Dynamically Negotiated Resource Management for Data Intensive Application Suites

Gary Nutt*, Scott Brandt, Adam Griff, and Sam Siewert
Department of Computer Science, CB 430
University of Colorado
Boulder, CO 80309-0430

Marty Humphrey
Department of Computer Science and Engineering
University of Colorado at Denver
Denver, CO 80217

Toby Berk
School of Computer Science
Florida International University
Miami, FL 33159

Abstract

In contemporary computers, and networks of computers, various application domains are making increasing demands on the system to move data from one place to another, particularly under some form of soft real-time constraint. A brute force technique for implementing applications in this type of domain demands excessive system resources, even though the actual requirements by different parts of the application vary according to the way it is being used at the moment. A more sophisticated approach is to provide applications with the ability to dynamically adjust resource requirements according to their precise needs as well as the availability of system resources. This paper describes a set of principles for designing systems to provide support for soft real-time applications using dynamic negotiation. Next, the execution level abstraction is introduced as a specific mechanism for implementing the principles. The utility of the principles and the execution level abstraction is then shown in the design of three resource managers that facilitate dynamic application adaptation: Gryphon, EPA/RT-PCIP, and the DQM architectures.

*Authors Nutt, Brandt, and Griff were partially supported by NSF Grant No. IRI-9307619

1 Introduction

There is an emerging class of application programs, stimulated by the rapid evolution of computer hardware and networks. These distributed applications are data intensive, requiring that diverse data types (beyond the traditional numerical and character types) such as audio and video streams be moved from one computer to another. Because of the time-sensitive nature of moving stream data (and because of the similarity of the target applications with traditional hard real-time applications), these applications are often referred to as *soft real-time* applications. In contrast to hard real-time systems, not every deadline must be met for these applications to be considered a success (although most deadlines should be met).

A distributed virtual environment (DVE) is one example of soft real-time applications: a DVE supports a logical world containing various shared entities; users interact with the entities in the world using a multimedia workstation. DVEs are data-intensive, since shared information must be disseminated throughout the network of user machines. There are many other classes of applications that exhibit similar data movement characteristics, including multimedia systems, image-based information systems, image processing systems, video conferencing, and virtual reality.

The purpose of an operating system is to manage the resources and facilities used by processes, which distribute the data among the objects. Traditionally, general purpose operating systems are designed with built-in policies; such resource managers provide their *best effort* to satisfy all resource requests according to a relatively inflexible, but generally fair, policy. Best-effort resource management policies typically do not include support for deadline management, particularly where the precise nature of the deadline management depends on the application semantics.

In soft real-time applications, the importance of any specific data movement task depends on the importance of the two entities involved in the transfer. At one moment, it may be very important to the system's user to move data from location A to location B because the user is directing the computer to perform some task depending on A and B, whereas a few minutes later, interactions between A and B may be far less important since the user has shifted the focus to another set of computational entities. For example, suppose that an object containing a video clip is shared between two users, Joe and Betty, on two different workstations interconnected by a network; while Joe and Betty are discussing the video clip, it might be important that both workstations have a high fidelity moving image on their

screens. As the video continues to run, suppose Joe and Betty shift their attention to a different set of objects; it is no longer necessary to support the data transfer rate required for high fidelity video, particularly if the maintaining the data transfer for the video clip uses resources that are needed for Joe and Betty's new activity. Resource management policies that support such applications are sometimes called user-centric [13]. These types of policies stress that when the system cannot satisfy all of the resource requests of all running processes, the resources should be allocated to best satisfy the desires of the user.

For the past few years we have focused on resource management techniques for supporting soft real-time data movement. Early experimentation indicated that the operating system performance was inadequate to support this type of data movement. However, it was observed that the effect of resource bottlenecks could be minimized if the OS employed an allocation policy in which resources were directed at applications (or the parts of the application) that needed them at the moment, yet which could be changed as the user's activities changed — a resource management policy was needed that was sensitive to the dynamic needs of the users in the context of their applications. Therefore, the effort was focused on defining and experimenting with ways for the system to provide more effective support for these applications (as originally reported in the conference paper from which this paper is derived [21]).

There are two primary contributions in this paper. The first is the identification of a set of three principles to guide the development of soft real-time applications, along with the explanation of a mechanism — *execution levels* — to implement the three principles. The principles are based on the requirements of data-intensive, soft real-time applications that may collectively require more resources than are available. These principles define guidelines for programmers to develop soft real-time applications, and serve as requirements for a system that can provide the accompanying support. Informally, the execution level abstraction maps resource usage to goodness, and defines a unique mechanism both to design applications and to manage real-time performance. The execution level abstraction is one concrete mechanism for implementing software that embraces the principles.

The second contribution of the paper is the presentation and evaluation of three specific aspects of soft real-time support that illustrate the utility of the principles and execution levels:

- Many contemporary applications in the target domain are written as object-oriented programs,

requiring support for distributed objects. Object management policies are crucial to the overall data movement performance. The Gryphon distributed object system provides a means by which applications can influence the system's object placement, caching, and consistency policies [10]. Gryphon uses execution levels to tradeoff shared object consistency versus network bandwidth. Analysis and experiments show that by managing the policies to match the application strategy, remote object reference performance can be improved by several orders of magnitude.

- Applications frequently need to move large amounts of stream data from one device to another, e.g., from the network to a display. As data moves between devices, it sometimes needs to be filtered (e.g., to compress / decompress the data, or to extrapolate missing sections). The Real-Time Parametrically-Controlled In-kernel Pipe (RT-PCIP) mechanism provides a means to insert / remove kernel-level filters used when moving data between devices. The Execution Performance Agent (EPA) employs another form of execution levels, where a level is determined by the confidence and reliability of the application's pipeline service request estimates. Once the EPA has selected a level of execution (based on the application service estimates), it configures the pipeline, then controls the RT-PCIP operation by adjusting filter priorities. The EPA / RT-PCIP facility provides a form of soft real-time control not previously available using in-kernel pipe mechanisms.

- Generic soft real-time applications need certain resource levels in order to meet their deadlines. Such applications can be written to dynamically modify their processing according to the availability of resources such as CPU, network bandwidth, etc. The Dynamic Quality of Service Resource Manager (DQM) uses execution levels to allow applications to dynamically negotiate for CPU allocation. As a result, these applications can implement a broad range of soft real-time strategies not possible in other systems.

Section 2 explains the system design principles and shows how execution levels provide a mechanism for implementing them. Section 3 introduces the Gryphon distributed object system, explains how it uses execution levels, then discusses the performance of the approach. Section 4 presents the EPA / RT-PCIP mechanism and shows how it allows applications to provide one approach to soft real-time to control in-kernel filter mechanisms. Section 5 explains how the DQM mechanism uses execution levels, then discusses several aspects of its behavior. Section 6 is the summary and conclusion.

2 Design Principles

Various researchers have addressed different aspects of soft real-time support (e.g., see [6, 7, 8, 9, 12, 14, 17, 19, 23, 25]). One problem with many of these studies is in the inherent definition of the term "soft real-time:" It can mean that applications almost always meet deadlines for computation, that priorities can be elevated if the deadline miss-rate is too high, that an application's period can be lengthened if it misses too many deadlines, etc. One conclusion to draw from this diversity of perspectives is that all of the characteristics are important in one context or another. Unfortunately, it would be difficult to design an OS to behave properly on an case-by-case basis. It then follows that one might consider a meta approach in which there is a framework for the way applications make their requirements known to the OS; the responsibility for casting the specific soft real-time requirements into the framework is then the responsibility of the application designer. This study is based on that meta approach. It is best explained by considering a set of underlying principles that have been derived from studying various soft real-time applications, particularly a DVE, and by carefully considering the types of OS support these applications require. This section explains the rationale for the principles, the principles themselves, and the execution level realization of the principles. The remainder of the paper shows how the principles have been applied to three different aspects of system support.

2.1 Motivation for the Principles

The target applications have substantial data movement between units, where a unit is an application, an object, a compound object such as one intended to represent a person in a DVE, a thread, etc. In these applications, any unit needs to be able to change the importance of its interactions with other units according to information that can only be known at runtime (e.g. the focus of the user's attention). Based on this information, the relative importance of the units can be changed dynamically to reflect the best interests of the users.

As a representative of the target class of applications, we built a prototype *virtual planning room* (VPR) [22]. The VPR is a multiperson DVE supporting free-form communication in a manner similar to electronic meeting rooms and other distributed virtual environments. A VPR world is a collection of objects, with VRML representations and behaviors of varying complexity. A compound object representing a human participant (an "avatar") becomes a part of the world when the user enters the VPR.

The basic role of the VPR is to provide real-time audio and video support across the network, to render objects on each user's screen (according to the user's avatar orientation), and to provide an environment in which one can add domain-specific extensions. The VPR is a client-server system where each person uses a workstation to implement the human-computer interface. Hence, the client machine must render each visible artifact from the VRML description of appropriate objects in the world, and cause behaviors (such as modifications to objects) to be reflected in all other appropriate clients. Once the VPR had been developed, it was used to begin exploring system software design and organization that might be well-suited to soft real-time applications.

Programmers in such an environment quickly learn to design objects so that they use different strategies for performing work, according to the perceived importance of that work to the user:

- If a graphic object or video stream is not the focus of the user's attention (i.e., the user has not oriented the avatar's eye directly at the video stream object), considerable system resources will be used to render the image(s) when the user does not really care about it.
- If three people are using the VPR and two of them are engaged in high frequency manipulation of a complex, shared object, the third person probably does not want to use inordinate workstation resources tracking minor changes to the shared object.
- If a user is engaged with a video stream for which the system is unable to deliver a full 24 frames per second, the user will often choose to run the video playback at 12 frames per second rather than having it fluctuate between 16 and 24 frames per second.
- If the network bandwidth into a workstation is relatively underutilized and local resources are oversubscribed, then local resources are momentarily more important than network bandwidth and the processing should be changed accordingly. For example, application throughput (and therefore end user satisfaction) would be improved if the workstation received an uncompressed data stream from a remote site rather than decompressing the stream locally as it is received.

