Dynamically Negotiated Resource Management for Virtual Environment Applications

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

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

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

Abstract

Collaboration technology calls for new, innovative techniques for supporting informal communication and coordinated work. Distributed virtual environments provide one avenue for supporting this aspect of computer-supported work. We have built a multiperson distributed virtual environment using low-cost workstations interconnected with relatively high-speed networks. This domain makes use of interactive and on-demand continuous media in addition to a number of other tasks that fall on a spectrum between hard real-time and best-effort response. 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 the virtual environment 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 current needs and the availability of system resources. This paper describes three ways we have approached resource management based on a principle of dynamic negotiation between the application and the system resource management.

[*]Authors Nutt, Brandt, and Griff were 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 applications go beyond the traditional numerical and character data into various forms of continuous media such as audio and video; they also take advantage of object technology as a programming paradigm. *Distributed virtual environments* (DVEs) are an instance of this new class: a DVE supports a world containing various shared entities; users interact with the entities in the world using a multimedia workstation. DVEs are data-intensive environments, since shared information must be disseminated throughout the network of user machines. Operating systems manage the resources / facilities used for distributing the data using built-in policies; experience with contemporary operation systems shows that they do not provide the type of support need to handle data movement for applications like a DVE.

This paper describes a study of diverse aspects of systems support for the capture, management, and delivery of data in DVEs. We have created an instance of a DVE called the *virtual planning room* (VPR) to explore the feasibility of current software and hardware technology in supporting collaborative work in a DVE. Early experimentation with the VPR indicated that while it had potential as a means for providing the necessary functionality for collaboration, its performance was woefully inadequate for qualitatively evaluating the approach. It has been observed that the bottlenecks could be eliminated with a resource allocation policy in which resources were directed at the parts of the application that needed them at the moment—a resource management policy was needed that was sensitive to the dynamic needs of the user in the context of the VPR. Therefore, we began to define and experiment with ways for the system to provide more effective support for the VPR (as originally reported in the conference paper from which this paper is derived [28]). We found that the resource management policy for distributed objects is crucial to the overall performance, which led us to develop the Gryphon distributed object system [19]; this work is described in Section 2. Collaboration environments such as the VPR cause huge amounts of audio and video data to be frequently moved; Section 3 describes new, flexible support for device-to-device processing involving confidence-based execution of threads managing the real-time delivery and processing of this data across nodes [35]. Section 4 presents a generalized software architecture in which applications cooperate with resource managers through the use of *execution levels* and a dynamic quality of service manager (DQM) to allow applications and resource managers to dynamically negotiate the level of resource usage by the application [7, 20]. The approach used in all three cases enables sets of applications to generate or consume the most appropriate amount of data given the precise dynamics of the situation. This paper presents our current results in each of these areas, though our work continues. In the remainder of this section we elaborate on our motivating application (the VPR), highlight the resource management tradeoffs raised by its use, then introduce a new resource management philosophy that guides our work.

1.1 Motivation

There is a revolution taking place in the way people want to use computers in their work and play. Since its introduction in 1980, the personal computer has become a highly refined and cost-effective tool for supporting an individual's work (especially in providing tools for publishing documents and decision support). Modern desktop systems provide sophisticated tools for word processing, document publishing, spreadsheets, web browsing, and data manipulation.

Despite the emergence of networks in the same timeframe, the technology has not provided the same kind of revolutionary support for *collaborative work* as it has for personal work. Web-based technology has provided a path for the most significant mechanism for supporting collaboration, though the granularity of the interaction is essentially at the file / document level.[1] Though there is clearly adequate technology for many forms of group interaction (e.g., MBone [11], chat rooms, etc.) computers and networks have not provided the level of utility to users that has been done by other group-oriented technologies such as the telephone.

Elsewhere we described barriers to the effective use of distributed systems for supporting collaborative work [30]. Briefly, the fundamental issues relate to the style in which the system provides its support: In the *situated work* style, the computer is expected to be a logical lever for the work performed by an individual, with the work of partitioning a job into tasks then coordinating the execution of those tasks left to human ingenuity (e.g., see [9]). The *workflow* camp also advocates the use of computers for personal productivity, but believes that the system should also play a hand in coordinating the execution of the tasks (e.g., see [16, 26, 34, 38]). It is beyond the scope of this paper to address all the details of the differences between these two camps; suffice it to say that they are essentially at an im-

[1]Of course applets in an HTML document interact with their server at a much finer grain than a file, though they, too, have not yet provided the kind of environment that will directly address collaborative work.

passé. The essential element that is missing is a technology to support situated work at the same time it makes workflow more flexible for group work. This technology relates to the ability of group members to easily discuss their work in an environment that is superior to face-to-face meetings, telephone conversations, or even electronic mail exchanges. The basic premise of our work is that with today's hardware technology, it is quickly becoming feasible—and cost-effective—to use a network of computers to support informal communication in the context of shared electronic artifacts. One manifestation of such an environment is a DVE containing *domain-specific tools*. Not only can group members conduct informal discussions in the DVE, the common (virtual) artifact frames the discussion and provides a set of domain-specific tools manipulating that artifact.

Our VPR [31] is a multiperson DVE supporting free-form communication in a manner similar to electronic meeting rooms [15, 37] and other virtual environments [1, 5, 12], but is unique in its support for domain-specific tools. A VPR *world* is defined by a collection of objects, with visual representations and behaviors of varying complexity. An object that represents a human participant is called an *avatar*, which is a compound object with an eye object, ear object, and hand object. The avatar is a part of the world that other avatars can see and hear. Activity takes place in the world when objects interact with one another. Domain-specific tools are added to the VPR by incorporating additional objects having "complex" behavior. For example, a formal workflow / process modeling system can be embedded in the VPR to focus on group coordination [29].

