The DQM Architecture: Middleware for Application-centered QoS Resource Management

Marty Humphrey
Computer Science and Engineering Dept.
University of Colorado at Denver
Denver, Colorado 80217

Scott Brandt
Computer Science Dept., CB 430
University of Colorado at Boulder
Boulder, Colorado 80309

Gary Nutt*
Computer Science Dept., CB 430
University of Colorado at Boulder
Boulder, Colorado 80309

Toby Berk
School of Computer Science
Florida International University
Miami, Florida 33199

Abstract

Multimedia applications often fail to perform as designed, when resources must be timeshared between multiple applications at run-time. To address this problem, a software architecture is investigated in which a centralized Dynamic Quality of Service Manager (DQM) *mediates resource usage between the operating system and the applications. Applications are written to be able to execute at a range of* operating levels*—a level is defined by a certain amount of resource usage and the corresponding application quality. The DQM lowers the operating level of one or more applications when applications are performing poorly. This action immediately reduces overall resource consumption, which subsequently increases the resource availability to those applications performing poorly. Similarly, the DQM raises operating levels in conditions of underload, thus maximizing resource utilization and collective quality.*

1 Introduction

End-to-end, real-time Quality-of-Service (QoS) assurance in dynamic, heterogeneous distributed systems requires the ability of applications to dynamically negotiate access to resources as workloads and priorities change. The current state of operating systems support for distributed QoS is that multimedia applications (and other time-dependent computations) do not play a large enough role in resource management [2, 8, 13, 14]. Rather, the operating system decides how much CPU time, physical memory, network bandwidth, etc., each application receives. Typically, a multimedia application is written assuming that it will receive a fixed, large portion of resources. If, at the time at which access is granted to a resource, the amount of resource the application receives is less than expected, the performance of the multimedia application may range from a general slowness to the failure of internal time-dependent operations to execute by their deadlines. The problems are both that the resource management system of the underlying operating system does not account for the precise requirements of applications (in terms of performance as a function of resource allocation), and that the applications cannot adequately execute under a range of resource availability.

These problems are particularly evident in complex real-time and multimedia applications such as the *Virtual Planning Room (VPR)* [11], which is a multiperson, distributed virtual environment that supports collaboration among human users in a free-form communication environment. The VPR is unique in that it is designed to have its domain-independent tools be extended with "domain-specific" tools that provide additional application support for the specific problem being addressed by a VPR session. For example, a formal workflow/process modeling system can be embedded in the VPR to focus on group coordination [12].

The system hardware provides a limited set of capabilities to the operating system, which are delivered to the VPR. The VPR supports applications by providing real-time audio and video support, which includes the rendering of objects, and assists in applications' use of system resources. The rendering and behavior of objects places a great strain on the under-

*The research of this author has been supported by the National Science Foundation under grant IRI-9307619.

lying system hardware, which has led us to explore more flexible resource management approaches based on Quality of Service. However, unlike strict QoS approaches where each application must receive a QoS guarantee of its worst-case resource needs before it is allowed to enter the system, we are interested in a solution in which applications cooperate with a QoS Manager to mediate their resource usage in accordance with the currently available system resources. Three fundamental questions arise when considering this approach:

Question 1. How difficult is it to write multimedia (and other real-time) applications using operating levels, where an operating level is defined as a mapping between resource usage and quality?

Question 2. What are the challenges of writing an adaptive QoS Manager that, in order to maximize overall performance of a collection of applications, adjusts application operating levels up and down in response to resource availability?

Question 3. Are the features of general-purpose operating systems sufficient for a robust QoS Manager, or are additional features needed (e.g., beyond the real-time scheduling features of POSIX)?