2.2 System Design Principles

Based on the experience with the VPR and various underlying systems, we developed a relatively straightforward set of principles to direct our ongoing system research. Though the principles are sim-

ple, they highlight characteristics of the soft real-time application domain that most current operating systems do not address.

Soft Real-Time. Many of the aspects of the target application domain involve *periodic computation* in which some processing must be done repeatedly and according to a regularly occurring deadline. For example, display frame update must occur at least 24 times per second (many systems are designed to support 30 frames per second). However, users are often able to tolerate certain failures to meet the deadline, especially if it means that another aspect of the user's work will receive higher quality service. The failure mode (*softening* of the deadline requirement) can vary according to the exact nature of the application mix: Occasional missed deadlines are acceptable, provided they are not regular; consistently missing the deadline by a small amount of time may be acceptable; the application may be able to scale back its service time requirement to make it possible for the system to make the deadline; some other application may be able to scale back its resource usage so that all units meet their deadlines; etc.

> **Principle 1:** Resource managers should support diverse definitions of application failure and provide sufficient mechanisms to react to missed deadlines.

Application Knowledge. In an oversubscribed, multithreaded system, the resource managers must block some threads while they allocate their resources to others. A best effort resource manager has a built-in policy to guide the way it allocates resources. In an environment such as a DVE, the relative importance of each thread changes according to the nature of the relevant objects and the attention of the user. The resource manager needs additional information to perceive the situation — information that is only available at runtime (via the application).

> **Principle 2:** Applications should provide more information than a single number (e.g., priority) to represent their resource utilization needs, and the resource managers should be designed to use this knowledge to influence the allocation strategy.

Dynamic Negotiation. In a conventional environment, when an application makes a request for resources, it assumes that the resources will be available and that the application will, in turn, provide its best service. In a more flexible environment, the application might request an amount of resource, K_0,

that the resource manager is unable to satisfy. The resource manager could respond to the application by saying that it can offer K_1 units of the resource based on periodic usage — a form of admission from hard real-time or quality of service (QoS) technologies. The application could respond by saying that it would be willing to change, say, its period, and would need K_2 units of the resource, etc. That is, the application and the resource manager potentially enter a *negotiation* whenever the application asks for resources, or for an assurance of resource availability in the case of periodic computing. The nature of soft real-time computing makes it very difficult for the application to provide an "optimized" request, since it does not know the state of the system's resources. In a hard real-time system (and in many soft real-time systems), the request is made using the worst case estimate; unfortunately, worst case requests tend to tie up resources during times when they are not really needed, aggravating the oversubscription problem.

Real-time systems determine their behavior at admission time by requiring that each application unequivocally determine the maximum amount of resource that it will ever need. If a hard real-time system supports dynamic admission, it must analyze each new request in the context of all extant resource commitments. In the target domain, applications may enter and leave the system at any time, and threads / objects may frequently change their resource needs: This suggests that resource requests should change, and that the nature of each individual negotiation might change whenever any unit makes a resource request.

> **Principle 3:** The resource management interface should be designed so that the level of allocation can be negotiated between the two parties, with negotiation initiated by either party at any time.

2.3 Execution Levels: A Mechanism for Dynamic Negotiation

These principles discuss some responsibilities of soft real-time applications and resource managers and the communication interface between them. The first principle focuses on application behavior — what the application should do to address different forms of soft real-time, and how it should deal with missed deadlines. The second principle addresses the interface for the application and resource manager to interact with one another. The third principle is concerned with the resource manager's obligation in the negotiated policy.

The principles suggest that a resource management philosophy is needed where the application can assume part of the responsibility for the resource allocation strategy (Principles 1 and 2), yet which fits within the general framework of a multiprogrammed operating system. This requires that the nature of the application-system programming interface be enhanced to address the principles. Others have also recognized that this kind of shift in the interface could substantially improve overall system performance for "single-application, multiprogrammed" domains e.g., see [20, 23, 25].

Principle 3 suggests that there is a framework in which applications and resource managers can pose resource allocation scenarios to one another. This can be accomplished by providing a *language* for interaction, then ensuring that the two parties are prepared to interact with one another.

The principles can be realized using a software abstraction called *execution levels*. An application's execution levels are defined during the design and implementation of the software for the application. Execution levels do not provide a mechanism for designing or selecting a soft real-time strategy; this is still the responsibility of the application developer. However, once a strategy is selected, execution levels allow an application to specify the resource requirements that are consistent with the strategy that it implements.

A set of execution levels represents varying strata of resource allocation under which an application is able to operate. In the simplest case, the execution levels for an application are:

$$< Level_i, R_{1,i}, R_{2,i}, ..., R_{m,i} >$$

for $1 \leq i \leq n$ different levels and m different resource types. The application unit can provide its highest quality service if it is able to acquire $R_{j,1}$ units of the j resource types. The application writer can also design a first alternative strategy that uses $R_{j,2}$ units of the j resource types, providing degraded service, e.g., graphics figures may not be rendered as well, a period may need to be longer, or the frame update rate may be lower.

Applications written to run in oversubscribed environments commonly use a form of execution levels as a matter of course (without any system support). For example, graphics programs frequently use "wireframes" to represent geometric solids during certain phases of graphic editing. Table 1 shows a set of execution levels related to this kind of graphic programming technique used in the VPR (the table represents the amount of CPU time used for various rendering options in the VPR). A simple moving object changes its processing time over a 4:1 range in 12 execution levels by varying only 3

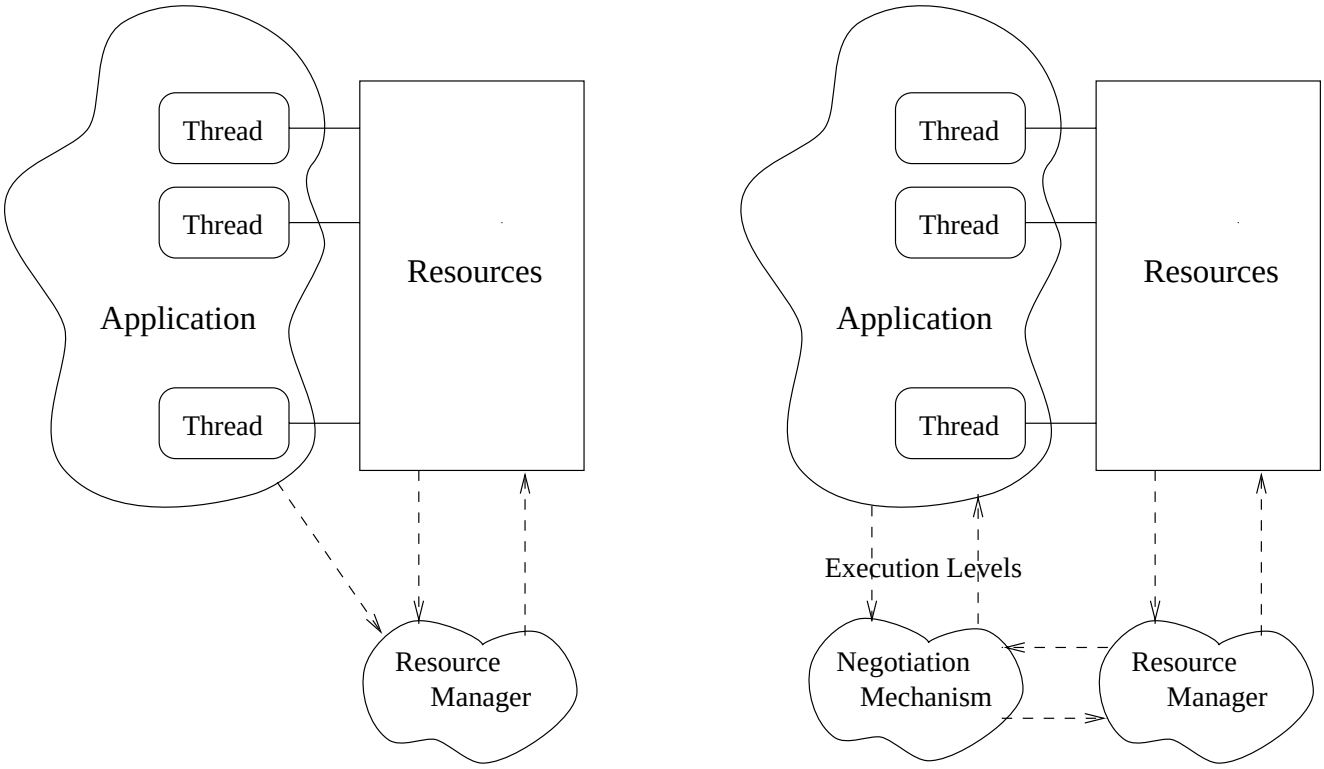
Rendering	Lights	Polygons	Frames per second	% of Max
smooth	1	2X	3.19	100.0%
flat	1	2X	3.34	95.5%
wireframe	1	2X	4.45	71.7%
smooth	0	2X	4.76	67.0%
flat	1	2X	5.15	61.9%
smooth	1	1X	5.87	54.3%
flat	1	1X	6.09	52.4%
wireframe	0	2X	7.70	41.4%
smooth	0	1X	7.97	40.0%
flat	0	1X	8.63	37.0%
wireframe	1	1X	8.94	35.7%
wireframe	0	1X	12.74	25.0%

Table 1: Varying Resource Usage in the VPR

parameters: rendering mode (wireframe, flat shading, or smooth shading), number of specific light sources (0 or 1), and number of polygons (those marked 2X used twice as many polygons as those marked 1X). The table shows frames per second generated and CPU time used as a percentage of the highest level. The OpenGL Performance Characterization Organization [24] has similar performance measurements showing applications that exhibit 10 different execution levels with CPU requirements varying by as much as a factor of 10. See [11] for further justification for using execution levels in applications.