The fundamental role of the VPR is to to provide real-time audio and video support across the network, to render objects on each user's screen (according to the avatar's orientation), and to provide an environment in which to add domain-specific extensions. The VPR is a client-server system where each person uses a client workstation to implement the human-computer interface. Hence, the client machine must render each visible artifact from a (VRML) representation in the corresponding object, and cause behaviors (such as modifications to objects) to be reflected in all other appropriate clients.

For our prototype, we used widely-available system software and interfaces: CORBA for the object interface, POSIX for the system call interface, and various network interfaces to support streams. This allowed us to explore the VPR design, implementation, and functionality, using commonly-available implementations, even though the performance of the prototypes was severely limited by the system implementation. Despite the number of papers focusing on VE / VR functionality, design, and user interfaces, (e.g., see [2, 21, 22]) there is surprisingly little on the effect the operating system has on the

(D)VE's performance. Once we had developed the rudimentary VPR, we were able to explore system software design and organization that might be well-suited to this application domain.

We believe that applications like the VPR will become increasingly popular, since they evolve distributed computer systems from environments for coarse-grained information sharing into intelligent communication environments. There are many issues to address in such a computing environment, including cognitive models, human factors aspects, etc. Our focus is on (1) the logical effect of providing an immersive, domain-specific, robust communication environment, and (2) on the system software technology required that can make this technology feasible and usable for collaborative work. We focus on the second issue in this paper.

1.2 Tradeoffs

In experimenting with the VPR, we have encountered a number of barriers to providing efficient allocation of the hardware to the VPR and its applications. Each of these barriers defines a set of tradeoffs that are encountered in order to deliver and manage the audio and video data across nodes and within a single node.

High-performance Graphic Rendering of Diverse Types of Artifacts A client machine is expected to render 24 frames / second, independent of the nature of the VRML descriptions and of other load on the machine. If the processing load is too large, frames will be lost. Ideally, if the VPR knows frames will be dropped, it could simplify some of the images; i.e., the VPR should be able to tradeoff the quality of the rendering of some objects to preserve the frame rate, or to decide to drop the frame rate and preserve a minimum quality of the image for certain objects. The VPR needs an indication from the operating system of its ability to service the load.

Supporting Continuous Media Audio and video streams flow among VPR objects. While protocols can make assurances regarding the isochronous network transfer rate, only a few operating systems attempt to make guarantees regarding the throughput rate through the operating system itself [10, 17, 18, 24, 25, 27]. We wanted VPR applications to be able to tradeoff loss, jitter, and latency in each stream with other activity in the VPR.

Distributed Objects Our application software is all object-oriented. The distributed nature of the VPR requires that most objects be shared among client machines, even though each machine may be rendering the VRML component of the object 24 times per second. Performance requirements made it impossible to use a distributed object manager with location transparency. An application should to be able to tradeoff shared object consistency against network traffic.

VPR Applications While it might be technically possible to statically tune a system to provide optimized support for the appearance and behavior of each object in a particular VPR, when extensions are added, the tradeoffs change according to the requirements of these extensions as well as the VPR. A better approach is to allow the VPR and its extensions to influence the tradeoffs on resource allocation that must be made by the operating system.

1.3 Dynamically Negotiated Resource Allocation

The VPR and similar multimedia applications produce transient loads on the system's resources, frequently resulting in localize overload conditions. At times, one subset of objects needs to dominate the personal computer's resources (because they implement the function that the user is attempting to accomplish), yet only a few seconds later a different set of objects needs the highest priority to the resources. We desired a resource management facility that took on certain characteristics of embedded software—the application assumes part of the responsibility for the resource allocation strategy—yet which fits within the general framework of a multiprogrammed operating system. We recognized that the brute force approach is for each component of the VPR to acquire (and perhaps even use) the maximum amount of resources it will ever need at all times. However, this approach leads to excessive hardware requirements that can only be met through extreme over-allocation. (It is arguable whether this approach is even feasible in software environments with many resource-intensive objects.) Instead, we pursued a programming environment in which sophisticated applications could dynamically adjust their requirements according to the availability of resources and the activity directed by the user. For example, if a part of the VPR is dormant, then we do not want to expend much resource on supporting it; if it is active, we want to direct as much resource to the component as necessary.

We base our approach on a general form of quality of service (QoS) contracts between application components and the system. A contract can change according to changes in the overall system load—the QoS is *dynamic*. When the situation changes, the application repertoire and the system jointly choose a new QoS contract through *negotiation*. We desire resource allocation policies based on a *dynamically negotiated QoS*.

2 The Gryphon Distributed Object Manager

The VPR, like many distributed applications, relies on distributed objects as a fundamental programming tool. From the outset, it was clear that the VPR components needed to have more control over object resource policies than is provided by a manager with built-in location transparency. Without this flexibility, the applications could not make performance tradeoffs based on access demands. This section first discusses the requirements of an object manager for the VPR, and then presents the *Gryphon* system, which has been designed and implemented to satisfy these requirements.

It is common for distributed shared memory systems to use caching; it is also clear that the distributed object service could benefit substantially if caching were applied to objects. Once an object is cached, its coherence becomes the dominant problem. Our resource management philosophy suggests that the application suite is the only software that has the appropriate knowledge for how often inconsistent objects should be made consistent. It follows that the distributed object manager should dynamically negotiate both the location and consistency policies with the application suite, consistent with the dynamic negotiation philosophy explained in Section 1. QoS.

In the VPR, we currently distinguish among four ways objects should be handled: (1) a *private-local object* exists in an address space, (2) a *shared-local object* can be referenced from another address space, (3) a *shared-nonlocal object* is a shared-local object in a remote address space, and (4) a *global object* exists in a shared address space. A private-local object is any ordinary C++ object in the VPR client application software, e.g., a list of objects of interest to the local avatar. A shared-local object is one which other clients can access with method calls, but which is supported by the local client—a shared-nonlocal object is the dual of a shared-local object; avatars are examples of shared-local / nonlocal objects. A global object is a part of the virtual environment that is not logically part of any client's responsibilities, e.g., a wall in the VPR.

