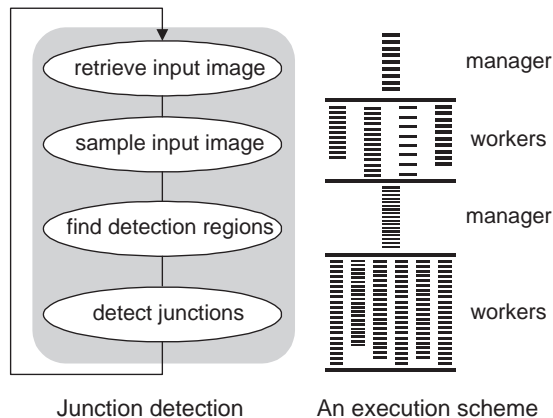


Figure 2. Structure of the parallel Junction Detection application and a possible processor-allocation scheme.

Detection is a core component of several image-processing applications and detects distinguished sets of pixels in an image where the intensity or color changes abruptly. Figure 2 shows the overall structure of the computation, which consists of a driver routine initiating a component for processing an individual image. The per-image component consists of three parallel steps. The first step samples a subset of the pixels in parallel and performs a quick test to determine whether or not the tested pixel is of interest. A pixel is of interest if the difference among intensities/colors of its neighbor pixels is beyond a threshold. The second step draws a region of interest around a cluster of interesting pixels. The region is essentially a convex hull containing at least a certain number of interesting pixels in close proximity. Finally, the third step runs a compute-intensive algorithm for every pixel in the regions of interest.

This application permits adaptation by exploiting different number of worker machines for its parallel tasks and selecting appropriate sampling schemes for images with different numbers of interesting objects. The resultant executions trade off resource utilization for application performance: allocating more worker machines to a single application may speed up its execution but degrade the overall system performance; and the arrival of images of different characteristics requires different sampling schemes to best utilize the system resources.

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 to available system resources. The alternate configurations can be built out of different application modules or from 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 the different configurations 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* first introduced in [8]. The tunability interface provides language support to express the availability of multiple execution paths for the application, as well as the means by which application progress can be monitored and influenced (by switching to a different path).

3.1. Tunability Interface

In our framework, 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, exported in terms of the tunability interface. In general, the tunability interface of a component provides five kinds of information, which are described below. The sample code fragments correspond to our current implementation of this interface, which is based on eXtensible Markup Language (XML) annotations to C/C++ source.

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. These parameters are usually local variables inside a program module; but are promoted as part of the tunability interface by giving them an external name and listing them in the `control_param` construct. For example,

```
<control_param>
  <param type="int" externname="compress"
    localname="c" onchange="func" />
</control_param>
```

identifies an local variable `c` of type `int` and promote it as a control parameter `compress`. The local variables and control parameters exchange their values when the application execution enters or leaves the corresponding

module. The value of control parameters could also be changed through an external notification from a system scheduler or another application instance. In such cases, the function `func` (as specified by the `onchange` attribute) is invoked given the old value and new value, allowing application-specific actions.

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 component behavior. For instance, a host is characterized by its CPU, memory, and network resources. Execution environments are specified using the `resource_param` construct. For example,

```
<resource_param>
  <host name="client" local="true" />
  <resc type="networkBW" name="net"
    host="client" />
</resource_param>
```

specifies a host `client` with one resource parameter of interest, the network bandwidth (referred to by the name `net`) in the execution environment. The system monitors the specified resources to detect any changes in their characteristics, potentially requiring adaptation of the application components.

3. *QoS metrics* and expressions for evaluating them, which specify the component metrics of interest as well as application-provided means of computing them. Although QoS metrics are application-dependent, they are assumed to be comparable and have min/max values. The construct

```
<QoS_param>
  <qos type="double" dir="dec"
    externname="response_time"
    localname="response" />
</QoS_param>
```

specifies one QoS metric `response_time` of type `double`, mapped to a local variable `response`. Application performance improves whenever the value of this parameter decreases (indicated by `dec`). QoS metrics are evaluated in `QoS_monitor` constructs (not shown) embedded in the source code.

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

```
<component name="module{compress}{...}" >
  <condition> expr </condition>
  other constructs ...
  application source code ...
</component>
```

describes a component with guarded expression `expr` and a name comprising of characters and control parameters. 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. Inside component constructs, other constructs could be defined such as control parameters, execution environment, QoS metrics, and nested components.

5. *Transition functions*, which encapsulate application-specific actions such as updates to local variables and/or sending control messages to other components required upon a change in the component configuration. Compared to the `component` construct that allows adaptation upon module entry, transition functions enable adaptation inside a component. The construct

```
<transition from="oldctl" to="newctl">
  application-specific code ...
</transition>
```

defines a transition point with two sets of control parameters `oldctl` and `newctl`, where the `newctl` parameter identifies the configuration suggested by the system based on the current resource conditions. The application-specific transition function could examine these two sets and decide how to proceed, possibly overriding the systems decision. At the end of executing this function, the control parameters reflect `newctl` values.