The contribution of this paper is an initial evaluation that addresses these three questions. Specifically, a compute-intensive multimedia application (the "Spinning Dinosaur") is written to execute at varying operating levels. A centralized *Dynamic Quality of Service Manager (DQM)* executes as a user process; multiple instances of the Spinning Dinosaur provide information to the DQM to dynamically determine the appropriate operating level for each. Uniprocessor results show that this software architecture executing as user processes yields significant control over the performance of the multimedia applications. These results generalize to support the use of the DQM Architecture for distributed real-time applications.

The organization of this paper is as follows. Section 2 discusses the issues in creating applications to work in such an environment. Section 3 further describes the DQM. Section 4 discusses an experiment to verify the basic assumptions of the DQM architecture and investigates its behavior with a sample multimedia application. Section 5 contains a discussion of the results of our experimentation and discusses some future directions for this project. Section 6 discusses projects related to this work. Section 7 contains the conclusions of this paper.

2 Operating Levels

Previous flexible QoS-based systems have assumed that applications can operate adequately given *any* degree of resource allocation (possibly within a prespecified range) [3, 9], and/or depend on real-time QoS facilities within the kernel [5, 9]. The assumption that an application can run adequately given any degree of resource allocation is overly broad and does not generally seem feasible. However, we believe that it is possible to write multimedia applications in such a way as to have graduated operating levels with reduced resource requirements, with lower levels providing correspondingly reduced quality. Operating levels are designed to provide a mechanism for the management of those resource classes that are divisible. Resources such as the microphone are not divisible, and thus are not directly manageable by the notion of an operating level (presumably, the resource is required for all operating levels of the application).

An example of an application that can be written such that it both defines and adheres to a collection of operating levels is a video display application that displays video frames received over a network link. Such an application can be expected to display video frames at a rate of 30 frames/second. However, if it is not possible to achieve this frame rate, the application could back off to 15 frames/second by skipping every other frame and still achieve reasonable quality. Additional discrete configurations, such as 10 frames/second or 7.5 frames/second, are possible.

Objects in the VPR can either individually or collectively use operating levels. Table 1 illustrates how a simple moving object changes its required processing time over a 4:1 range in 12 operating levels by varying only 3 parameters: rendering mode (wireframe, flat shading or smooth shading), number of 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 (FPS) generated and time used as a percentage of the highest level. Quality as perceived by the user has not been determined for these operating levels. This information is not within the scope of this paper; we merely argue that such quality can be determined and thus used for resource allocation decisions.

These examples demonstrate that by varying relatively few parameters, it is possible to modify a multimedia application such that its resource requirements (CPU, in this case) can vary significantly while still maintaining a satisfactory, although reduced, quality. We believe that these examples are representative of a large class of multimedia and other real-time applications that can, with relatively little effort, be modified to exhibit similar operating levels.

Rendering	Lights	Poly.	FPS	% of Max
wireframe	0	1X	12.74	25.0%
wireframe	1	1X	8.94	35.7%
flat	0	1X	8.63	37.0%
smooth	0	1X	7.97	40.0%
wireframe	0	2X	7.7	41.4%
flat	1	1X	6.09	52.4%
smooth	1	1X	5.87	54.3%
flat	1	2X	5.15	61.9%
smooth	0	2X	4.76	67.0%
wireframe	1	2X	4.45	71.7%
flat	1	2X	3.34	95.5%
smooth	1	2X	3.19	100.0%

Table 1: Varying Resource Usage in the VPR

3 DQM

Applications that are capable of executing at more than one operating level provide the possibility of adaptively modifying application resource usage in order to maximize overall performance. The purpose of the DQM is to dynamically mediate between the applications and the operating system in order to keep application resource demands within the limits of the available resources while maximizing quality. It does so by monitoring system load and dynamically adjusting the operating level of each application up or down as necessary.

The decision regarding whether to modify current operating levels of applications is ultimately dependent on the ability of applications to collectively perform as intended. Performance can be defined as the ability of applications to meet internal deadlines. These deadlines can be a function of the operating level. For example, when an application reduces the frequency by which updates are drawn to the screen, the deadline is directly related to the operating level. Deadlines can also be independent of the operating level, such as the removal of network packets from system buffers. Data must be removed from system buffers irrespective of the amount of subsequent processing done on the data once removed from system buffers (which can be a function of operating level).