Execution levels define a total order over a resource vector for all system resource types. While this is the underlying theory for the approach, none of the projects described in this paper currently address more than one resource type. Thus, it is relatively easy to define the total order so that as the level increases (i.e., the quality of the solution decreases), the resource requirements also decrease; this is the fundamental constraint the approach makes on the application in defining its notion of soft real-time. Even though the resource requirement versus level is monotonic, in practical applications it would not normally be linear. As the level increases (the quality of the application's service decreases), there is usually some point above which the application provides no service. For example, once the video frame rate of a streaming video application falls below 5 frames per second, its quality is effectively zero.

Execution levels are a mechanism that enables applications and resource managers to implement soft real-time consistent with the principles described above. Figure 1(a) represents the conventional



(a) Conventional

(b) Execution Levels

Figure 1: Execution Level API for Resource Manager

relationship among threads in an application and the resources they use (the undirected, solid lines). The dashed lines in the figure represent control flow among the resource, the resource manager, and the application. In best effort approaches, the resource management policy is built into the resource manager at the time it is designed; there is no interaction between the policy module and the set of applications.

Figure 1(b) shows a new framework with a logical component, called a *negotiation mechanism*, that interacts with the application using execution levels. Conceptually, the negotiation mechanism appears as a conventional application to resource managers, and the negotiation mechanism appears as a resource manager capable of dynamic negotiation to the applications. In the framework, applications are able to create their own tactics for defining and managing soft real-time, then for expressing their resource needs to a resource manager (the negotiation mechanism) using execution levels; this supports Principles 1 and 2. The negotiation mechanism is an extension of the work done by conventional

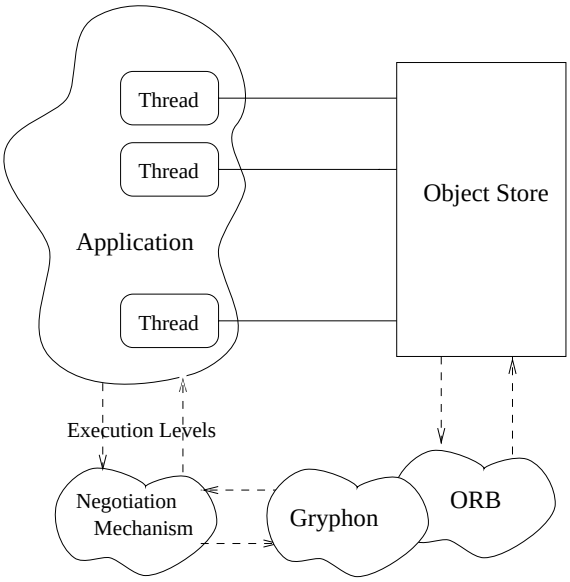


Figure 2: An ORB with Gryphon

resource managers — it provides a module to do dynamic negotiation; this supports Principles 2 and 3.

Execution levels are a sufficient language for supporting dynamic negotiation, though the problem now shifts to how the application and negotiation modules should be designed. The distributed object manager study, the real-time pipe control study, and the dynamic QoS manager study all employ different approaches for designing and implementing these modules. In the remainder of the paper we consider each of these studies in detail.

3 The Gryphon Distributed Object Manager

Network bandwidth is the limited resource being managed and shared in this aspect of the work. For object oriented systems, it is natural to study distributed object managers as a means of addressing network bandwidth performance (e.g., see [15, 16, 18, 26]). The Gryphon is an enhancement to a conventional distributed object manager (such as a CORBA ORB). The purpose of the Gryphon is to support dynamic negotiation of object placement policies according to the needs of the application and the state of the system resources. Figure 2 describes the general architecture of the Gryphon approach (in the context of Figure 1). Each application uses the CORBA IDL interface for normal object references, and a supplementary language for describing the object placement and caching policy preferred

by the application; the supplementary parts of the language are called policy *hints*. A set of hints constitutes one execution level, e.g., the distributed view of the object may use a strong consistency model at one level and a weaker form of consistency at a lower level. The negotiation mechanism defines the execution level order for the application, then uses that definition in directing the Gryphon according to observed performance of the system. If the negotiation mechanism detects a shortage of network bandwidth, it lowers the execution level of the applications to free up bandwidth. The Gryphon has been designed to analyze the hint information it receives from all applications, then to set policies in the ORB.

The Gryphon system supports the fundamental principles described in Section 2 as follows:

Principle 1: Soft Real-Time Using execution levels an application can have variable object coherence, object placement, and various timeliness updates policies.

Principle 2: Application Knowledge The application supplies information regarding location and caching (for each level) in the form of per user, per object update granularity and consistency requirements.

Principle 3: Dynamic Negotiation The application dynamically modifies the object coherence, placement, and timeliness policies. For example, users change their focus from one set of objects to another.

3.1 Representing Execution Levels

Soft real-time applications can have a wide variety of object reference patterns. For example each of the following scenarios represent one recurring class of VPR applications (there are also other types, though these three will illustrate the approach). Each of these scenarios generates a radically different set of requirements on the object manager:

Scenario A: A Learning Laboratory The DVE is used as a laboratory in which a student or small group of students can conduct various experiments. A student may browse through different experiments without communicating with other students, or join a group to work with other people. The laboratory has a number of static objects with complex VRML specifications, e.g., lab apparatus and documentation. Avatars move infrequently, but most other objects do not move at all.

Scenario B: Collaboratively Flying an Unoccupied Air Vehicle Siewert built an unoccupied air vehicle, FLOATERS, to test various parts of the EPA / RT-PCIP work (see Section 4). The VPR has been used to "fly" FLOATERS, i.e., one can navigate FLOATERS by manipulating its virtualization in the VPR. This scenario is for a group of people to navigate FLOATERS as collaborative work from within the VPR. Here avatars are in the virtual space together and they can see each other and other objects in the room.

Scenario C: A Weather Modeling Application Weather modeling is highly data and computation intensive, with the end result being weather information displayed in a DVE. Weather data is partitioned into small regional subsets, then intense processing is performed on each partition. After the first phase of processing is complete, data at the fringes of the subsets are distributed to other processes and then computation continues.

If any of these scenarios are implemented in a system like the VPR, objects will be shared across many workstations, causing substantial network traffic. Conventional distributed object managers provide location transparency, though the experience with the VPR shows that the distributed components needed to have substantial influence over object location policies (without this flexibility, the applications could not make performance tradeoffs based on access demands).

There are several techniques that can reduce network traffic due to remote object reference:

Placement If an object is stored on host X and is frequently referenced (only) from host Y, the traffic and delay to the application could be reduced by storing the object at host Y.

Caching If an object is stored on host X, and is frequently read from hosts Y and Z, then keeping copies of the object at hosts Y and Z can reduce network traffic.

Consistency If an object is stored at host X and is rapidly being changed by arbitrary hosts but is being read by host Y, traffic and processing overhead at host Y can be reduced by allowing Y to keep an out-of-date copy of the object (as opposed to updating Y's copy each time X's copy changes).

The Gryphon approach is based on the idea that the applications are the only component capable of choosing among these techniques, since application behavior is a critical factor in the benefit of each technique. Table 2 represents the relationship between the application hints and levels for Scenarios

Level	Hints	Semantics
1	(none)	Centralized location Strong consistency
2	(none)	Distributed location No caching
3	Location	Best location No caching
4	Location Cache strong	Best location Strong consistency
5	Location Cache sequential	Best location Sequential consistency
6	Location App directed caching	Best location App directed caching
7-N	Location Caching Update frequency	Best location Some updates not propagated

Table 2: Execution Levels for Scenarios A and B

A and B (which differ primarily in number of objects and their movement). An application implementing Scenario A or B would use decreasing amounts of network bandwidth with decreasing level (higher level number). In these scenarios, an application would be inspired to run at a lower (higher-numbered) level if some of its components were missing soft deadlines. That is, levels 1-6 all produce the same behavior, though the application has to provide more information in level i than it does in level i-1, and the overall system benefits due to reduced network traffic. The distinction between levels 1 and 2 is related to using a single object storage location versus distributing the objects to multiple storage servers; while this would be an unusual application option, it is included to emphasize that centralized servers cause higher network traffic. In levels 7 to N, the application's fidelity erodes since at level 7, a host machine allows changing objects to become inconsistent due to lack of update propagation. (The differences among these levels is in the number of updates a host is willing to miss.)

Scenario C (Table 3) has a different set of data reference patterns than do the other two scenarios (and consequently it can operate under a different set of resource allocation criteria than Scenarios A and B). Data are distributed to host machines that perform localized computation — this is the "best location, strong consistency" case. Again, lower (higher-numbered) levels represent situations that use less network bandwidth, meaning that the application must do more work to achieve the same result, but that there will be more bandwidth available to the system.

Level	Hints	Semantics
1	(none)	Central location Strong consistency
2	(none)	Random distributed location Strong consistency
3	Location	Best location Sequential consistency
4	Location	Best location No caching
5	Location App directed caching	Best location App directed caching
6-N	Location Caching	Best location App directed caching Different algorithms

Table 3: Execution Levels for Scenario C

3.2 Performance Analysis

To analyze the performance of a Gryphon system implementation, models based on the scenarios (and others not discussed here [10]) were used to characterize traffic patterns resulting from different object managers. In the VPR, object state changes when the object moves (it may also change due to other behaviors, though this simplification is sufficient for this analysis). Assuming that a single message is used to move an object, and that all messages are small and fit into one network data packet, Table 4 shows the parameters to characterize message traffic, and Table 5 shows the values to represent the three scenarios. Scenario A is notable for its large number of objects (S = 10,000) with many of them moving (N = 1,000); there are also many processes using objects (L = 1,000) and many VPR processes (V = 1,000); finally, video fidelity is required to be very good in this scenario (G = 24). Scenario B has many of the same characteristics as Scenario A, except that the number of objects is greatly reduced (S = 100, N = 20, L = 20, and V = 20). Scenario C represents a different kind of application where there are many moving objects (N = 10,000), many objects being updated at a time (M = 1,000), and relatively high update rates (U = 1,000); however, the frame update rate is zero.