The use of shared objects is constrained by access times—which is, in turn, determined by location and caching policy. If client X_0 is using object Y implemented on a server, then each time the client does anything that might cause an interaction with Y (e.g., move its own avatar), X_0 must check Y. Further, if X_0 *changes* Y (e.g., Y is to change position or orientation in the VPR due to actions by X_0), then all other X_i must see the effect of the change to Y.

In the VPR, objects should be cached based on the dynamic caching policy specified for the object

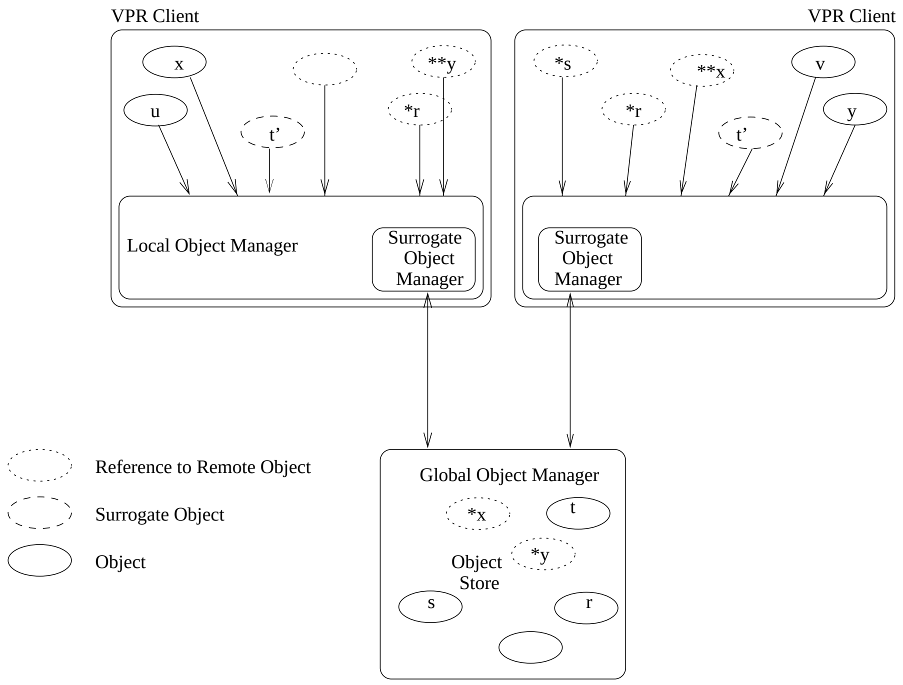


Figure 1: Object Locations

by the application (not by a policy built into the object service). Since some objects (such as a wall in a VPR) essentially never change in a session, cached copies are a natural solution. However, other objects might be in a high state of flux due to interaction with several different clients. How frequently should the object copies be made to be consistent? Again, this knowledge lies in the VPR software, not in the object server. This argues for a technique by which the application dynamically negotiates with the operating system to arrive at a *consistency update policy* for cached objects. In the VPR, the application writer should also be able to influence the choice of storage location.

Figure 1 illustrates the set of location policies that can be used in the VPR to reduce the requisite message traffic when objects are referenced. Some objects (like objects u and v) are private to an application. Other objects (like r and s in the figure) are kept only in a server with all references to the object being remote references over the network. Cached objects (such as t and cached copies $t^{'}$) have the original object stored on the server and copies in clients. Finally, objects such as x and y in the figure are placed at a client, yet can be referenced from other clients.

In addition to influencing the location and caching policies of an object, the application can weaken the consistency requirement [36] for cached objects. For example, if an object is being edited by one avatar on one client, other client machines may only need to update their cached copy of the object

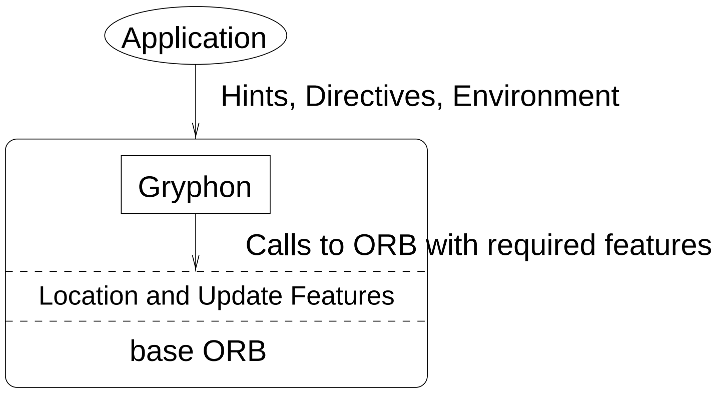

Application
Hints, Directives, Environment
Gryphon
Calls to ORB with required features
Location and Update Features
base ORB

Figure 2: ORB with the Gryphon System

every few minutes (they may even use session semantics for this type of object consistency).

The requirements of object management have led to the development of the *Gryphon* system [19], in which applications influence object management policy by dynamically providing *hints* and other directives regarding the location, caching, and consistency policy on a per object basis These hints are analyzed by the distributed Gryphon system (embedded in each object manager) which translates them into object manager policies for placing, caching, and updating each object. If no hints are provided, the object manager uses its default policies. The hints are evaluated at runtime, allowing objects to be changed as their requirements change. Thus these hints affect the object's distribution and its update rate on a global level and on a host-by-host basis.

Figure 2 describes the general architecture of the Gryphon system. Each application uses the CORBA IDL interface to reference objects. The base ORB is extended to include a Gryphon system to process policy hints as specified by the application. An object provides hints and directives (using additional method calls) which are caught by the Gryphon policy module and analyzed in the context of the state of the system and the nature of the the collective hints regarding each object. A more complete description of the hint semantics and the Gryphon system are beyond the scope of this paper, but they can be found in [19].