The above interface, in the form of source-level annotations, is processed by a preprocessor implemented using an XML parser. The control parameters, QoS metrics, and execution environment definitions are converted into data structures in the source language (e.g., structure/class declarations in C/C++), accessible to other annotation constructs. Different execution paths are achieved by expanding the tunable component 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.

Besides generating the application components required for flexible run-time execution, the preprocessor also generates configuration files that serve as input to both the virtual execution environment (see Section 4 for details) and the external resource scheduler. The configuration files identify 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).

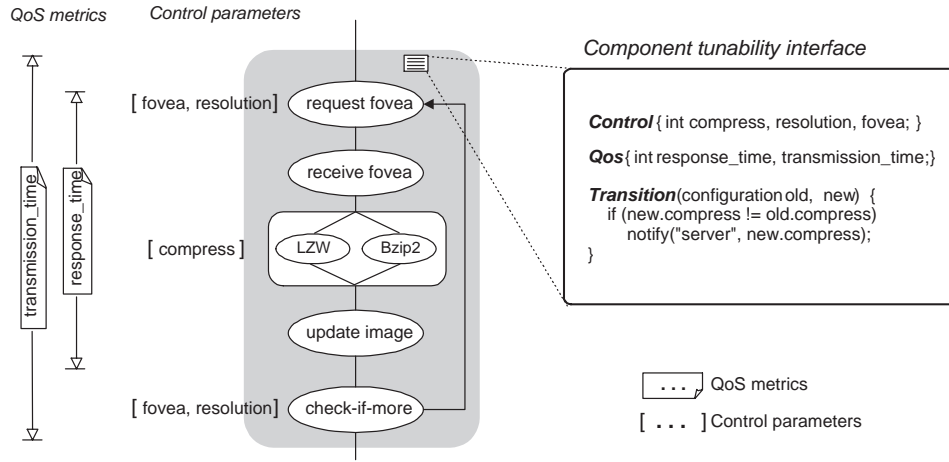


Figure 3. Exporting component tunability interface for client side of Active Visualization. Tunability interface of a component includes control parameters, QoS metrics, and transition functions called upon configuration changes.

3.2. Tunability in Active Visualization

Figure 3 shows the adaptation structure of the client side of the Active Visualization application as exposed using the tunability interface. The progressive refinement component 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. Note that the control parameter resolution is at the same time a QoS metric for this application. 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.

3.3. Tunability in Junction Detection

Figure 4 shows the adaptation structure of the Junction Detection application. The application comprises a single tunable component, responsible for detecting junctions within a single image. The execution of this component is influenced by two control parameters: "workers", the number of worker machines allocated to this application, and "sampling", the sampling scheme selected for a particular image. In addition, the input data characteristics is also viewed as a control parameter "image". The settings of these variables are configured upon component entry. The only QoS metric of interest is "duration", the application execution time. The execution environment for the component describes the set of worker machines on which the parallel tasks run. Although, this description could be arbitrarily detailed including per-host parameters such as CPU load and network bandwidth, we restrict our attention to homogeneous hosts whose

resource conditions can be abstracted in terms of just the number of available workers.

Unlike the Active Visualization application, where control parameter updates involve setting the values of component-local variables, in the Junction Detection application, such updates need to be accompanied by a handshake between the application's steering agent and the underlying Calypso system. This handshake, expressed using the "onchange" function attribute of the control parameters construct, results in the addition or removal of Calypso workers employed for executing the parallel tasks of this component.

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 *virtual execution environments*. These environments run on top of a static distributed system,

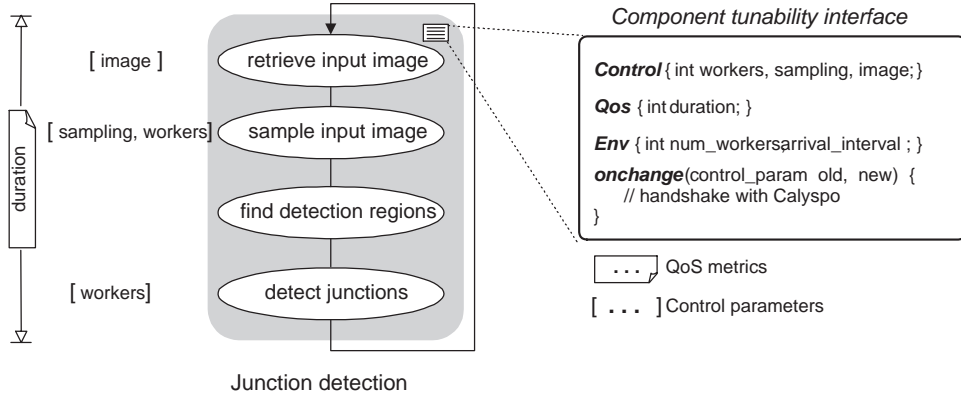


Figure 4. Exporting component tunability interface for Junction Detection.