An important issue is whether or not the DQM must exist as part of the operating system kernel. Because we are concerned about applications in general operating system environments such as Linux (which is the current development platform of the VPR), we would prefer not to depend on the availability of any real-time or other QoS-based facilities in the kernel. The Linux scheduler does not know about application deadlines, nor does it provide any direct mechanism for limiting the amount of CPU used by any particular application to a predetermined operating level. Consequently, in such an environment, the DQM determines system overload or underload on the basis of the measured performance of the applications. The measured performance of each application is based on the queries of the DQM to the underlying operating system (such as CPU usage, frequency of page faults, number of dropped frames, etc.) on a per-application or systemwide basis in combination with direct monitoring of the applications. The direct monitoring of an application is facilitated by the iterative nature of many applications—for example, a moving image must continually be updated on the window. For each iteration, there is a deadline that must be met. The DQM can use the frequency of meeting deadlines both as a justification for increasing resource availability to an application, and a justification for decreasing resource availability to an application.

As mentioned above, the deadline can be a function of the operating level at which an application is operating. Similarly, the operating level may impact the decision concerning whether or not the number/percentage of missed deadlines justifies a change. For example, a high operating level might also require a relatively few number of missed deadlines. Similarly, if an application is executing at a low operating level, it might be allowed to incur a large number of missed deadlines. In many applications, however, the threshold by which to decide to change operating levels is independent of the level. This is the case for the experiment of Section 4.

The design of the DQM does not preclude the use of QoS contracts in which applications are not admitted unless the DQM and the operating system guarantee access to specific resources. Contracts of this kind generally require finer control over resource usage than is possible if the DQM operates as a user process. Other researchers [2, 14] indicate that real-time scheduling assistance such as Earliest Deadline First or Rate Monotonic is needed in order to support QoS contracts from the operating system.

4 Experiment

To begin to evaluate the proposed software architecture, we performed a series of experiments with a simple DQM and a simple graphics application. The application (a Spinning Dinosaur) is derived from a sample application that comes with GLUT, the OpenGL Utility Toolkit [6]. This application was chosen because executing multiple copies concurrently results in each copy degrading to an unacceptable quality as perceived by the user. To be able to describe this performance quantitatively, a clock was superimposed on the dinosaur. An internal deadline

was created—performance is defined as the ability of the application to update the clock every tenth of a second. Thus, the deadline is independent of operating level in this experiment. A screen dump of the dinosaur and the clock is shown in Figure 1.

Figure 1: The Spinning Dinosaur Application

In order to perform experimental studies, the application has to be able to simulate the ability to use different amounts of resources based on the operating level. We added additional computation to the rendering of the application so that the amount of time used was monotonically increasing with the operating level. This made the application easy to modify for the purposes of this experiment. This additional computation represents the increasing requirement for CPU cycles in order to render the object with increasing quality.

A simple DQM was written to start and manage the dinosaur applications. The dinosaur applications and the DQM communicate through shared memory. The information that flows from the DQM to each dinosaur application is the operating level at which each application should execute; the information that flows from the dinosaur application to the DQM is the deadline, enter time, and exit time of each rendering iteration. The DQM determines if the application has met its deadline.

The DQM is responsible for controlling the execution of the dinosaur applications by monitoring their performance, which is defined as the ability of the dinosaur applications to meet their deadlines. The DQM based its decisions regarding modification to application operating levels solely on performance within a time window of either one or five seconds (i.e., cumulative performance was not used as a basis for modifying operating levels).