These parameters are used to derive equations for three metrics:

$T_{\mathbf{VPR}}$ Amount of network traffic to all VPR processes in messages per second

$T_{\mathbf{app}}$ Amount of network traffic to all non-VPR processes in messages per second

N	Number of moving objects
M	Number of objects being modified at each process
U	Update rate for each of the moving objects
L	Number of processes using the object
V	Number of VPR processes
S	Number of static (not moving) objects
F	Update rate of display frames
R	Ratio of updates that get propagated

Table 4: Parameters used to Model Network Traffic

Scenario	N	M	U	L	V	S	F	R
A: Art Museum	1,000	1	2	1,000	1,000	10,000	24	0.01
B: FLOATERS	20	1	2	20	20	100	24	0.5
C: Weather Modeling	10,000	1,000	1,000	10	0	0	0	0.0001

Table 5: Characteristics for Scenarios used to Evaluate the Gryphon System

$T_{\textbf{total}}$ Total traffic in the network in messages per second

Using the model and the scenarios, Gryphon system performance can be compared with centralized and distributed CORBA object managers:

System 1 (ORB centralized CORBA) This object manager is a centralized ORB. There is a single server that stores all objects, so any reference to an object requires a remote reference. In addition, since the ORB has no special knowledge of the application, a send and a receive message is required to determine the state of an object. Because the ORB is centralized and because of the amount of traffic, the server will likely be a bottleneck.

$$T_{\text{VPR}} = 2(MU + F(N + S))$$

$$T_{\text{app}} = 2MU$$

$$T_{\text{total}} = 2(NU + FV(N + S))$$

System 2 (ORB distributed CORBA) The object manager is a distributed configuration of an ORB. All objects are randomly and equally distributed among the processes. The ORB is not centralized and local objects do not result in message traffic. Now, accesses that would have gone to the central ORB now go to the process where the object is located (distribution addresses the implicit

bottleneck due to centralized configurations). In T_{VPR} the first part of the expression represents read operations by the local client and the second part represents reads by external clients to the data stored on the local server. Note that the expression includes references due to frame updates (a DVE needs to render objects, it would implicitly read each object at the frame update rate).

$$T_{\text{VPR}} = 2MU(((L-1)/L) + ((L-1)/L)) + 2F(N+S)(((L-1)/L) + (V-1)/L))$$

$$T_{\text{app}} = 2MU(((L-1)/L) + ((L-1)/L)) + 2F(N+S)((1/L)V)$$

$$T_{\text{total}} = 2NU((L-1)/L) + 2F(N+S)((L-1)/L)V$$

System 3 (Gryphon with location policy) The object manager includes a Gryphon capable of acting only on location hints. Like System 2, objects are evenly distributed across processes but in this case they are assumed to be located on the client making the modifications.

$$T_{\text{VPR}} = 2F(N+S)(((L-1)/L) + (V-1)/L))$$

$$T_{\text{app}} = 2F(N+S)((1/L)V)$$

$$T_{\text{total}} = 2F(N+S)((L-1)/L)V$$

System 4 (Gryphon with location and caching policies) The object manager includes a Gryphon capable of acting on location and caching hints. The model reflects the fact that data are pushed to the clients instead of being pulled via request messages (i.e., we remove the 2X multiplier to reflect the absence of a send message).

$$T_{\text{VPR}} = MU(L-1)$$

$$T_{\text{app}} = TVPR$$

$$T_{\text{total}} = NU(L-1)$$

System 5 (Gryphon with location, caching, and consistency policies) This configuration is the full Gryphon system.

$$T_{\text{VPR}} = MU(L-1)R$$

$$T_{\text{app}} = T_{\text{VPR}}$$

$$T_{\text{total}} = NU(L-1)R$$

Models	Scenario A	Scenario B	Scenario C
System 1	528,004	5,764	2,000,000
System 2	1,054,950	10,952	3,600,000
System 3	1,054,940	10,944	0
System 4	1,998	38	9,000,000
System 5	20	19	900

T_{VPR} Comparison

System	Scenario A	Scenario B	Scenario C
System 1	4	4	2,000,000
System 2	528,008	5,768	3,600,000
System 3	528,000	5,760	0
System 4	1,998	38	9,000,000
System 5	20	19	900

T_{app} Comparison

System	Scenario A	Scenario B	Scenario C
System 1	528,004,000	115,280	20,000,000
System 2	527,476,000	109,516	18,000,000
System 3	527,472,000	109,440	0
System 4	1,998,000	760	90,000,000
System 5	19,980	380	9,000

T_{total} Comparison

Table 6: Gryphon System Performance Comparison

Table 6 summarizes network message traffic using the load generated by the three scenarios, and supported by the five different object management configurations. Some highlights of the results are:

- System 1 (centralized CORBA) and System 2 (distributed CORBA) are subject to substantially more traffic than the others in almost every scenario due to their requirement to support location transparency (and not supporting caching).
- Application-favored object placement has a substantial impact in Scenario C, and consequently Gryphon performs much better than the other systems with location transparency.
- Caching (System 3) results in large performance gains in Scenario B, but results in unnecessary cache consistency updating in Scenarios A and C.

- The network traffic for Scenario 3 with System 3 is zero since the analysis did not show infrequent reads of small portions of data. In any case, the load is negligible.

In general, the table shows that the Gryphon approach significantly reduces the message traffic rate compared to the other approaches; the total message traffic, T_{total}, for the Gryphon system is only a fraction of a percent of centralized and distributed CORBA systems for all three scenarios. This work illustrates how object policies can be cast as execution levels to support the principles, and also shows the relative performance at the different levels. Further experiments and results are reported in [10].

4 In-Kernel Pipeline Module Thread Control

This aspect of the work focuses on support for continuous media flow between devices. Several studies have shown how to embed application-specific code in a kernel so that it can perform operations specific to the data stream (e.g., see [6, 7, 9]). However, these approaches do not allow the application to influence the way resources are allocated to the components to address application-specific tradeoffs. The Real-time, Parametrically Controlled In-kernel Pipe (RT-PCIP) facility provides a means for inserting / deleting filter programs into / from a logical stream between two devices, where each filter can be parametrically controlled by the negotiation mechanism (see Figure 3). The Execution Performance Agent (EPA) is the negotiation mechanism that interacts with the user-space part of the application to dynamically adjust scheduling priorities to achieve soft real-time control over the pipeline.

The EPA / RT-PCIP supports the principles for soft real-time as follows:

Principle 1: Soft Real-Time Diverse forms of computation can be expressed in terms of deadline confidence and execution time reliability (analogous to execution levels).

Principle 2: Application Knowledge The application provides desired deadline confidence and reliability in the expected execution time rather than a simple priority to the EPA.

Principle 3: Dynamic Negotation The EPA supports negotiated management of pipelines through an admission policy based on relative execution time reliabilities and requested deadline confidences. Requested confidence in deadlines may be renegotiated online.

The EPA and the policy for admission and confidence negotiation are derived from an extension to the deadline monotonic scheduling and admission policy which relaxes the hard real-time require-

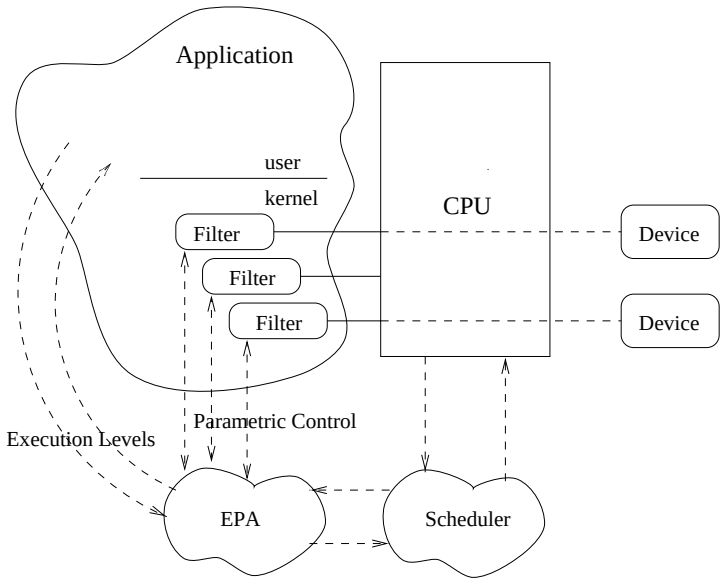


Figure 3: The EPA / RT-PCIP Architecture

ment that worst-case deterministic execution time be provided. The EPA provides the confidence and reliability semantics and the online monitoring of actual reliability and admission testing. In order to describe how the EPA is able to provide execution control in terms of deadline confidence and execution time reliability, we will review deadline monotonic hard real-time scheduling and show how it can be extended to implement the EPA.

In this aspect of the work, the goal is to dynamically negotiate the policy for allocating resources used to move data from one node to another, or within one node, from one device such as a disk to another device such as the sound card. The RT-PCIP architecture uses existing techniques for creating modules to be embedded in kernel space as extensions of device drivers (c.f. [3, 7]). Each device has an interface module that can be connected to an arbitrary pipe-stage filter; a pipeline is dynamically configured by inserting filters between a source and sink device interface. An application in user-space monitors summary information from the kernel in order to control the movement of data between the source and sink devices. The purpose of the EPA is to interact with the user-space application and with the modules in the pipeline. Specifically, it provides status information to the user-space program, and accepts parameters that control the behavior of the filter modules, and ensures that data flows through the pipeline according to real-time constraints and estimated module execution times.

4.1 Soft Real-Time Pipeline Control

In a thread-based operating system environment, pipe module execution is controlled by a kernel thread scheduler—typically a best-effort scheduler. As long as the system does not become overloaded, the pipe facility will provide satisfactory service. In overload conditions the EPA dynamically computes new priorities for the threads executing the modules, then provides them to the scheduler so that it can allocate the CPU to threads with imminent deadlines.