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 (informally defined as configurations that outperform other configurations under at least one resource situation). These measurements are interpolated to get performance curves that summarize configuration performance.

4.1. Virtual Execution Environment

A virtual execution environment is implemented by effectively creating a “sandbox” around an application, which constrains application utilization of system resources such 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 [1,14]. This code continually monitors application requests for operating system resources and estimates a “progress” metric. Upon detecting that the application has received either more or less than the prescribed for the virtual environment, the injected code actively *controls* future application execution, relying on generally available OS mechanisms, to ensure compliance with specified limits. This technique allows accurate control over resource availability to applications on commodity OSes without modification to either OS or application source code. Its application to three representative

resources—CPU, memory, and network—is described below.

Constraining CPU resources For CPU resources, the sandbox ensures that the application receives a stable, predictable CPU share, as if it were executing on a virtual processor of the equivalent speed. The sandbox uses a monitor process that either starts the application or attaches to it at run time. The monitor process periodically (every 10ms) samples the OS monitoring infrastructure to estimate a progress metric. If other applications take up excessive CPU at the expense of the sandboxed application, the monitor compensates by giving the application a higher share of the CPU than what has been requested. However, if the application’s CPU usage exceeds the prescribed processor share, the monitor would reduce its CPU quantum for a while, until the average utilization drops down to the requested level. On NT, granting and reducing CPU quanta is achieved using fine-grained adjustment of application process priorities. An application process not making sufficient progress is boosted to a high priority level, where it preempts the background processes and occupies the CPU. A process that has exceeded its share is lowered to a low priority level, allowing other processes (possibly running within their own sandboxes) or in their absence, a dedicated background process, to use the CPU.

This approach enables stable control of CPU resources in the 1% to 97% range. Figure 5(a) is a snapshot of the performance monitor display showing three sandboxed applications running on the same host. They start at times t_1 , t_2 , and t_3 , requesting 10%, 30%, and 50% of the CPU share, respectively. With the total CPU load at 90%, all three applications receive a steady CPU share until time t_4 , when the allocation is deliberately perturbed by dragging of a DOS window. This causes the total available CPU to decrease drastically (because of the kernel activity), and a sharp decrease in the immediate CPU shares available to each application. However, this drop is compensated with additional resources once the system reacquires CPU resources (end of window movement). These results indicate that the sandbox can support accurate and stable CPU sharing with resilient compensation. Figure 5(b) compares the execution time of a tight loop on the virtual execution environment (realized on

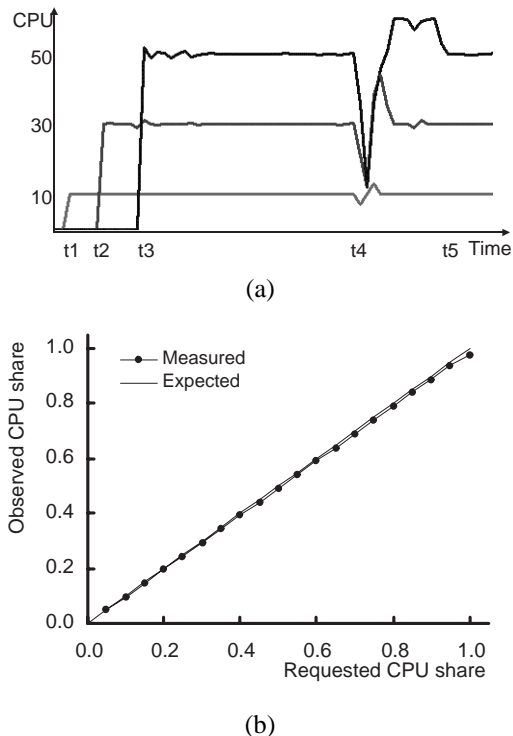


Figure 5. (a) Weighted CPU sharing for multiple applications. (b) Overhead and accuracy of the virtual execution environment with a tight loop.

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.

Constraining memory resources For memory resources, the sandbox ensures that physical memory allocated to the application does not exceed a prescribed threshold. The basic idea is to inject a user-level pager into the application to decide both (a) *when* a page must be swapped out, and (b) *which* page this should be. The sandbox intercepts memory allocation APIs (e.g., `VirtualAlloc` and `HeapAlloc`) to build up its own representation of the process working set. When the allocated pages exceed the desired working set size, the extra pages are marked `NoAccess` and are swapped out automatically by the NT operating system. When such a page is accessed, a protection fault is triggered: the sandbox catches this fault and changes page protection to `ReadWrite`. Note that this might enlarge the working set of the process, in which case a FIFO policy is used to evict a page from the (sandbox-maintained view of the) working set.