To analyze the performance of a Gryphon system implementation, models based on the VPR environment are 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 1 identifies parameters used to characterize message traffic:

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 1: Parameters used to Model Network Traffic

These parameters are used to derive equations for three metrics:

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

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

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

In the Gryphon system, the metrics are:

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

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

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

Next, we compare the Gryphon system performance with three different object managers in terms of the network traffic:

System 1 (Centralized object manager) This object manager is a centralized object manager implemented to provide the VPR prototype with distributed objects. There is a single server that allows objects to be cached to each client location. The object manager leaves consistency entirely up to the applications. In the model, we assume that any reference to an object requires consistency, thus the reference is remote.

$$T_{\text{VPR}} = U(N+M)$$

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

$$T_{\text{total}} = UL(N+M)$$

System 2 (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 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 3 (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. The problem for measurement is that 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).

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

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

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

In order to compare the four approaches, we modeled five different scenarios to represent common object reference patterns:

Scenario A: Virtual Art Museum The DVE contains a number of works of art available for viewing and discussion. A person enters the virtual museum, and may browse various works without communication. However, it is expected that the person will wish to find other people who are interested in specific works, then to discuss the work. The DVE represents a person's presence through the presence of an avatar in the environment; when one person sees a number of avatars near an interesting piece of work, then that person can join the group, view the work, and begin discussing it. The virtual art museum has a number of static objects with complex VRML specifications. Avatars move infrequently, but most other objects do not move at all. The first part of Table 2 shows the values used for each load parameter to model this scenario.

Scenario B: Collaboratively Flying an Unoccupied Air Vehicle In the RT-PCIP work (see Section 3) we build an unoccupied air vehicle, FLOATERS, for proof-of-concept testing. We also used considered techniques for accomplishing collaborative work—in this case, flying FLOATERS—using the VPR. In this FLOATERS experiment, avatars are in the virtual space together and they can see each other and other objects in the room.

Scenario C: Collaborative Office Work in the VPR In a small department, people randomly visit different work areas in the office, such as the copier, the file room, the printer, etc. Each worker has a set of supplementary tools that can be invoked on demand, e.g., word processors or database query interfaces. Each worker generally does not need to be intimately aware of the location of other workers in the office, except when there is collaborative work to be accomplished. Worker avatar objects change their state frequently, though other office objects do not tend to change.

Scenario D: Model-Based Virtual Environment In Section 1 we described model-based virtual environments as collaborative environments containing a model to provide context for the collaboration. Multiple workers interact with one another and with isolated parts of a larger artifact (a shared model of work, software methodology, etc.). In this scenario, an avatar may interact with many different components at a relatively high rate, but these pairwise interactions need not be updated at a high frequency at any workstation other than the one manipulating the objects. Assume 5 participants each modify 3 objects at a time. Also assume each object in the environment controls itself, so no application is needed.

Scenario E: A Weather Modeling Application Weather modeling is a highly data and computation intensive process with the end result being weather information displayed in a VE. Each object contains data which can be thought of as one point of data in the large grid of data. In weather modeling, data is broken into small regional subsets and intense processing is performed on that

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: Office Work	100	20	5	5	5	1,000	24	0.017
D: Model-Based VE	20	4	5	5	5	10,000	24	0.01
E: Weather Modeling	10,000	1,000	1,000	5	0	0	0	0.0001

Characteristics for Scenarios used to Evaluate the Gryphon System

Models	Scenario A	Scenario B	Scenario C	Scenario D	Scenario E
System 1	2.0×10^3	42	600	120	1.1×10^7
System 2	5.3×10^5	5.8×10^3	5.3×10^4	4.8×10^5	2.0×10^6
System 3	1.1×10^6	1.1×10^4	8.5×10^4	7.7×10^5	3.2×10^6
Gryphon	20	19	7	8	400

T_{VPR} Comparison

System	Scenario A	Scenario B	Scenario C	Scenario D	Scenario E
System 1	2.0×10^3	42	600	120	1.1×10^7
System 2	4	4	200	40	2.0×10^6
System 3	5.3×10^5	5.8×10^3	5.3×10^4	4.8×10^5	3.2×10^6
Gryphon	20	19	7	8	400

T_{app} Comparison

System	Scenario A	Scenario B	Scenario C	Scenario D	Scenario E
System 1	2.0×10^6	840	3.0×10^3	600	5.5×10^7
System 2	5.2×10^8	1.2×10^5	2.7×10^5	2.4×10^6	2.0×10^7
System 3	5.3×10^8	1.1×10^5	2.1×10^5	1.9×10^6	1.6×10^7
Gryphon	2.0×10^4	380	34	40	4.0×10^3

T_{total} Comparison

Table 2: Gryphon System Performance Comparison

data. After a large amount of processing is performed, data at the fringes of the subsets are distributed to a subset of other processes and then computation continues.

Table 2 summarizes the main result illustrating the viability of using the Gryphon system in each ORB. The top of Table 2 summarizes the scenarios. System 2 (centralized CORBA) and System 3 (distributed CORBA) do not perform well for many cases due to their location transparency policy. The table illustrates the predicted message traffic for the Gryphon system approach in Scenario E, where objects are placed in an application-favored location resulting in a large performance gain compared to all location transparent approaches. Note that in Scenario E, except for the configuration using the Gryphon system, the models do not show the extra messages that occur from the infrequent reads of small portions of the data. For many applications, caching (System 1) can also result in large performance gains while in some applications, Scenario A and E, caching results in unnecessary cache consistency updates. The Gryphon system shows a significant reduction in message traffic except in Scenario B, providing a good illustration of why update policies are important since Scenario B can be interpreted as Scenario A with update strategies already applied. The figure shows significant differences in the message traffic rate; 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 5 scenarios.