Since hard real-time systems guarantees that each task admitted to the system can be completed prior to a prespecified deadline, they are, of necessity, conservative. Processing time estimates are expressed in terms of the worst case execution time (WCET), admission is based on the assumption that every task uses its maximum amount of resources, and the schedule ensures that all admitted tasks execute by their deadline. Continuous media applications have softened deadline requirements: The threads in a continuous media pipe must *usually* meet deadlines, but it is acceptable to occasionally miss one. When the system is overloaded—the frequency of missed deadlines is too high—the EPA reduces the loading conditions by reconfiguring the pipeline, e.g., by removing a compression filter (trading off network bandwidth for CPU bandwidth).

The EPA design is driven by experience and practicality: Rather than using WCET for computing the schedule, it uses a range of values with an associated confidence level to specify the execution time. The additional requirement on the application is to provide execution time estimates with a range and a confidence; this is only a slightly more complex approach than is described in the use of Rialto [14]. An application that loads pipeline stages must specify the following parameters:

- Service type common to all modules in a single pipeline: guaranteed, reliable, or best-effort
- Computation time: WCET for guaranteed service, expected execution time (with specification of distribution, such as a normal distribution with mean σ and a specified number of samples) for reliable service, or none for best-effort service
- Input source or device interface designation
- Input and output block sizes
- Desired termination and soft deadlines with confidence for reliable service (D_{term}, D_{soft}, $confidence_{term}$, and $confidence_{soft}$)

- Minimum, R_{min}, and optimal, R_{opt}, time for output response
- Release period (expected minimum interarrival time for aperiodics) and I/O periods

4.2 EPA-DM Approach to Thread Scheduling

The approach for scheduling RT-PCIP thread execution is based on a branch of hard real-time scheduling theory called Deadline Monotonic (DM) [1]. DM consists of fixed-priority scheduling in which threads are periodic in nature and are assigned priorities in inverse relation to their deadlines. For example, the thread with the smallest deadline is assigned the highest priority. DM has been proven to be an optimal scheduling policy for a set of periodic threads in which the deadline of every thread is less than or equal to the period of the thread.

In addition, the concept of EPA-DM thread scheduling for pipeline stages is based on a definition of soft and termination deadlines in terms of utility and potential damage to the system controlled by the application (see Figure 4 and [5]). Figure 4 shows response time utility and damage in relation to soft and termination deadlines as well as early responses. The EPA signals the controlling application when either deadline is missed, and specifically will abort any thread not completed by its termination deadline. Likewise, the EPA will buffer early responses for later release at R_{opt}, or at R_{min} worst case. Signaled controlling applications can handle deadline misses according to specific performance goals, using the EPA interface for renegotiation of service. For applications where missed termination deadline damage is catastrophic (i.e. the termination deadline is a "hard deadline"), the pipeline must be configured for guaranteed service rather than reliable service.

The DM theories do not apply directly to the in-kernel pipeline mechanism, because DM is appropriate only for hard real-time systems. The EPA-DM schedulability test eases restriction on the DM admission requirements to allow threads to be admitted with *expected* execution times (in terms of an execution confidence interval), rather than requiring deterministic WCET. The expected time is determined using offline estimates of the execution time based on confidence intervals. Knowledge of expected time can be refined online by the EPA each time a thread is run. By relaxing the WCET admission requirement, more complex processing can be incorporated, and pessimistic WCET with conservative assumptions (e.g. cache misses and pipeline stalls) need not reduce utility of performance-oriented pipelines which can tolerate occasional missed deadlines (especially if the probability of a deadline miss can be quantified beforehand).

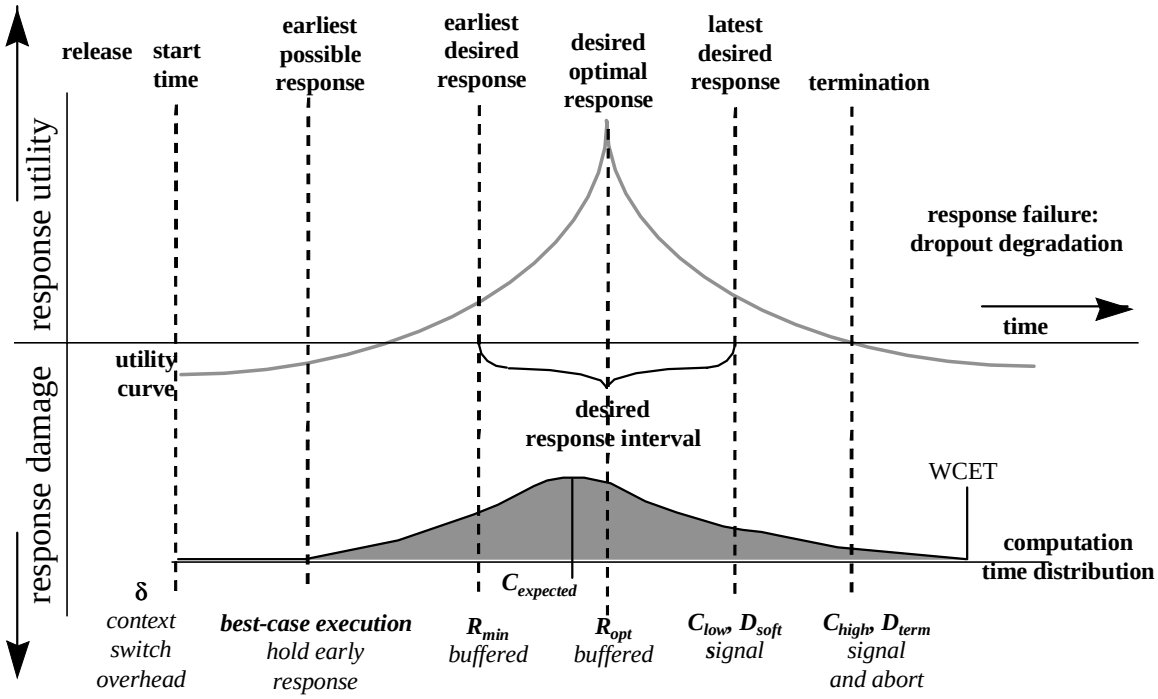


Figure 4: Execution Events Showing Utility and Desired Response

The evaluation of the EPA-DM schedulability test based on an execution duration described by confidence intervals results in probabilistic performance predictions on a per-thread basis, in terms of the expected number of missed soft and termination deadlines. For simplification in the formulas, all other threads are assumed to contribute the maximum amount of "interference", which can be loosely defined as the amount of time spent executing threads other than the one in question. The confidence in the number of missed soft and termination deadlines is largely a function of the confidence the EPA user has in the execution time. For example, if a thread has an execution time confidence of 99.9% and passes the admission test, then it is expected to miss its associated deadline 0.1% of the time or less.

The sufficient (but not necessary) schedulability tests for DM is used in part to determine schedulability in the EPA-DM scheduling policy shown in Figure 5; here it is assumed that computation time is expressed as a normal distribution (the normal distribution assumption is not required, but greatly reduces the number of offline samples needed compared to assuming no distribution). $I_{max}(i)$ is the interference time by higher priority threads $j = 1$ to $i - 1$ which preempt and execute a number of times during the period in which thread i runs. The number of times that thread k executes during a period of thread i is based on the period and execution time of thread k. $C_{low}(i)$ is the shortest execution duration of thread i, $C_{high}(i)$ is the longest execution duration of thread i, and T_j is the period

Eq. 1: (From probability theory for a normal distribution)

$$C_{low\ or\ high}(i) = C_{expected}(i) + Z_{p_{low\ or\ high}}(i)(\frac{\sigma(i)}{\sqrt{N_{\text{trials}}(i)}})$$

Eq. 2: (EPA-DM admission test)

$$\forall i : 1 \leq i \leq n : \frac{C_{low\ or\ high}(i)}{D_{soft\ or\ term}(i)} + \frac{I_{max}(i)}{D_{soft\ or\ term}(i)} \leq 1.0?$$

where

$$I_{max}(i) = \sum_{j=1}^{i-1} \lceil \frac{D_{term}(i)}{T(j)} \rceil C_{term}(j)$$

Figure 5: Schedulability Formulas for EPA-DM Policy

of thread j. $Z_{p_{low}}(i)$ and $Z_{p_{high}}(i)$ are the unit normal distribution quantiles for the execution time of thread i.

An example illustrates the use of the EPA-DM scheduling theory. Assuming that there are two threads that have a normal distribution of execution times, and that the worst-case execution time, WCET(i), is known for comparison, the attributes of the threads are shown in Table 7. If these threads can be scheduled based on the EPA-DM scheduling admission test, then thread 1 has a probability of completing execution before D_{soft} of at least 99.9% expressed $P(C_{low} < D_{soft}) \geq 0.999$. Similarly, probability $P(C_{high} < D_{term}) \geq 0.9998$. Likewise thread 2 has respective deadline confidences $P(C_{low} < D_{soft}) \geq 0.95$ and $P(C_{high} < D_{term}) \geq 0.9998$.

Thread	$C_{exp.}$	σ	N_{trials}	$Z_{p_{low}}$	$conf_{soft}$	$Z_{p_{high}}$	$conf_{term}$	WCET	D_{soft}	D_{term}	T
1	40	15	32	3.29	99.9%	3.72	99.98%	58	50	60	250
2	230	50	32	1.96	95%	3.72	99.98%	310	400	420	500

Table 7: Parameters for Example Threads

The equations in Figure 5 are used to determine the schedulability of the two threads using execution time confidence and desired D_{soft} and D_{term} confidence.

Thread 1

Using eq. 1:

$$C_{high}(1) = 40 + 3.72\frac{15}{\sqrt{32}} = 49.86$$

$$C_{low}(1) = 40 + 3.29\frac{15}{\sqrt{32}} = 48.72$$

Because Thread 1 has the shorter deadline of the two threads, it is assigned the highest priority. Therefore, the interference term, $I_{max}(i)$, is zero, which simplifies the schedulability test for Thread 1. In this case, Equation 2, as applied to Thread 1, becomes:

$$\frac{C_{low\ or\ high}(i)}{D_{soft\ or\ term}(i)} \leq 1.0$$

The use of $C_{high}(1)$ in this formula shows $\frac{48.72}{50} \leq 1.0$, while the use of $C_{low}(1)$ in this formula shows $\frac{49.86}{50} \leq 1.0$, so this thread is schedulable.