Our experiments show that, on a 450 MHz Pentium II machine with 128MB memory, this sandbox implementation can effectively control actual physical memory usage from 1.5MB up to around 100MB. The lower bound marks the minimal memory consumption when the application is loaded, including that by system DLLs. Our current imple-

mentation can be easily extended using similar techniques to more accurately emulate additional effects, such as the impact of paging overhead on application execution.

Constraining network resources For network resources, the sandbox ensures a prescribed sending or receiving bandwidth available to the application. It intercepts network APIs such as `send` and `recv` and makes the application bandwidth compliant by stretching out the interaction over a longer time period (e.g., by using fine-grained sleeps) if the application is seen to exceed its bandwidth threshold.

Our experiments show that the sandbox can effectively constrain bandwidth from 1 Bps to about 11 MBps with an error of less than 2% on Pentium II (450 MHz) machines connected to a 10/100 auto-sensing Fast Ethernet switch, using a ping-pong application that exchanges 4KB-sized messages and that achieves a peak bandwidth of 11 MBps when running outside the sandbox. Figure 6 shows how accurately the sandbox controls network resources for constraints in the range 0.5 KBps to 8 MBps.

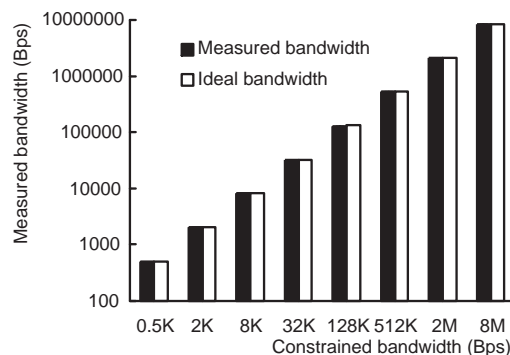


Figure 6. Comparison of application observed network bandwidth and the ideal values under the control of sandbox.

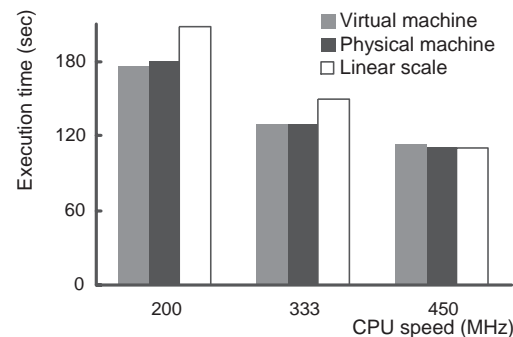


Figure 7. Comparison of application execution times using the virtual execution environment and physical machines for the active visualization application.

Sandboxing Large Applications Figure 7 shows the accuracy of the virtual execution environment in emulating di-

verse resource conditions for large distributed applications 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 these machines. WinBench is a Windows 98/NT benchmark that measures the aggregate impact of CPU speed and memory configuration. 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, indicating the necessity to measure application behavior under different resource conditions.

A detailed description of the techniques used for constructing the virtual execution environment and its capabilities on both Windows NT and Linux is reported in [7].

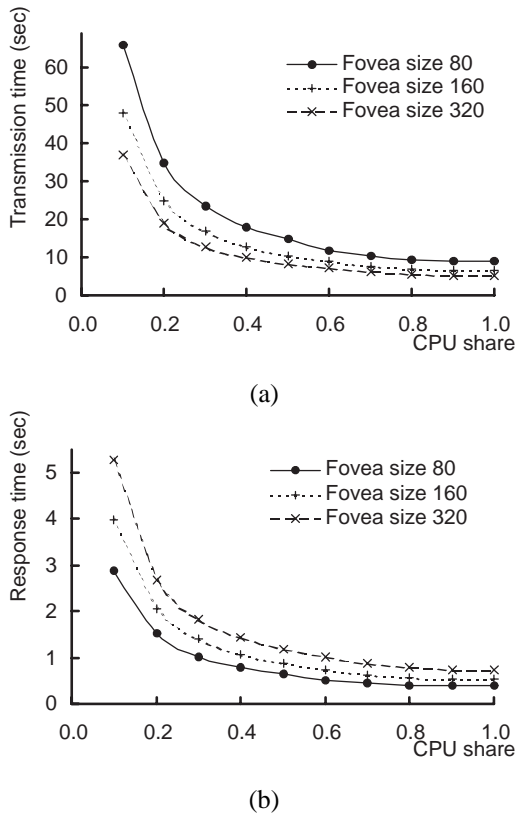


Figure 8. Image transmission time (a) and response time (b) in the Active Visualization application, for different fovea sizes as CPU share varies.