3 Real Time Parametrically Controlled In-Kernel Pipes

Continuous media support at the operating system level has focused on ways to provide application code with access and control of kernel-level data, e.g, see [10, 13, 17]. However, these interfaces do not allow the application to influence the way resources are allocated to the components to address application-specific tradeoffs. The second form of dynamic negotiation explicitly addresses the management of the deadline-sensitive aspects of continuous media movement. The *real-time, parametrically-controlled in-kernel pipe* (RT-PCIP) mechanism, used in conjunction with an *execution performance agent* (EPA) tool, manages threads that execute modules in a device-to-device pipeline architecture [35].

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 (e.g., see [6, 13]). 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 (see Figure 3). 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 execution performance agent (EPA) in Figure 3 is to interact with the user-space application and with the modules in the pipeline. Specifically, it (1) provides status information to the user-space program, and accepts parameters that control the

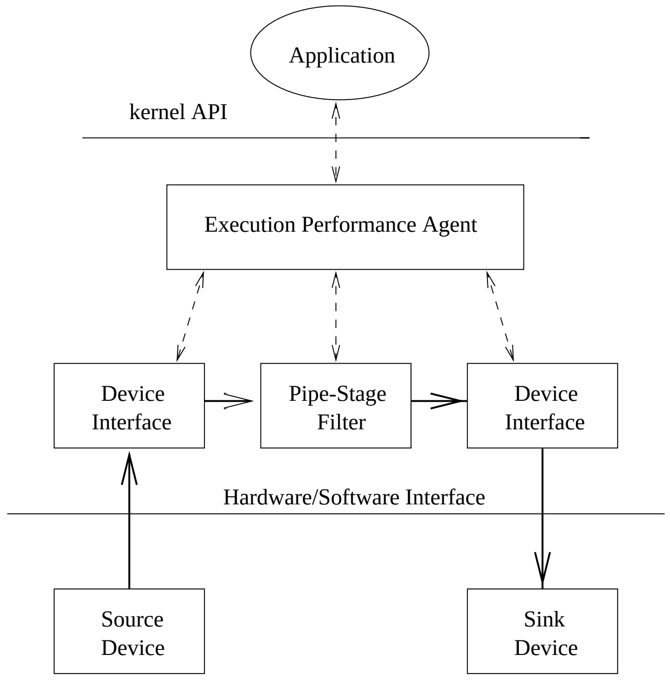


Figure 3: RT-PCIP Architecture

behavior of the filter modules, and (2) ensures that data flows through the pipeline according to real-time constraints and estimated module execution times.

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.

Hard real-time system technology has been developed in domains where the operating system must *guarantee* that each task admitted to the system can be completed prior to a prespecified *deadline* [33]. Such systems are, of necessity, conservative: Task processing 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 less stringent deadline requirements: The threads in a continuous media pipe must *usually* meet deadlines, but it is acceptable to occasionally miss one. In the RT-PCIP, 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, we use 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 [24].

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

3.1 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) [3]. Deadline Monotonic 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. Deadline Monotonic 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

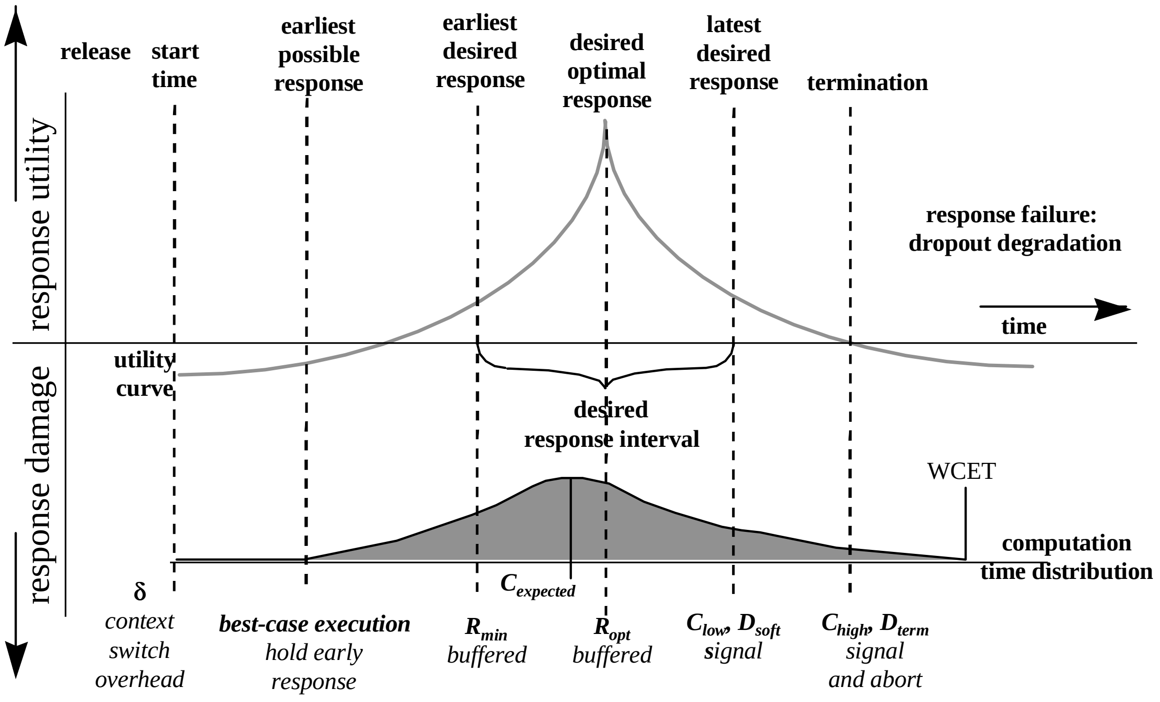


Figure 4: Execution Events Showing Utility and Desired Response

the application (see Figure 4 and [8]). 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 Deadline Monotonic theories do not apply directly to this in-kernel pipeline mechanism, because Deadline Monotonic is appropriate only for hard real-time systems (implying that the worst-case execution time is known). 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 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).