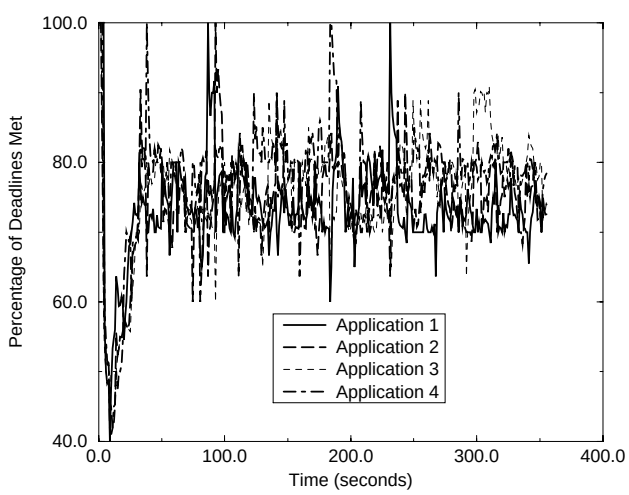
The DQM bases its decisions regarding increasing or decreasing a dinosaur application's operating level on CPU usage, the ability of each dinosaur to meet its deadlines, and the current operating level of each dinosaur. For these experiments, the minimum sufficient percentage of deadlines met to be considered executing well was arbitrarily defined as 70%. That is, it was decided that quality as perceived by the user was based on the ability to provide a timely update to the real-time clock every 7 out of 10 tenths of a second. Any application whose deadline hit rate falls below 70% is considered to be operating poorly. In this case, the DQM will make adjustments to application operating levels until the performance of all applications are within a desired range. The frequency of DQM decisions regarding the operating levels of each application was varied in these experiments. At each decision point, the DQM could either raise or lower a single dinosaur application's operating level (or do nothing). Two policies were investigated for lowering operating levels:

Self The dinosaur application that is performing worst has its operating level decreased by one, reducing its resource requirements and thereby decreasing its deadline miss rate.

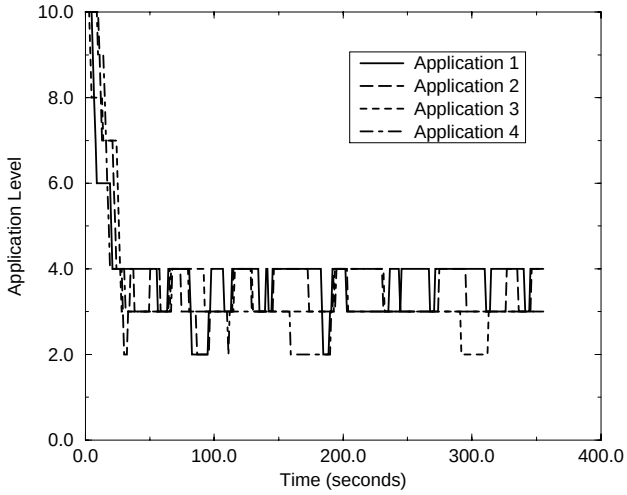
Others The dinosaur application that is performing the best has its operating level decreased by one, making more resources available to other applications and thereby reducing their deadline miss rates.

A single policy was investigated for raising operating levels, which was to increment the operating level of the lowest level application. In order to reduce the instability of the system, a threshold was added: an application's operating level was only increased if all applications were making 75% or more of their deadlines.