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

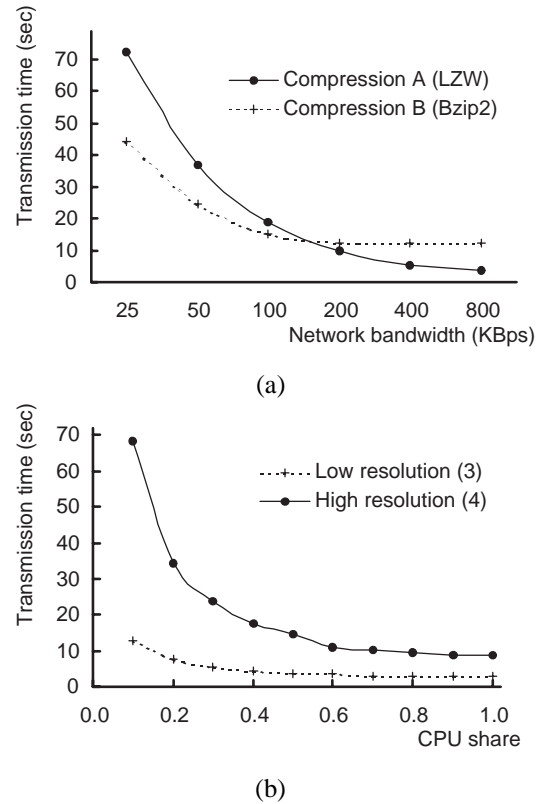


Figure 9. Image transmission time in the Active Visualization application, for (a) different compression methods as network bandwidth varies, and for (b) images of different resolutions as CPU share varies.

of resource conditions. Figures 8 and 9 present a small subset of these measurements and the resulting mappings. The measurements reported here are obtained on a Pentium II 450MHz machine and rely upon manual determination of appropriate resource settings for the testbed.

Figure 8 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 9(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 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

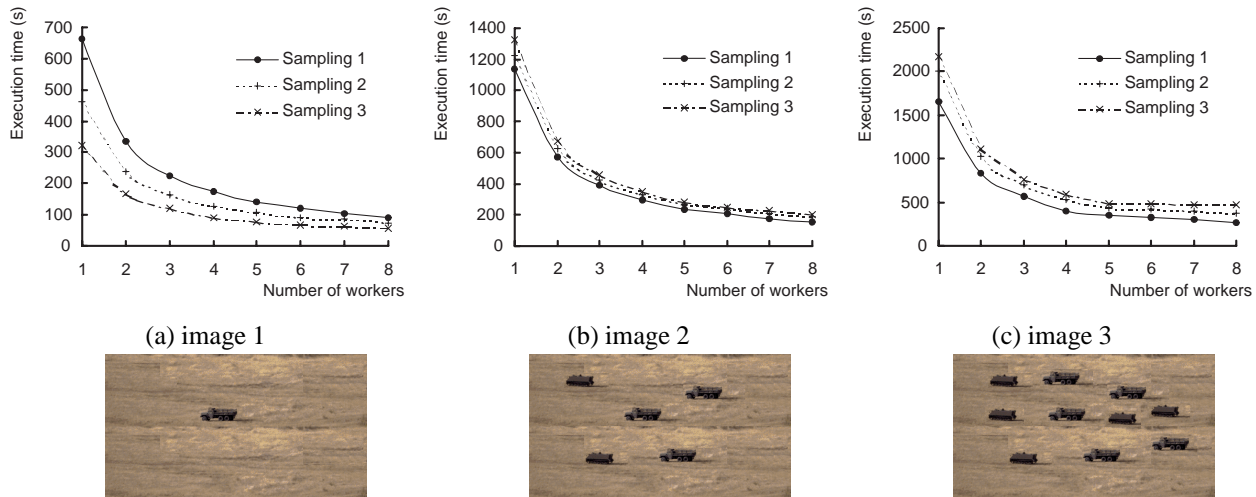


Figure 10. Execution times in the Junction Detection application, for three image classes with of different sampling schemes as the number of machines allocated vary.

transmitted. However compression A performs better when the network bandwidth is high because the CPU becomes the bottleneck then.

Figure 9(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.

4.3. Profiles for Junction Detection

As with the Active Visualization application, a driver program repeatedly executes the Junction Detection application on a cluster of PCs to obtain a mapping between the control parameter values (including the application input) and the output qualities for a range of resource conditions. For simplicity, we restrict our attention to execution environments where the only allowed change in resource conditions is the number of worker machines available to the application. This permits us to summarize the resource conditions in terms of a single parameter, the number of workers, assuming a homogeneous cluster with constant CPU speed and network bandwidth. To model input image characteristics, images are grouped into equivalence classes and representative subsets from these classes are profiled. The measurements reported here are obtained on a cluster of Pentium Pro 200 MHz worker machines connected by a 100 Mbps Ethernet Switch.

Figure 10 shows the execution time for Junction Detection in images belonging to three representative classes for different sampling schemes as the number of worker machines is varied. The three sampling schemes (referred to as Sampling 1, Sampling 2, and Sampling 3 in Figure 10) correspond to the "sampling" control parameter being set to values of 4 pixels, 16 pixels, and 64 pixels respectively. The graphs show that the more artifacts an image has, the

more time it takes to compute all of its junctions. However, different sampling schemes best suit images with different amounts of artifacts. Sampling 1 yields better performance for images with a large amount of artifacts (e.g., image 3), while Sampling 3 yields better performance for images with a small number of artifacts (e.g., image 1). In addition, as expected application execution time decreases as the number of worker machines increases; however, the speedup does not scale linearly attesting to the importance of having measured profiles.