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 Deadline Monotonic is used in part to determine schedulability in the EPA-DM scheduling policy shown in Figure 5; here we assume 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). $I_{max}(i)$

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

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 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 Figure 3. 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 3: 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 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 the VPR. Threads can be created, executed, and monitored in order to deliver predictable, quantifiable performance. Operating system overhead is kept to a minimum, as the amount of dynamic interaction between application code and the operating system is low. The RT-PCIP mechanism will be increasingly utilized in the development of the VPR, as quantifiable real-time movement of data within a node and across nodes is required.

4 Dynamically Negotiated Scheduling

The design and implementation of the RT-PCIP mechanism has shown to be important for flexible and predictable control of device-to-device processing. There are more general cases where there is a need to carefully control the amount of data produced or consumed in applications in the VPR. An application must be able to control the amount and volume of data it produces at any given time, based on the relative importance of the data to the user, the amount of physical resources available to the application, and the importance of concurrently-executing applications. Furthermore, applications must execute according to soft deadlines—applications must produce or consume data in a timely manner, although occasional missed deadlines can be tolerated.

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) [4] 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 a fixed percentage of the CPU can be reserved for the sole use of each application. 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, the application makes a more optimistic estimate of its resource needs, expecting that the operating system will generally be able to meet those needs on demand and will inform the application when it is unable to meet its service assurance.

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 [25] and Rialto [24]; (2) specify a range of resource allocations as in MMOSS [14]; or (3) specify a measure of application importance that can be used to compute a fair resource allocation as in SMART [27]. 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 [25]. 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 applications are constructed to take advantage of such mechanisms without having to participate in a detailed negotiation protocol. The framework is based on the notion of 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 4.3 describes experiments conducted to evaluate the approach.

4.1 Execution Levels

The *execution level* is an abstraction used and defined during the design and implementation of an application that directly responds to changing resource availability (such as reduced network bandwidth or reduced CPU availability). In general, execution levels are used to represent varying degrees of satisfaction using varying amounts of resources. Application execution is characterized by a set of triples:

$$\{Level_i, Resource_i, Benefit_j\}$$

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

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 4: Varying Resource Usage in the VPR

Table 4 shows a set of execution levels in an a VPR application. It illustrates how a simple moving object changes its required processing time over a 4:1 range in 12 execution levels by varying only 3 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 time used as a percentage of the highest level. The OpenGL Performance Characterization Organization [32] has similar benchmark examples showing applications that exhibit 10 different execution levels with CPU requirements varying by as much as a factor of 10.

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

Level	CPU	Benefit
1	1.00	1.00
2	0.80	0.90
3	0.65	0.80
4	0.40	0.25
5	0.25	0.10
6	0.00	0.00

Table 5: Triples for an Example Application

At run-time, each application specifies its maximum CPU requirements, maximum benefit, and a set of triples ⟨Level, Resource usage, Benefit⟩ to the DQM. Level 1 represents the 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 5, 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%).

4.2 Dynamic QoS Manager (DQM)

The DQM dynamically determines a specific allocation profile that best suits the needs of the applications while conforming to the requirements imposed by resource availability, as delivered by the operating system. At run-time, 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 it should be executing. 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 [23].

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.

4.3 DQM Experiments

We used synthetic applications to represent VPR applications to drive the experiments. 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.

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

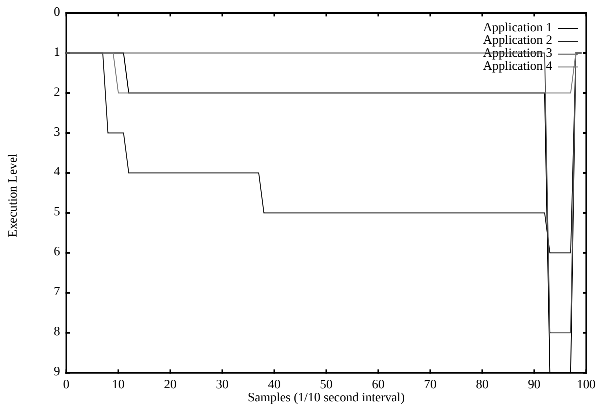
Level	CPU	Benefit
1	1.00	1.00
2	0.35	0.31
3	0.21	0.20
4	0.00	0.00

Table 6: Synthetic Program Characteristics

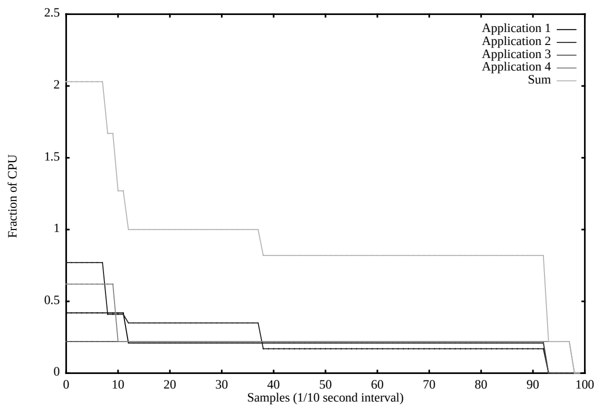
For a given set of applications, data was generated by running the applications and the DQM and recording 100 samples of the current level, expected CPU usage, and actual CPU usage for each application, as well as the total CPU usage, total benefit over all applications, and current system idle time. 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.

Additional insight is gained from a separate simulation tool called the *Decider*; this tool takes the execution level data for a set of applications and determines all of the level changes that would occur with a given decision algorithm in a system with no available resources. This tool simulates starting the applications assuming 100% resource availability, then sequentially adjusting application levels to lower the overall CPU usage until all applications have stopped running. The Decider is used to examine the types of decisions that will be made by each decision algorithm in actual situations of changing resource availability. In particular, this tool gives interesting insight into the stability of each algorithm, where stability is defined to be the distance in level space from one decision to the next. Algorithms that result in a smoother Decider output have greater stability. We believe that stability will prove to be an important measure when in the development of more substantial applications, as it reflects the changes in application fidelity over time under situations of changing resource availability that the user will see when running the applications under this model of application execution.