Thread 2

Using eq. 1:

$$C_{high}(2) = 230 + 3.72\frac{50}{\sqrt{32}} = 262.88$$

$$C_{low}(2) = 230 + 1.96\frac{50}{\sqrt{32}} = 247.32$$

Using eq. 2:

$$\frac{C_{loworhigh}(2)}{D_{softorterm}(2)} + \frac{I_{max}(2)}{D_{softorterm}(2)} \leq 1.0?$$

$$I_{max}(2) = \lceil\frac{D_{term}(2)}{T(1)}\rceil D_{term}(1) = 2 * 60$$

The meaning of $I_{max}(2) = 2 * 60$ is that Thread 2 can be interrupted twice during its period by Thread 1, and that in each case Thread 1 might execute until it is terminated by the EPA at $D_{term}(2)$. Evaluating with C_{high} yields

$$\frac{247.32}{400} + \frac{2 * 60}{400} \leq 1.0$$

Evaluating with C_{low} yields

$$\frac{262.88}{420} + \frac{2 * 60}{420} \leq 1.0$$

Because both of these formulas are satisfied, Thread 2 is schedulable.

The example shows how the EPA-DM scheduling approach supports soft real-time computation in which it is not necessary to guarantee that every instance of a periodic computation complete execution by its deadline. In fact, although it is not shown here, the use of WCET in the basic DM formulas result in the lack of schedulability of thread 2. WCET is a statistical extreme, and cannot be guaranteed.

In general, the RT-PCIP mechanism, in conjunction with the EPA-DM scheduling approach offers new, flexible support for device-to-device processing such as needed by DVEs. Threads can be created, executed, and monitored in order to deliver predictable, quantifiable performance according to the application's needs. Operating system overhead is kept to a minimum, as the amount of dynamic interaction between application code and the operating system is low.

5 Dynamically Negotiated Scheduling

There are also more general cases where there is a need to control the soft real-time execution of applications in a DVE. Soft real-time applications must be able to modify their resource consumption (and consequently the quality of their output) at any given time based on the relative importance of the data to the user, the amount of physical resources available to the applications, and the importance of other concurrently executing applications.

This section discusses our work in applying dynamic negotiation to CPU scheduling in support of soft real-time application execution in which a middleware *Dynamic QoS Manager* (DQM) allocates a CPU to individual applications according to dynamic application need and corresponding user satisfaction. Applications are able to trade off individual performance for overall user satisfaction, cooperating to maximize user satisfaction by selectively reducing or increasing resource consumption as available resources and requirements change.

A Quality of Service (QoS) [2] approach can be applied to scheduling to provide operating system support for soft real-time application execution. A QoS system allows an application to reserve a certain amount of resources at initialization time (subject to resource availability), and guarantees that these resources will be available to the application for the duration of its execution. Applied to scheduling, this means that each application can reserve a fixed percentage of the CPU for its sole use. Once the available CPU cycles have been committed, no new applications can begin executing until other applications have finished, freeing up enough CPU for the new applications requests to be met.

In a soft real-time environment, the application needs a reasonable assurance (rather than an absolute assurance) that resources will be available on request. In both QoS and hard real-time environments, the system makes strict guarantees of service, and requires that each application make a strict statement of its resource needs. As a result, applications in these environment must use worst case

estimates of resource need. In soft real-time systems, applications make more optimistic estimates of their resource needs, expecting that the operating system will generally be able to meet those needs on demand and will inform the applications when it is unable to do so.

Several operating systems designers have created designs and interfaces to support some form of soft real-time operation. These new operating systems interfaces allow a process to either (1) negotiate with the operating system for a specific amount of resources as in RT Mach [17] and Rialto [14]; (2) specify a range of resource allocations as in MMOSS [8]; or (3) specify a measure of application importance that can be used to compute a fair resource allocation as in SMART [19]. These systems all provide a mechanism that can be used to reduce the resource allotment granted to the running applications. Even though the system is able to allocate resources more aggressively, the hypothesis is that soft real-time applications will still perform acceptably. Since their average case resource requirements may be significantly lower than the worst-case estimates, resources can be allocated so that the benefit is amortized over the set of executing applications.

In creating resource management mechanisms, operating systems developers have assumed that it is possible for applications to adjust their behavior according to the availability of resources, but without providing a general model of application development for such an environment. In the extreme, the applications may be forced to dynamically adapt to a strategy in which the resource allocation is less than that required for average-case execution. Mercer, et al. suggest that a dynamic resource manager could be created to deal with situation of processor overload [17]. In Rialto, the researchers have used the mechanism to develop an application repertoire (though there was apparently no attempt to define a general model for its use).

In the DQM framework (see Figure 6) applications are constructed to take advantage of such mechanisms without having to participate directly in a detailed negotiation protocol. The framework is based on execution levels; each application program is constructed using a set of strategies for achieving its goals where the strategies are ordered by their relative resource usage and the relative quality of their output. The DQM interprets resource usage information from the operating system and execution level information from the community of applications to balance the system load, overall user satisfaction, and available resources across the collection of applications. Section 5.3 describes experiments conducted to evaluate the approach.

The DQM framework supports the fundamental principles asserted in this paper for soft real-time

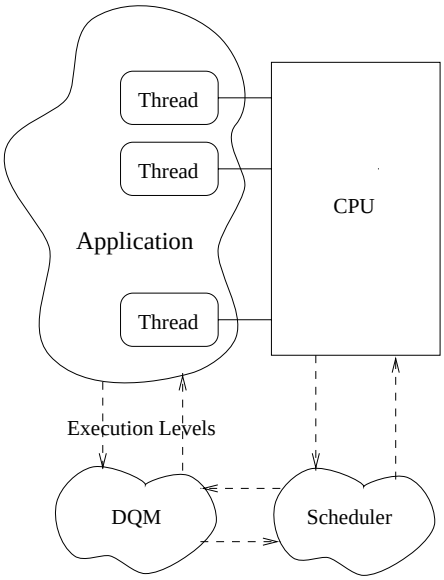


Figure 6: The DQM Architecture

processes as follows:

Principle 1: Soft Real-Time The DQM supports generic softening of real-time processes. Through execution levels, applications may modify their period and algorithms to implement any arbitrary soft real-time policy.

Principle 2: Application Knowledge The application provides three pieces of information per execution level: Resources needed, benefit provided, and period.

Principle 3: Dynamic Negotation The DQM dynamically adjusts application levels at runtime based on application deadline misses and current system state.

5.1 Execution Levels in the DQM

For this aspect of the work, each application execution is characterized by a set of quadruples:

$$\{Level_i, Resource_i, Benefit_i, Period_i\}$$

where $Level_i > Level_j \Rightarrow Resource_i > Resource_j$, and where $Level_i > Level_j \Rightarrow Benefit_i > Benefit_j$.

At runtime, each application specifies its maximum CPU requirements, maximum benefit, and its set of quadruples (Level, Resource usage, Benefit, Period) to the DQM. Level 1 represents the

Maximum benefit: 6
Maximum CPU usage: 0.75
Number of execution levels: 6

Level	CPU	Benefit	Period(mS)
1	1.00	1.00	100
2	0.80	0.90	100
3	0.65	0.80	100
4	0.40	0.25	100
5	0.25	0.10	100
6	0.00	0.00	100000

Table 8: Quadruples for an Example Application

highest level and provides the maximum benefit using the maximum amount of resources, and lower execution levels are represented with larger numbers. For example, an application might provide information such as shown in Table 8, which indicates that the maximum amount of CPU that the application will require is 75% of the CPU, when running at its maximum level, and that at this level it will provide a user-specified benefit of 6. The table further shows that the application can run with relatively high benefit (80%) with 65% of its maximum resource allocation, but that if the level of allocation is reduced to 40%, the quality of the result will be substantially less (25%).

5.2 Dynamic QoS Manager (DQM)

The DQM dynamically determines a specific allocation profile that best suits the needs of the applications and the user while conforming to the requirements imposed by resource availability, as delivered by the operating system. At runtime, applications monitor themselves to determine when deadlines have been missed and notify the DQM in such an event. In response, the DQM informs each application of the level at which it should execute. A modification of execution level causes the application to internally change the algorithm used to execute. This allows the DQM to leverage the mechanisms provided by systems such as RT Mach, Rialto, and SMART in order to provide CPU availability to applications.

The DQM dynamically determines a level for the running applications based on the available resources and benefit. Resource availability can be determined in a few different ways. CPU overload is determined by the incidence of deadline misses in the running applications. CPU underutilization is determined by CPU idle time. In the current DQM this is done by reading the CPU usage of a low

priority application. In situations of CPU overload (and consequently missed deadlines), levels are selected that reduce overall CPU usage while maintaining adequate performance over the set of running applications. Similarly, in situations of CPU underutilization, levels are selected so as to increase overall CPU usage.

Four resource allocation policies have been examined for use with the DQM:

Distributed. When an application misses a deadline, the application autonomously selects the next lower level. A variation of this policy allows applications to raise their level when they have successfully met N consecutive deadlines, where N is application-specific. This policy could be used in conjunction with RT Mach reserves, MMOSS, and SMART.

Fair. This policy has an even and a proportional option: In the event of a deadline miss, the *even* option reduces the level of the application that is currently using the most CPU. It assumes that all applications are equally important and therefore attempts to distribute the CPU resource fairly among the running applications. In the event of underutilization, this policy raises the level of the application that is currently using the least CPU time. The *proportional* option uses the benefit parameter and raises or lowers the level of the application with the highest or lowest benefit / CPU ratio. This policy approximates the scheduling used in the SMART system.