5. Run-time Adaptation

As described earlier in Section 3, the program annotated with tunability interface specifications is converted by a pre-processor into an executable form; the latter includes application modules that make up the different paths as well as steering and monitoring agents 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 11): (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. For simplicity, we assume a relatively restricted form of this func-

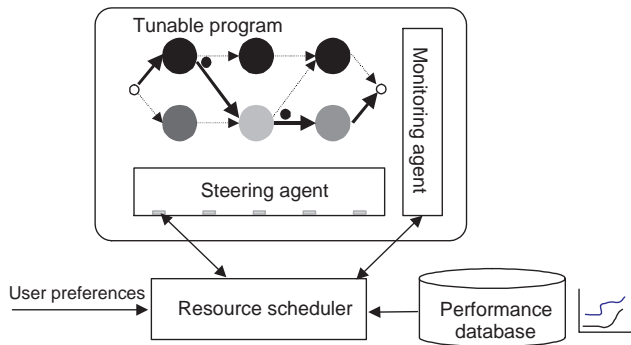


Figure 11. Runtime components (e.g., monitoring agent, steering agent, and resource scheduler) cooperate to enable application-independent adaptation to satisfy user preferences.

tion: maximizing or minimizing a single quality metric. These constraints when considered together with measured resource characteristics, restrict the suitable set of application 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 incurs 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 execution of the application-specific transitional functions in preparation for the activation of the corresponding configuration.

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. Figures 12(a)–(d) show the quality met-

rics achieved by the adaptable form of the application and contrast 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. Note that the steering and monitoring code for the adaptable application are automatically generated from source-level annotations as described in Section 3.

Experiment 1: Adapting compression method to network conditions Figure 12(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 9(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 9(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 to complete. Subsequent image transmissions use compression B and complete in 24 seconds apiece.

Experiment 2: Adapting image resolution to CPU conditions Figure 12(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

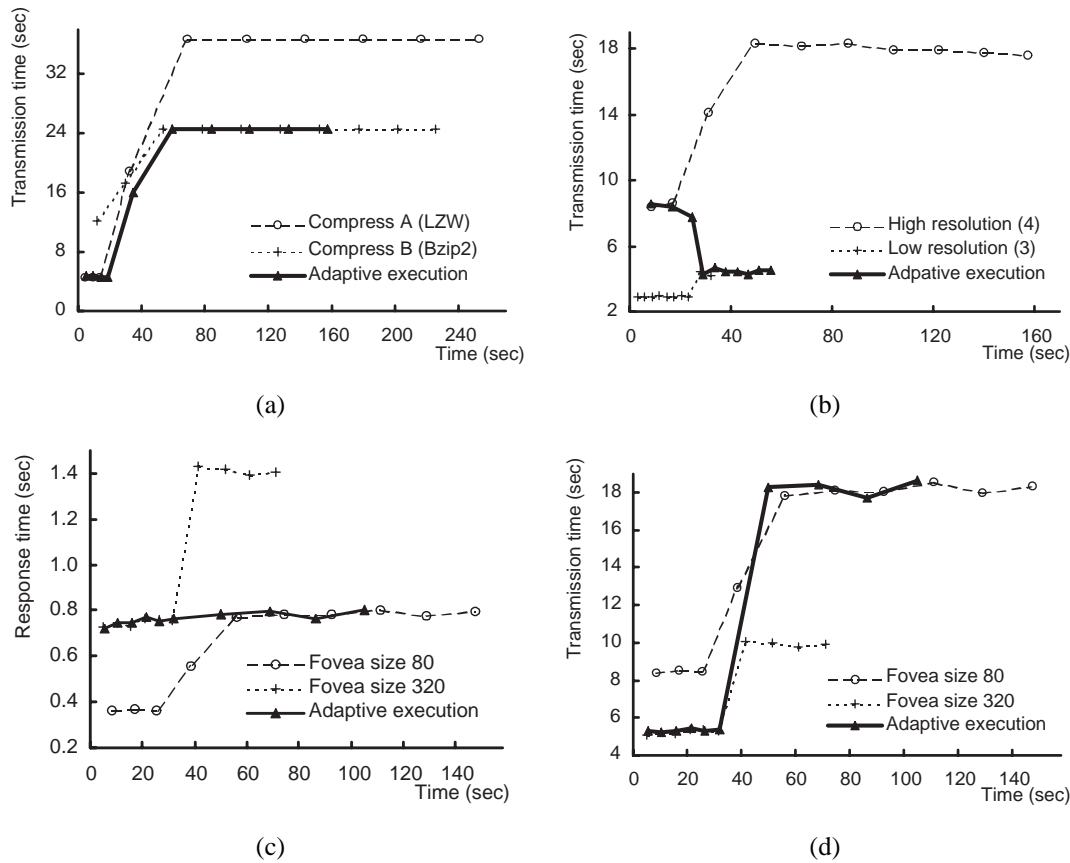


Figure 12. The Active Visualization application adapts to resource variations by: (a) switching compression methods when network bandwidth drops, (b) degrading image resolution as CPU share decreases, (c) and (d) changing fovea size as CPU share changes.