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 6. There are 4 applications,



(a) Execution Levels



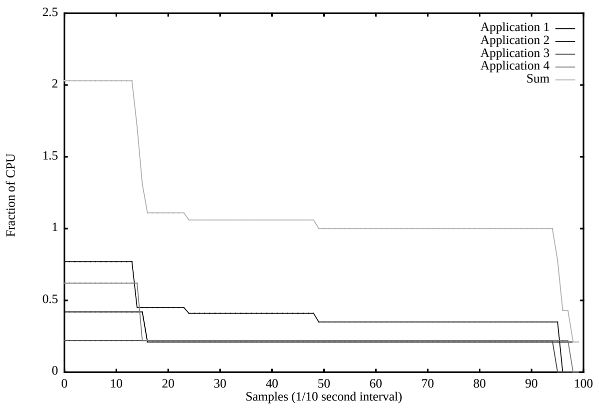
(b) CPU Usage

Figure 6: Performance of Distributed (skip=0)

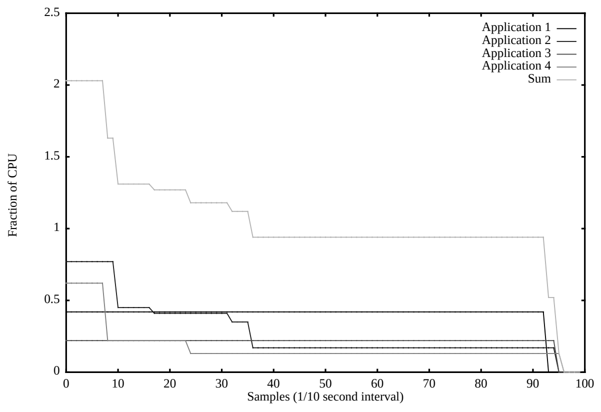
each having between 4 and 9 levels with associated benefit and CPU usage numbers.

Figure 6(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 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 sample until after they have finished executing. Figure 6(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 7(a) shows the CPU usage for the Distributed policy, with a skip value of 2. Using a larger



(a) Distributed (skip=2)



(b) Fair (proportional)

Figure 7: CPU Usage

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 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 6(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 7(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 8 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

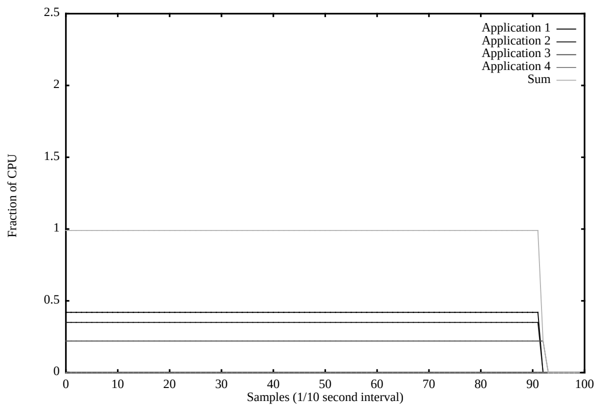
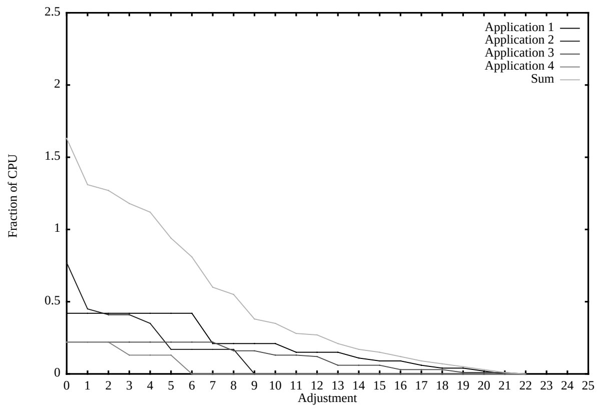
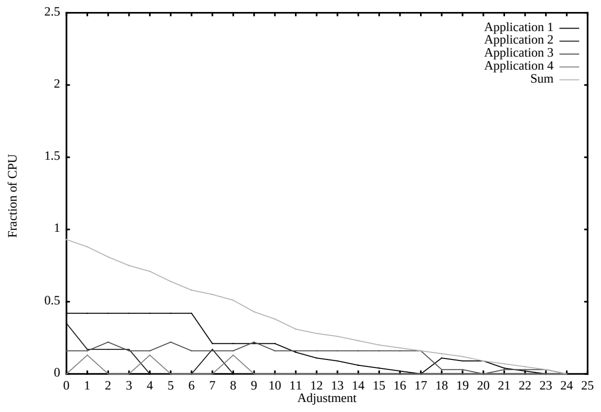


Figure 8: CPU Usage with Optimal



(a) Fair (proportional)