Optimal. This policy uses each application's user-specified benefit (i.e., importance, utility, or priority) and application-specified maximum CPU usage, as well as the relative CPU usage and benefit information specified for each level to determine a QoS allocation of CPU resources that maximizes overall user benefit. This policy performs well for initial QoS allocations, but our experiments have shown that execution level choice can fluctuate wildly. As a result, a second option was implemented that restricts the change in level to at most 1. This policy is similar to the value-based approach proposed for the Alpha kernel [12].

Hybrid. This policy uses Optimal to specify the initial QoS allocations, and then uses different algorithms to decide which levels to modify dynamically as resource availability changes. The two options we have implemented use absolute benefit and benefit density (benefit / incremental CPU usage) to determine execution level changes.

Application 1
Maximum benefit: 8
Max CPU usage: 0.42
No. of levels: 9

Level	CPU	Benefit
1	1.00	1.00
2	0.51	0.69
3	0.35	0.40
4	0.27	0.30
5	0.22	0.24
6	0.15	0.16
7	0.10	0.10
8	0.05	0.05
9	0.00	0.00

Application 2
Maximum benefit: 4
Max CPU usage: 0.77
No. of levels: 6

Level	CPU	Benefit
1	1.00	1.00
2	0.59	0.64
3	0.53	0.55
4	0.45	0.47
5	0.22	0.24
6	0.00	0.00

Application 3
Maximum benefit: 5
Max CPU usage: 0.22
No. of levels: 8

Level	CPU	Benefit
1	1.00	1.00
2	0.74	0.92
3	0.60	0.39
4	0.55	0.34
5	0.27	0.23
6	0.12	0.11
7	0.05	0.06
8	0.00	0.00

Application 4
Maximum benefit: 2
Max CPU usage: 0.62
No. of levels: 4

Level	CPU	Benefit
1	1.00	1.00
2	0.35	0.31
3	0.21	0.20
4	0.00	0.00

Table 9: Synthetic Program Characteristics

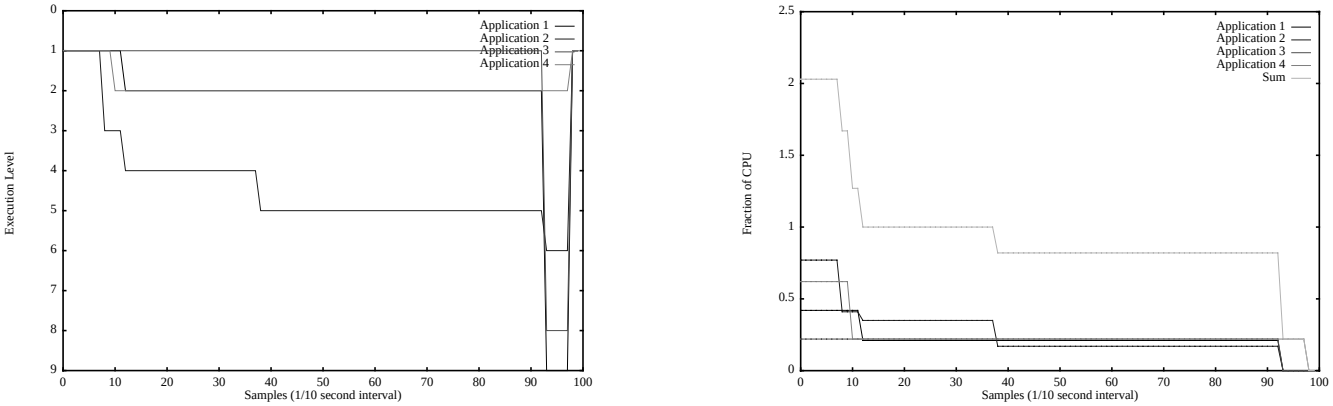
5.3 DQM Experiments

For the experiments presented in this section we represented VPR applications with synthetic applications. The synthetic applications consume CPU cycles and attempt to meet deadlines in accordance with their specified execution levels, without performing any useful work. The synthetic applications are generated as random programs that meet the desired general criteria—random total QoS requirement, absolute benefit, number of execution levels, and relative QoS requirements and benefit for each level. The synthetic applications are periodic in nature, with a constant period of 0.1 second—applications must perform some work every period. While this does not reflect the complete variability of real applications, it simplifies the analysis of the resulting data.

For a given set of applications, data was generated by running the applications and the DQM and recording 100 samples of the current level, expected CPU usage, and actual CPU usage for each application, as well as the total CPU usage, total benefit over all applications, and current system idle time. The applications ran for a total of 10 seconds (100 periods). Our results indicate that this is adequate for observing the performance of the policies at steady state.

The experiments can be run with 1–9 applications each having between 2 and 9 levels. For simplifying the comparison presented here, a single representative set of synthetic applications was used. The execution level information for the application set is shown in Table 9. There are 4 applications, each having between 4 and 9 levels with associated benefit and CPU usage numbers.

Figure 7(a) shows the execution levels that result for the given application set when running the DQM with the Distributed policy with a skip value of 0. The skip value indicates the number of missed



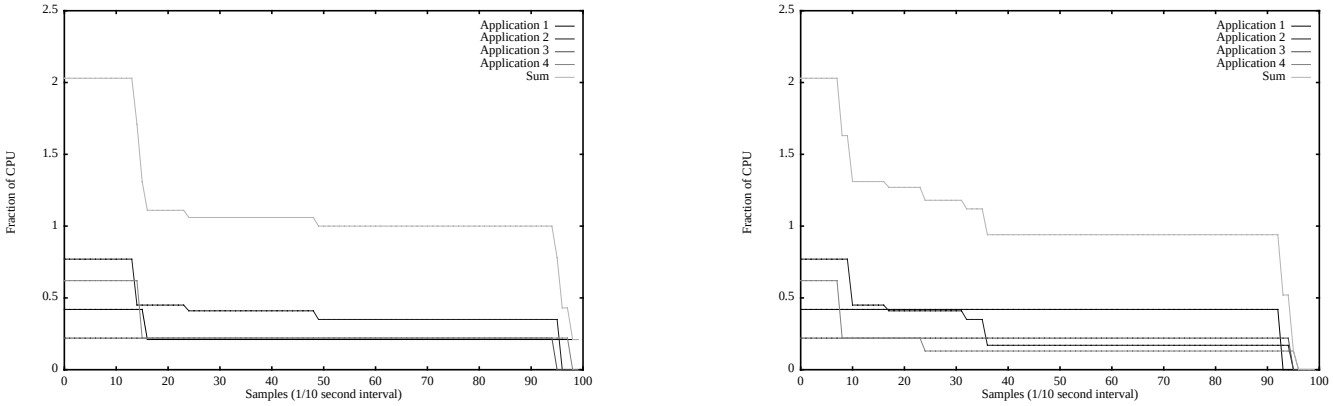
(a) Execution Levels

(b) CPU Usage

Figure 7: Performance of Distributed (skip=0)

deadlines that must occur in succession before the application reduces its execution level. The skip value of 0 means that the application reacts instantly in lowering its level, regardless of the transient nature of the overload situation. The execution levels can be seen to change rapidly at the beginning, because the system is started in a state of CPU overload, i.e. the combined QoS requirement for the complete set of applications running at the highest level (level 1) is approximately 200% of the CPU. By the 10th sample, the applications have stabilized at levels that can operate within the available CPU resources. There is an additional level adjustment of application 3 at the 38th sample due to an additional missed deadline probably resulting from transient CPU load generated by some non-QoS application. The lack of changes at the very beginning and the wild fluctuations at the end of each graph are a result of the start-up and termination of the applications at the beginning and end of each experiment, combined with a slightly longer than 1/10 second data recording sample interval. Figure 7(b) shows the CPU usage for the applications in the same experiment. The total requested CPU usage (designated Sum) starts out at approximately twice the available CPU, and then drops down to 1 as the applications are adjusted to stable levels. Note also the same adjustment at sample 38, lowering the total CPU usage to approximately 80%.

Figure 8(a) shows the CPU usage for the Distributed policy, with a skip value of 2. Using a larger skip value desensitizes the algorithm to deadline misses such that a level adjustment is only made for every 3rd deadline miss, rather than for each one. This can result in a longer initial period before stability is reached, but will result in less overshoot as it gives the applications time to stabilize after



(a) Distributed (skip=2)