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 9(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 12(c) and 12(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 8. 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.

5.3. Adaptation in Junction Detection

We use an NT implementation of the Calypso system [10] running on a cluster of Pentium Pro 200 MHz machines connected to a 100Mbps Ethernet Switch. The user preferences of interest are to either minimize the execution time for detecting junctions within a single image, or to constrain this

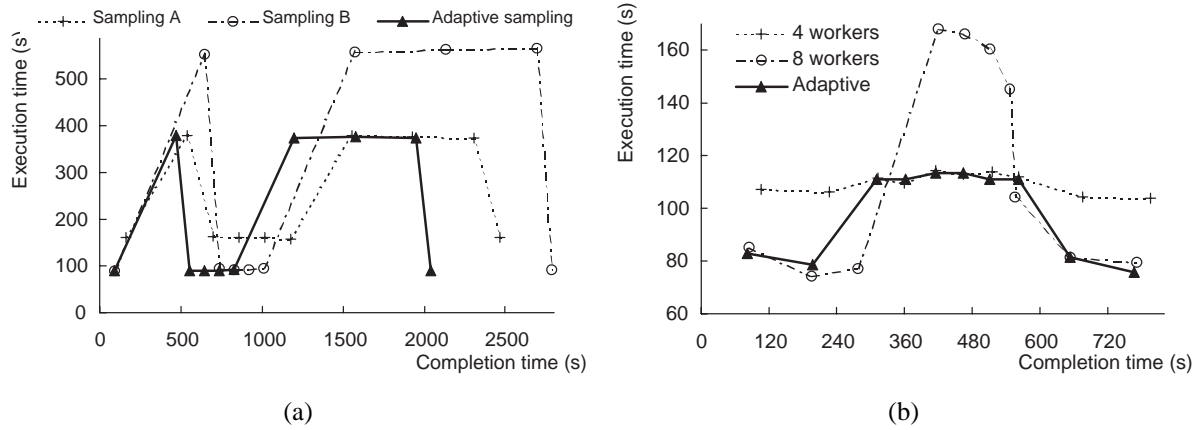


Figure 13. The Junction Detection application satisfies user preferences by: (a) adapting sampling schemes to different classes of input, (b) adapting allocation of worker machines to variation in image arrival intervals.

time to be within a deadline. The system input to the application is a sequence of images belonging to different image classes, which arrive in the beginning or periodically with a specific interval. Over the entire run of the application, this period is assumed to be quasi-static; i.e., it changes only at a coarse granularity. For adaptive execution, the external scheduler associated with the application searches the performance database to find the best sampling scheme and number of workers to process a collection of images. The external environmental changes of concern are the arrivals of images corresponding to different image classes and the variation of the arrival intervals. Both of the changes are assumed to be monitored by an external agent that informs the application through notification messages issued to the steering agent. As described earlier, user preferences are satisfied by reconfiguring the application at run time; the granularity of the reconfiguration is the processing of a single image.

To capture typical usage scenarios of the Junction Detection application, we describe two experiments below that show adaptation of the application to changes in the characteristics and arrival patterns of input images. These experiments carefully isolate the possible adaptation behaviors of the application: the first experiment shows how the application can adapt to an image sequence comprising images belonging to different classes, when the number of workers available for processing a particular image is kept fixed. The second experiment demonstrates application adaptation in response to changes in the image arrival interval assuming that all images belong to the same class; here the application adapts by controlling the number of workers devoted to processing a single image. Figure 13(a)–(b) shows the per-image execution time achieved by the application for a sequence of images. The x-axis of the plots refers to the time when processing of a particular image is complete. As before, the solid line in each plot shows the performance of the adaptable form of the application and the other lines show performance achieved by the corresponding non-adaptable versions. Details of the image sequence and system conditions are provided below.

Experiment 1: Adapting sampling scheme to input classes

Figure 13(a) shows adaptation of sampling schemes in response to arrival of different classes of input images, for a sequence of images whose classes are chosen randomly to be either type image 1 or type image 3 (see Figure 10). We assume that the number of worker machines allocated towards processing a single image is fixed to 4 and that all images arrive at the start of the experiment. The user preference is to minimize the total execution time required for processing the image sequence.

The adaptable version of the application automatically chooses sampling scheme 3 for images of type 1 and sampling scheme 1 for images of type 3, when notified of the class of the arriving image. As a result, it finishes much earlier than non-adaptable versions of the application with fixed sampling schemes, with a 17% and 27% improvement over versions relying upon sampling scheme 1 and sampling scheme 3, respectively.