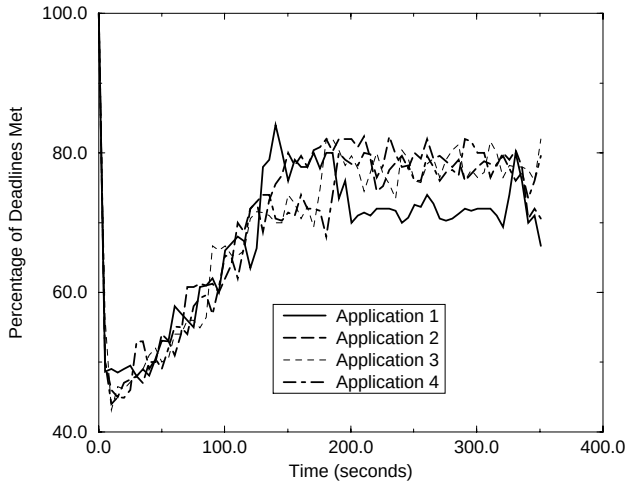
The experiments were conducted on a Dell Dimension XPS Pro 200n, with a 200 Mhz Pentium Pro and 32M RAM. The operating environment was Linux 2.0.0 with Xfree86. Each experiment consisted of executing four dinosaur applications and a DQM. The initial operating level of all dinosaur applications was 10 (the range was defined as 1-10). The duration of each experiment was six minutes, which was observed to be sufficient to stabilize the system. There were two parameters varied: the duration between subsequent decision points by the DQM (either 1 second or 5 seconds), and the policy for the action to take when a dinosaur application was not meeting the minimum acceptable performance (either *self* or *others*).



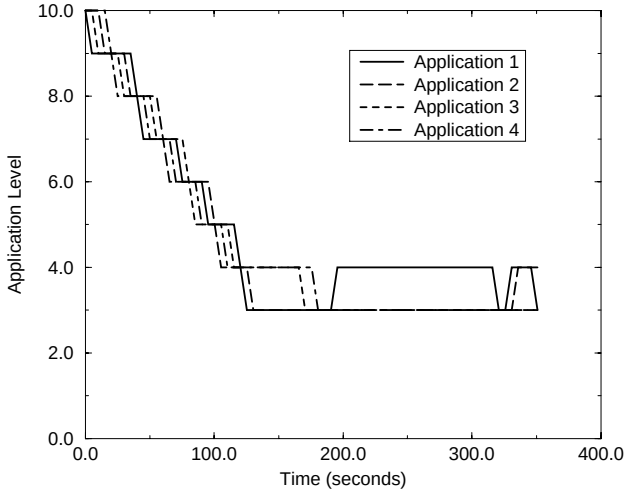
(a) Performance (period = 1 sec)



(b) Operating Levels (period = 1 sec)



(c) Performance (period = 5 sec)

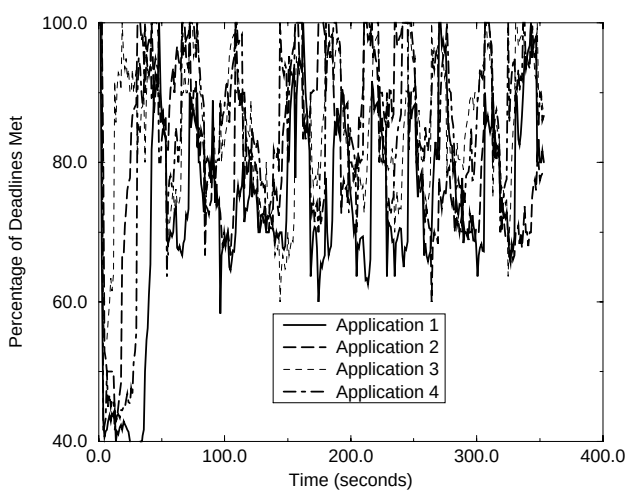


(d) Operating Levels (period = 5 sec)

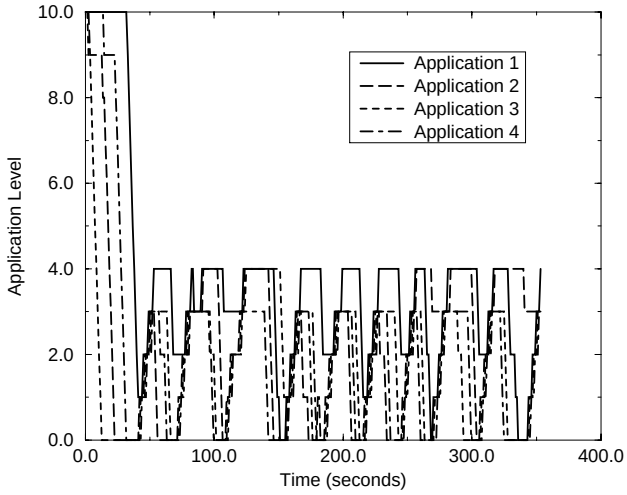
Figure 2: Dynamic Performance of Applications; DQM using "Self" policy