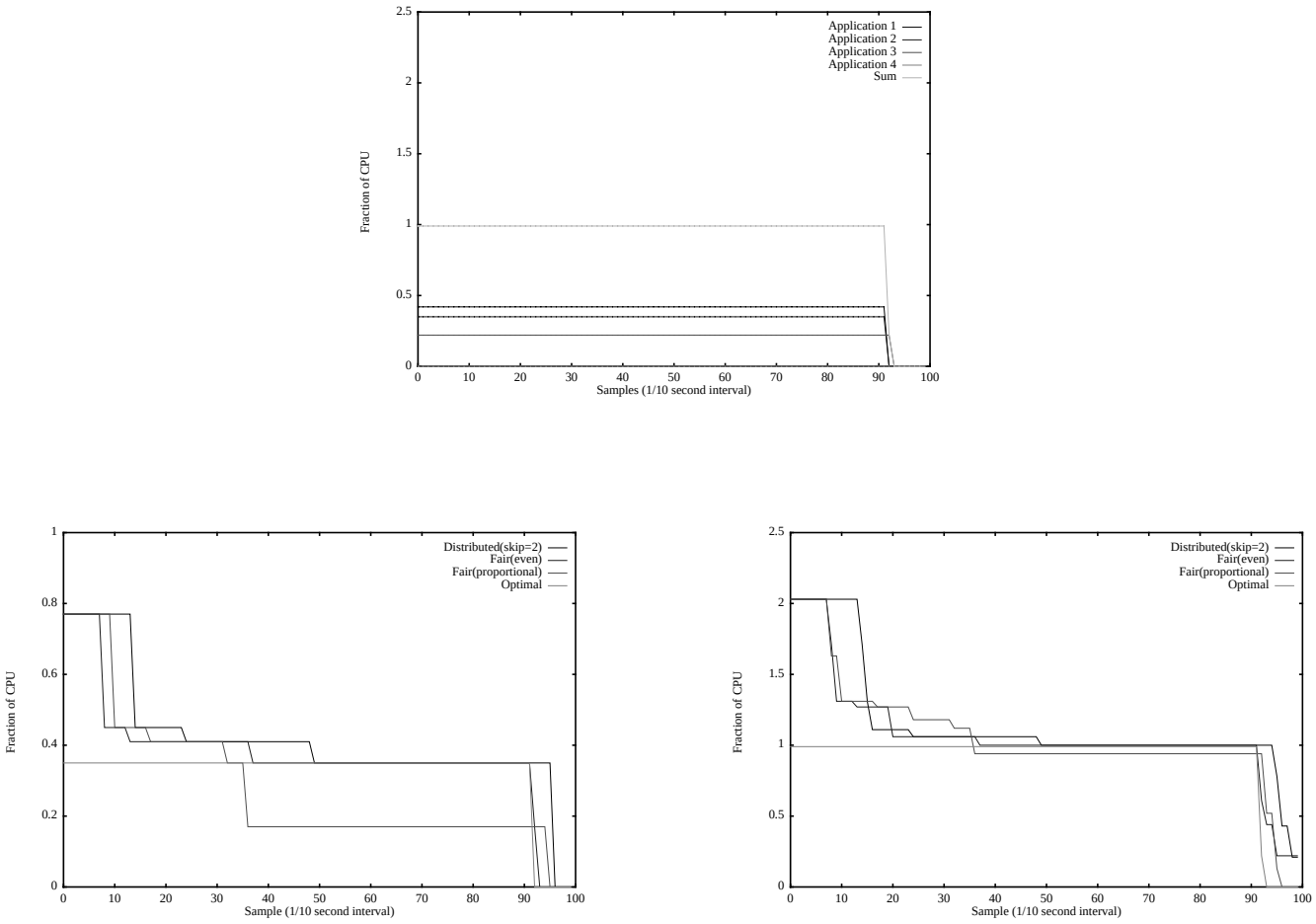
(b) Fair (proportional)

Figure 8: CPU Usage

level adjustments. Stability is not reached until about sample 16, and there are two small adjustments at samples 24 and 49. However, the overall CPU usage stays very close to 100% for the duration of the experiment with essentially no overshoot as is observed in Figure 7(b).

The results of running the applications with the Fair policy using the even option are not shown. This centralized policy makes decisions in an attempt to give all applications an equal share of the CPU. This policy generally produces results nearly identical to the Distributed policy, as it did with this set of applications. Figure 8(b) shows the results of running the applications with the Fair policy using the proportional option. This version of the policy attempts to distribute shares of the available CPU cycles to each application proportional to that application's benefit. Under the previous policies, the CPU percentage used by all applications was approximately the same. With this policy, the cpu-usage / benefit ratio is approximately the same for all applications. In fact, the ratio is as close to equal as can be reached given the execution levels defined for each applications.

Figure 9 shows the CPU usage for the applications running with the Optimal policy. This policy reaches steady state operation immediately, as the applications enter the system at a level that uses no more than the available CPU cycles. This policy optimizes the CPU allocation so as to maximize the total benefit for the set of applications, producing an overall benefit number of 14.88 as compared with 13.02 for the other policies. Note also that because this policy optimizes for benefit and not necessarily for utilization as in the other policies shown, it can result in a more stable steady state, yielding no additional deadline misses and requiring no corrections. However, this policy is the least stable given



(a) Application 2

(b) Sum

Figure 10: Performance of Four Policies

changing CPU resources, such as those caused by other applications entering or leaving the system.

Figure 10(a) shows the plots for application 2 with the four different policies. Figure 10(b) shows the summed CPU usage for the same four policies shown in Figure 10(a). This graph gives an indication of the time required for all applications to reach steady state, along with the CPU utilization resulting from the allocations. Figure 10(a), in particular, summarizes the differences between the various policies. The Optimal policy selects a feasible value immediately and so the level of the application is unchanged for the duration of the experiment. The Distributed and Fair (even) policies reach steady state at the same value, although they take different amounts of time to reach that state, the Distributed policy taking slightly longer. The Fair (proportional) policy reaches steady state at about the same time as the Distributed and Fair policies, although its allocation is slightly less in this case.

In general, these experiments show that given a set of level-based applications, it is possible to create a DQM that dynamically adjusts application execution levels to maximize user satisfaction within available resources, even in the absence of any underlying QoS or other soft real-time scheduling mechanisms. Four DQM decision policies demonstrate the range possibilities inherent in this model. In [4] the DQM is extended to support more extensive use of execution levels.

6 Summary and Conclusion

Next-generation multimedia applications require the timely delivery of complex data across and within nodes in dynamic computing environments. User requirements can change frequently; writing and executing applications that deliver and manage the data according to these rapidly-changing requirements requires new support from the operating system and development tools.

This paper has introduced a set of principles that have evolved from our work in designing a DVE and resource managers to support it. The principles address soft real-time, communicating application knowledge, and support for dynamic negotiation. The paper also presents a framework and execution levels to implement the principles. Execution levels have been used in three different contexts — the Gryphon distributed object manager, the EPA / RT-PCIP facility, and the DQM — to support soft real-time applications. In the Gryphon project the paper has shown how the approach can be used adjust object storage strategies to compensate for scarce network bandwidth. The EPA / RT-PCIP study describes how execution levels can be used to provide real-time support to kernel pipelines, using confidence and reliability as parameters in the execution levels. The DQM study demonstrates how execution levels can be used to implement very general dynamic negotiation on top of a best-effort operating system, including measurements to reflect the behavior of the system.

Acknowledgment

Authors Nutt, Brandt, and Griff were partially supported by NSF Grant No. IRI-9307619. Jim Mankovich has designed and almost single-handedly built three different versions of the VPR. Several graduate students, particularly Chris Gantz, have helped us through their participation in group discussions of virtual environments, human-computer interfaces, real-time, soft real-time, and performance of our prototype system.

References

[1] Neil C. Audsley, Alan Burns, Mike F. Richardson, and Andy J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *8th IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[2] Cristina Aurrecoechea, Andrew Campbell, and Linda Hauw. A survey of QoS architectures. In *Proceedings of the 4th IFIP International Workshop on Quality of Service*, March 1996.

[3] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, 1995.

[4] Scott Brandt, Gary Nutt, Toby Berk, and Marty Humphrey. Soft real-time application execution with dyanamic quality of service assurance. In *Sixth IEEE International Workshop on Quality of Service*, pages 154–163, May 1998.

[5] A. Burns. Scheduling hard real-time systems: A review. *Software Engineering Journal*, May 1991.

[6] Geoff Coulson, Andrew Campbell, Philippe Robin, Gordon Blair, Michael Papathomas, and David Hutchinson. The design of a QoS controlled ATM based communication system. *IEEE JSAC Special Issue on ATM Local Area Networks*, 1994.

[7] Kevin Fall and Joseph Pasquale. Exploiting in-kernel data paths to improve I/O throughput and CPU availability. In *Proceedings of the Winter 1993 USENIX Conference*, pages 327–333, January 1993.

[8] Changpeng Fan. Evalutions of soft real-time handling methods in a soft real-time framework. In *Proceedings of the 3rd International Conference on Multimedia Modeling*, Toulous, France, November 1996.

[9] Ramesh Govindan and David P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 68–80, 1991.

[10] Adam Jonathan Griff and Gary J. Nutt. Tailorable location policies for distributed object systems, March 1998. Submitted for publication. Available at http://www.cs.colorado.edu/ñutt/Home.html.

[11] Marty Humphrey, Toby Berk, Scott Brandt, and Gary Nutt. Dynamic quality of service resource management for multimedia applications on general purpose operating systems. In *1997 IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, December 1997.

[12] E. Douglas Jensen, C. Douglass Locke, and Hideyuki Toduda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 112–122. IEEE, 1985.

[13] M. B. Jones, P. Leach, R. Draves, and J. Barbera III. Support for user-centric modular real-time resource management in the Rialto operating system. In *Proceedings of the NOSSDAV'95*, 1995.

[14] Michael B. Jones, Daniela Rosu, and Marcel-Catalin Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, October 1997.

[15] R. Kordale, M. Ahamad, and M. Devarakonda. Object caching in a CORBA compliant system. *USENIX Computing Systems Journal*, 9(4), 1996.

[16] Silvano Maffeis and Douglas C. Schmidt. Constructing reliable distibuted communication systems with corba. *IEEE Communications*, 14(2), February 1997.

[17] Cliff Mercer, Stephan Savage, and Hideyuki Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 90–99, May 1994.

[18] Michael N. Nelson, Mark Linton, and Susan Owicki. A highly available, scalable ITV system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 54–67, 1995.

[19] Jason Nieh and Monica S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[20] Brian D. Noble, M. Sayanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile applications-aware adaptation for mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[21] Gary Nutt, Toby Berk, Scott Brandt, Marty Humphrey, and Sam Siewert. Resource management for a virtual planning room. In *Proceedings of the Third International Workshop on Multimedia Information Systems*, September 1997.

[22] Gary J. Nutt, Joe Antell, Scott Brandt, Chris Gantz, Adam Griff, and Jim Mankovich. Software support for a virtual planning room. Technical Report CU-CS-800-95, Department of Computer Science, University of Colorado, Boulder, December 1995.

[23] Shuichi Oikawa and Ragunathan Rajkumar. A resource centric approach to multimedia operating systems. In *Proceedings of IEEE Real-Time Systems Symposium Workshop on Resource Allocation Problems in Multimedia Systems*. IEEE, December 1996.

[24] OpenGL performance benchmarks. WWW page at http://www.specbench.org/gpc/opc.static, 1997.

[25] Ragunathan Rajkumar, Chen Lee, John Leboczky, and Dan Siewiorek. A resource allocation model for QoS management. In *Proceedings of the 1997 IEEE Real-Time Systems and Services Symposium*, 1997.

[26] M. Van Steen, P. Homburg, and A.S. Tanenbaum. The architectural design of globe: A wide-area distributed system, 1997.