Experiment 2: Adapting worker allocation to image arrival patterns

Figure 13(b) shows application adaptation by tuning the allocation of worker machines in response to variations in image arrival intervals. The user preference is to constrain the per-image processing time to be within a deadline of 120 seconds while trying to minimize the overall time required for processing the image sequence. The environment change is the image arrival pattern, which is initially set to one every 120 seconds, changes to one every 50 seconds after 200 seconds have elapsed in the experiment, and then reverts back to one every 120 seconds after 500 seconds have elapsed. We use a total number of 8 worker machines and 1 manager machine, permitting the application to use any number of workers from 1 up to 8 for processing a particular image. The experiment uses images of type 1 as input and is set to stop after 800 seconds have elapsed.

Initially when the image arrival interval is 120 seconds, the application scheduler decides to give all the eight worker machines to each image because images do not overlap in their request for resources. When the image arrival interval changes to 50 seconds, the application gets notified by an

external agent and triggers the scheduler to compute a new configuration. As a result, the scheduler decides to give each image 4 worker machines because two images may simultaneously need resources and must each finish within the specified deadline. In order to explain this scheduler decision, note from Figure 10(a) that the lowest execution time for processing images of class one is 100 seconds with 4 workers and 60 seconds with 8 workers and that the scheduler takes into consideration extra time needed to start and control Calypso GUI interface. Similarly, the scheduler reverts back to allocating all machines to each image when the arrival interval changes back to 120 seconds. To compare how user preferences are satisfied with non-adaptive executions, we run the same image sequence again with two worker allocation schemes, which are fixed at using 8 machines or 4 machines for each image respectively. The results show that with the first allocation scheme, some images miss their deadlines when the arrival interval is 50 seconds because at least two images (in the worst case four images) are time-sharing the worker machines at the same time, interfering each other's execution. For the second allocation scheme, all the images finish by the deadline (the slightly longer execution time in the middle of the curve is due to the sharing of the manager machine and network resources). However, the total execution time is 10% longer than with the adaptive execution.

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 [28,24] 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 [23] 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 [21,13] 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 [16] proposes build-

ing 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 [19] and AppLeS [3,26] 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 [25] 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 case studies how such structuring of distributed and parallel 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, analyzing application behavior for further automating the profile modeling process, and providing support for transparently endowing application binaries with adaptation capabilities.

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 grant number CCR-9988176; 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.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, 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. Dasgupta. Parallel processing with Windows NT networks. In *Proc. USENIX Windows NT Workshop*, Aug. 1997.
- [11] P. Dasgupta, Z. Kedem, and M. Rabin. Parallel processing on networks of workstations: Fault-tolerant high performance approach. In *Proc. 15th IEEE Intl. Conf. on Distributed Computing Systems*, 1995.
- [12] 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.
- [13] T. Gross, P. Steenkiste, and J. Subhlok. Adaptive distributed applications on heterogeneous networks. In *Proc. 8th Heterogeneous Computing Workshop (HCW '99)*, Apr. 1999.
- [14] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proc. 3rd USENIX Windows NT Symposium*, Jul. 1999.
- [15] H. Ishikawa and D. Geiger. Segmentation by grouping junctions. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 1998.
- [16] J. Itoh, R. Lea, and Y. Yokote. Using meta-objects to support optimization in the Apertos operating system. In *Proc. USENIX Conference on Object-Oriented Technologies (COOTS'95)*, Jun. 1995.
- [17] M. Jones and J. Regehr. CPU reservations and time constraints: Implementation experience on Windows NT. In *Proc. 3rd USENIX Windows NT Symposium*, Jul. 1999.
- [18] Z. Kedem, K. Palem, and P. Spirakis. Efficient robust parallel computations. In *Proc. 22nd ACM Symp. on Theory of Computing*, 1990.
- [19] P. Keleher, J. Hollingsworth, and D. Perkovic. Exploiting application alternatives. In *Proc. 19th Intl. Conf. on Distributed Computing Systems*, Jun. 1999.
- [20] C. Lee, J. Lehoczy, 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.
- [21] 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.
- [22] 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.
- [23] P. Chandra et al. Darwin: Customizable resource management for value-added network services. In *Proc. Sixth IEEE Intl. Conf. on Network Protocols*, 1998.
- [24] 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.
- [25] M. Shankar, M. DeMiguel, and J. Liu. An end-to-end QoS management architecture. In *Proc. Real-Time Applications Symposium*, Jun. 1999.
- [26] N. Spring and R. Wolski. Application level scheduling of gene sequence comparison on metacomputers. In *Proc. 12th ACM Intl. Conf. on Supercomputing*, Australia, 1998.
- [27] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource ReSerVation Protocol. *IEEE Network*, Sep. 1993.
- [28] J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, Jan. 1997.