Figure 2(a) shows the performance of each of the four dinosaur applications as a function of time, when the DQM used the *self* policy and the period for decisions by the DQM was 1 second. The performance measured is the percentage of deadlines missed during the five-second window immediately prior to the time. Figure 2(b) shows the operating level of each dinosaur application as a function of time. These plots indicate that, while the operating levels of each application stabilize to generally around 3 for all applications, a high rate of DQM decisions leads to a large variation in performance, as a function of time.

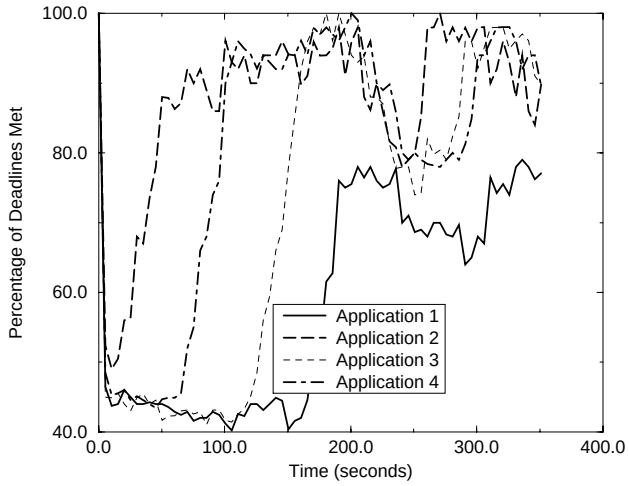
Figure 2(c) shows the performance when the period of the DQM is increased to five seconds. Figure 2(d) shows the operating level of each application as a function of time. When the DQM makes its decisions less frequently, the applications are given time to react to the modifications of the last decision. This results in greater stability in performance as shown in Figure 2(c). Notice however that the responsiveness of the system (as shown by the amount of time required to achieve minimum performance by all applications) is much slower than when the DQM makes a allocation decision every second.



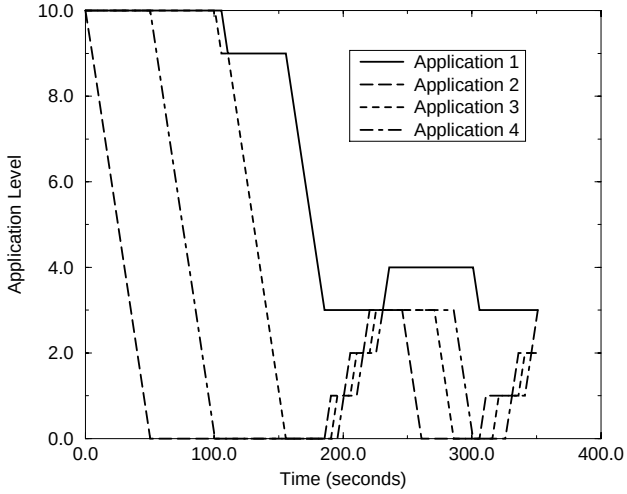
(a) Performance (period = 1 sec)



(b) Operating Levels (period = 1 sec)



(c) Performance (period = 5 sec)



(d) Operating Levels (period = 5 sec)

Figure 3: Dynamic Performance of Applications; DQM using "Others" policy

Figure 3(a) shows the performance of the dinosaur applications when the DQM makes a decision every second, and the DQM uses the *others* policy. Figure 3(b) illustrates the operating levels of each dinosaur application. The performance of each dinosaur application varies greatly under this policy. The explanation for this can be seen in the plot of the operating levels (Figure 3(b)). When one application performs poorly, one of the others is selected as the victim. The benefits of reducing the operating level of the victim are then *shared* by each of the other applications. The result is that, if one application is performing poorly, each of the other applications will have its operating level reduced in turn.