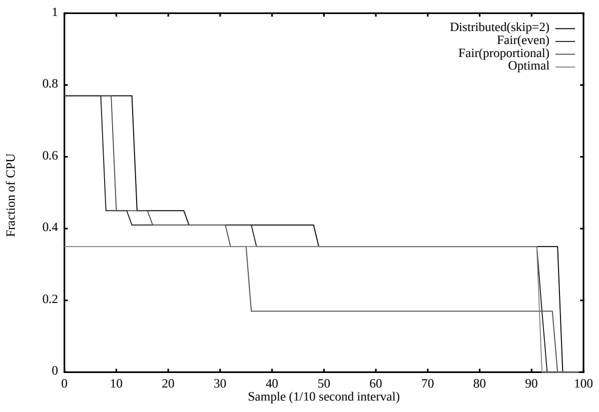


(b) Optimal

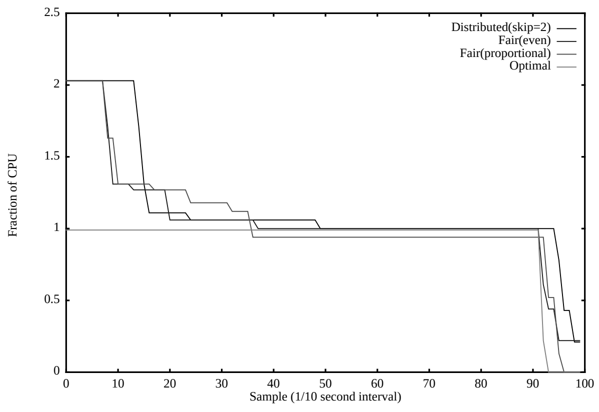
Figure 9: Decider Output

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

Figure 9(a) shows the Decider output for the Fair policy using option 2 (proportional). As stated previously, Decider output with monotonically decreasing levels indicates smooth transitions from one CPU availability to another. For this policy, the degradation shown is relatively graceful. As CPU resources change the level of each application changes slowly and evenly. Contrast this with the results of executing the Decider tool with the Optimal policy, as shown in Figure 9(b). In this case, while the



(a) Application 2



(b) Sum

Figure 10: Performance of Four Policies

sum moves smoothly from 1 to 0, the levels of the individual applications fluctuate significantly as the available CPU resources decrease. Application 2 gets the worst treatment, starting and stopping 3 times.

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. The next step in the continued development of this general software architecture is to directly incorporate these ideas into applications executing in the VPR. The ability to write and execute applications such that they can internally adjust to changing resource availability is crucial to the development of adaptive multimedia applications.

5 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.

In this paper, we have presented new support for multimedia computation, focusing on support for distributed virtual environments. The Gryphon system increases performance for the time-dependent management and delivery of objects in distributed systems. The RT-PCIP mechanism provides quantifiable device-to-device delivery of data within a single node, with minimized overhead due to the operating system. The DQM middleware component, in conjunction with the development and deployment of applications that use explicit execution levels, maximizes perceived user benefit from the execution of a collection of applications under changing resource availability.

The use of these mechanisms will increasingly be important for applications like the VPR, in which a large amount of data must be handled quickly, and worst case allocation of resources is neither feasible nor necessary. The tradeoff decisions between resource allocation and/or location and/or coherence and performance degradation cannot be made centrally, but must involve negotiation with the applications themselves.

Acknowledgment

Authors Nutt, Brandt, and Griff were 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] Denis Amselem. A window on shared virtual environments. *Presence: Teleoperators and Virtual Environments*, 4(2):130–145, 1995.

[2] Special Issue of AT&T Technical Journal on Multimedia, September / October 1995. Nikil Jayant, Technical Reviewing Editor.

[3] 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.

[4] 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.

[5] David A. Berkley and J. Robert Ensor. Multimedia research platforms. *AT&T Technical Journal*, 74(5):34–45, September / October 1995.

[6] 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.

[7] Scott Brandt, Gary Nutt, Toby Berk, and Marty Humphrey. Soft real-time application execution with dynamic quality of service assurance, November 1997. submitted for publication.

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

[9] Special Issue of Communications of the ACM, September 1995. Lucy Suchman, Guest Editor.

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

[11] Hans Eriksson. MBONE: The multicast backbone. *Communications of the ACM*, 37(8):54–60, August 1994.

[12] Lennart E. Fahlen, Charles Grant Brown, Olov Stahl, and Christer Carlsson. A space based model for user interaction in shared synthetic environments. In *Proceedings of Interchi '93*, pages 43–48, April 1993.

[13] 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.

[14] 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.

[15] Jania Gajewska, Jay Kistler, Mark S. Manasse, and David D. Redell. Argo: A system for distributed collaboration. In *Proceedings of the Second ACM International Conference on Multimedia*, pages 433–440, 1994.

[16] Dimitrios Georgakopoulos, Mark Hornick, and Amith Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):117–153, April 1995.

[17] 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.

[18] Pawan Goyal, Xingan Guo, and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 107–121, 1996.

[19] Adam Jonathan Griff and Gary J. Nutt. Tailorable location policies for distributed object systems, December 1997. submitted for publication.

[20] 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.

[21] Special Issue of IEEE Computer on Virtual Environments, July 1995. David R. Pratt, Michael Zyda, and Kristen Kelleher.

[22] Special Issue of IEEE Computer on Multimedia Systems and Applications, May 1995. Arturo A. Rodriguez and Lawrence A. Rowe, Guest Editors.

[23] 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.

[24] 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.

[25] 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.

[26] C. Mohan. Tutorial: State of the art in workflow management system research and products. a tutorial at the ACM SIGMOD International Conference on Management of Data, June 1996.

[27] 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.

[28] 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.

[29] Gary J. Nutt. Model-based virtual environments for collaboration. Technical Report CU-CS-799-95, Department of Computer Science, University of Colorado, Boulder, December 1995.

[30] Gary J. Nutt. The evolution toward flexible workflow systems. *Distributed Systems Engineering*, 3:276–294, 1996.

[31] 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.

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

[33] Krithi Ramamritham and John A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–68, January 1994.

[34] Amit Sheth, editor. *NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-art and Future Directions*. NSF and the University of Georgia, 1996.

[35] Sam Siewert, Gary J. Nutt, and Marty Humphrey. A real-time execution performance agent interface to parametrically controlled in-kernel pipelines. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, pages 172–177, June 1997.

[36] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Inc., 1995.

[37] Harrick M. Vin, Polle T. Zellweger, Daniel C. Swinehart, and P. Venkat Rangan. Multimedia conferencing in the etherphone environment. *IEEE Computer*, 24(10):237—268, October 1991.

[38] WFMC Members. A workflow management coalition specification: Glossary and document of understanding. Technical Report Document Number TC00-0011, Workflow Management Coalition, Brussells, Belgium, August 1994.