This effect is clearer when the period of the DQM is increased to five seconds: Figure 3(c) shows the performance, and Figure 3(d) illustrates the change in operating levels over time. As was the case when the period of the DQM was five seconds and the policy was *self*, the DQM is slow to react to poor performance. In fact, because the consequences of a decision by the DQM do not directly address the poor performance by a particular dinosaur application, the system is even less responsive.

5 Discussion and Future Work

These experimental results suggest some answers to the remaining two questions posed in Section 1. Specifically, we have demonstrated that it is possible to dynamically adjust application operating levels to maximize collective performance of a set of applications. Furthermore, this was done in a general-purpose operating system without a real-time QoS-based scheduler. We recognize, of course, the preliminary nature of this experiment and its results. These experiments considered only identical applications with uniformly distributed operating levels, no dynamic initiation and termination of applications, and no effects from time-varying loads not under DQM control. Nevertheless, we believe that these experiments show the value of the DQM approach.

The results of the experiments of Section 4 also provide insight into a fourth question:

Question 4. If an application is performing poorly, should the QoS Manager reduce the operating level of the application performing poorly, or should the QoS Manager reduce the operating level of applications performing well?

Figure 3(c) and Figure 3(d) shows the effect of reducing the operating level of one application to increase the performance of another application. The problem with this policy is that *all* executing applications attempt to utilize the newly-freed resource capacity—the result from creating additional resource availability has not been tied directly to the reason for doing so. This is largely a consequence of a DQM that does not have the capability of directly granting and revoking resource allocation for individual applications. Including the DQM as part of the kernel of an operating system with real-time features or otherwise utilizing real-time features are reasonable approaches if this type of allocation policy is to be used.

In the immediate future, we plan to continue our investigations to consider more realistic applications (including the VPR) with non-linear relationships between resource usage and operating level and varying levels of application importance; to develop appropriate protocols for distributed DQM-to-DQM resource reservations and control policies; to deal with resources other than just the CPU; and to examine the information flow between applications and the DQM.

6 Related Work

The goals of the VuSystem project [7] project are similar to the goals of this project. Instead of maintaining a fixed quality of service to applications (determined at call admission), both projects propose to allow users to concurrently execute as many programs as desired, even if an overload condition develops. However, in the VuSystem, under conditions of overload, the user *interactively* decides which applications should be given more resources (and which applications should be given less).

Fan investigates a software architecture in which applications request a continuous range of QoS commitment [3]. In that system, it is assumed that any application can be written in such a way as to work reasonably with any resource allocation within a particular range. Nieh and Lam also take this approach in their SMART scheduler [9]. Tan and Hsu are also investigating this approach for scheduling multimedia applications, utilizing a tighter coupling between the scheduler and the multimedia user applications than our approach [15]. We believe this assumption is impractical for the majority of real-time applications and that the constraint that an application must specify a set of operating levels to be more feasible.

The major difference between the Rialto real-time operating system [5] and this work regards system overload. Rialto uses a QoS-based scheduler to dynamically allocate system resources (in particular, the CPU) based on negotiated QoS guarantees. These guarantees are explicitly enforced by the scheduler. In this sense, Rialto is similar to the work on Processor Capacity Reserves, in which an application can explicitly reserve a portion of the CPU [8]. Our work differs from this in two ways. First, the scheduler used for our studies is a general-purpose UNIX scheduler that does not support any notion of deadlines or QoS guarantees, so our DQM relies solely on application-determined missed deadlines to inform it whether or not the system is overloaded. Second, our applications provide an explicit set of operating levels to the DQM, allowing the DQM to make resource decisions that more closely reflect the actual operation and associated resource needs of the applications.

The *Odyssey* project proposes an initial API as a set of extensions to UNIX for application-aware adaptation [10]. In Odyssey, there is no centralized manager of resources similar to the DQM; rather, an application registers a user-level procedure to invoke if a condition–such as network bandwidth falling below 10 Mb/s–occurs. We believe that without a centralized manager, applications will thrash with each other for resource usage.

Gopalakrishnan uses a *real-time upcall facility* to implement periodic activity [4]. The real-time upcall is similar to the mechanism in the DQM Architecture in which the DQM communicates with the individual applications. The DQM Architecture is a user-level software system for application adaptation and effi-

cient resource management, whereas real-time upcalls requires modifications to the operating system kernel.

The developers of the RTPOOL real-time middleware service are also investigating the use of QoS levels for applications for Automated Flight Control [1]. Their research indicates that levels can be used to express application-level semantics to control how performance is to be degraded under overload or failure conditions. Both projects agree that it is particularly challenging to quantify the perceived utility of users, which is necessary in the use of operating levels.

7 Conclusions

A software architecture has been investigated in which multimedia and other real-time applications are capable of executing at multiple operating levels. A centralized quality of service manager dynamically regulates the operating levels at which applications execute. Experimental results show that this architecture is valuable for regulating the overall quality of applications' execution.

There are two major conclusions that can be made. First, it is not necessary to modify the operating system, or even utilize real-time features of the operating system, in order to regulate applications' collective performance under conditions of overload. The DQM is a user process that monitored the performance of applications and instructed them to change their behavior to reflect resource availability. The simple notion of operating level was sufficient to regulate performance of these applications. Second, while these results are promising, more sophisticated resource management on the part of the DQM may require closer interaction with the operating system. Extending the DQM to also *enforce* resource usage–as might be the case if the DQM were to allow QoS contracts—may require specialized operating system support.

Acknowledgment

The authors would like to thank Jim Mankovich for the generation of the data shown in Table 1.

References

[1] Tarek F. Abdelzaher, Ella M. Atkins, and Kang G. Shin. QoS negotiation in real-time systems and its application to automated flight control. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, June 1997.

[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] Changpeng Fan. Realizing a soft real-time framework for supporting distributed multimedia applications. In *Proceedings of the 5th IEEE Workshop on the Future Trends of Distributed Computing Systems*, pages 128–134, Korea, August 1995.

[4] R. Gopalakrishnan and Guru M. Parulkar. Real-time upcalls: A mechanism to provide real-time processing guarantees. Department of Computer Science Technical Report WUCS-95-06, Washington University, 1995.

[5] Michael B. Jones, Joseph Barbera III, and Allessandro Forin. An overview of the Rialto real-time architecture. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 249–256, September 1996.

[6] Mark J. Kilgard. *Programming OpenGL for the X Window System*. Addison Wesley, New York, 1996.

[7] Christopher J. Lindblad and David L. Tennenhouse. The VuSystem: A programming system for compute-intensive multimedia. *IEEE Journal on Selected Areas in Communications*, 14(7):1298–1313, September 1996.

[8] 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.

[9] Jason Nieh and Monica S. Lam. Integrated processor scheduling for multimedia. In *Proceedings of the Fifth International Workshop on Network and Operating System Support for Digital Audio and Video*, Durham, New Hampshire, April 1995.

[10] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, J. Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[11] Gary Nutt, Toby Berk, Scott Brandt, Marty Humphrey, and Sam Siewert. Resource management for a virtual planning room. In *Proceedings of the 3rd International Workshop on Multimedia Information Systems*, September 1997.

[12] 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.

[13] 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.

[14] Ralf Steinmetz. Analyzing the multimedia operating system. *IEEE Multimedia*, Spring 1995.

[15] Teik Guan Tan and Wynne Hsu. Scheduling multimedia applications under overload and indeterminstic conditions. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, June 1997.