study is required to determine a set of appropriate metrics to apply to the algorithms. In addition, the research described in this paper was done using only a single resource, CPU usage. While this is clearly the most important resource with respect to soft real-time application execution (and the only one examined in most other QoS scheduling research), it is not the only important resource. In particular, it is clear that memory and bus or network bandwidth allocations can greatly impact the CPU usage of various algorithms. In many cases, decreased memory or communication bandwidth results in increased CPU usage and vice versa. While the experiments described in this paper do not address these issues, we believe that the DQM can deal with them directly via appropriate application-specified execution levels.

Our work on DQM policies and additional operating system support continues. In the future we will expand our application repertoire so that we can experiment with a wider class of applications. Next we will design and implement more decision policies within the DQM and experiment with ways that applications can use execution levels to achieve acceptable soft real-time execution. At that point we expect to determine other support that might be provided by the operating system to enable a broader class of policies to be implemented in the DQM.

Acknowledgments

Scott Brandt and Gary Nutt were partially supported by NSF grant number IRI-9307619. Jim Mankovich designed and implemented our main driving application, the VPR, and generated the data shown in Figure 1.

References

[1] A. Burns. *Scheduling Hard Real-Time Systems: A Review.* Software Engineering Journal, May 1991.

[2] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole. A Distributed Real-Time MPEG Video Audio Player. Proceedings of the Fifth International Workshop on Network and Operating System Support of Digital Audio and Video (NOSSDAV'95). April 18-21, 1995.

[3] C. Compton and D. Tennenhouse. *Collaborative Load Shedding*. Proceedings of the Workshop on the Role of Real-Time in Multimedia/Interactive Computing Systems, December 1993.

[4] C. Fan. *Realizing a Soft Real-Time Framework for Supporting Distributed Multimedia Applications*. Proceedings of the 5th IEEE Workshop on the Future Trends of Distributed Computing Systems, pp. 128-134, August 1995.

[5] W. Feng and J. Liu. *Algorithms for Scheduling Real-Time Tasks with Input Error and End-to-End Deadlines*. IEEE Transactions on Software Engineering, Vol. 20, No. 2, Feb. 1997.

[6] H. Fujita, T. Nakajima and H. Tezuka. *A Processor Reservation System Supporting Dynamic QoS Control*. 2nd International Workshop on Real-Time Computing Systems and Applications, October 1995.

[7] D. Hull, W. Feng, and J. Liu. *Operating System Support for Imprecise Computation*. AAAI Fall Symposium on Flexible Computation, Nov. 1996.

[8] M. Humphrey, T. Berk, S. Brandt, G. Nutt. *The DQM Architecture: Middleware for Application-centered QoS Resource Management*. IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, Dec. 1997.

[9] E. Jensen and C. Locke and H. Tokuda. *A Time-Driven Scheduling Model for Real-Time Operating Systems*. Proceedings of the IEEE Real-Time Systems Symposium, pp. 112-122, 1985.

[10] M. Jones, J. Barbera III, and A. Forin. *An Overview of the Rialto Real-Time Architecture*. Proceedings of the Seventh ACM SIGOPS European Workshop, pp. 249-256, September 1996.

[11] M. Jones, D. Rosu, M. Rosu. *CPU Reservations & Time Constraints: Efficient Predictable Scheduling of Independent Activities*. Proceedings of the 16th ACM Symposium on Operating Systems Principles, October 1997.

[12] K. Kawachiya, M. Ogata, N. Nishio and H. Tokuda. *Evaluation of QoS-Control Servers on Real-Time Mach*. Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video, pp. 123-126, April 1995.

[13] J. Kay and P. Lauder. *A Fair Share Scheduler.* Communications of the ACM, 31(1):44-55, January 1988.

[14] C. Lee, R. Rajkumar and C. Mercer. *Experience with Processor reservation and Dynamic QoS in Real-Time Mach*. Proceedings of Multimedia Japan, March 1996.

[15] H. Massalin, and C. Pu. *Fine-Grain Adaptive Scheduling using Feedback*. Computing Systems, 3(1):139-173, Winter 1990.

[16] C. Mercer, S. Savage and H. Tokuda. *Processor Capacity Reserves: Operating System Support for Multimedia Applications*. Proceedings of the International Conference on Multimedia Computing and Systems, pp. 90-99, May 1994.

[17] T. Nakajima and H. Tezuka. *A Continuous Media Application Supporting Dynamic QoS Control on Real-Time Mach*. Proceedings of the Second ACM International Conference on Multimedia, pp. 289-297, 1994.

[18] J. Nieh and M. Lam. *The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications*.Proceedings of the 16th ACM Symposium on Operating Systems Principles, October 1997.

[19] J. Nieh and M. Lam. *Integrated Processor Scheduling for Multimedia*. Proceedings of the Fifth International Workshop on Network and Operating System Support for Digital Audio and Video, April 1995.

[20] G. Nutt. *Model-Based Virtual Environments for Collaboration*. Technical Report CU-CD-799-95, Department of Computer Science, University of Colorado at Boulder, December 1995.

[21] G. Nutt, T. Berk, S. Brandt, M. Humphrey, and S. Siewert. *Resource Management of a Virtual Planning Room*. Proceedings of the Third International Workshop on Multimedia Information Systems, September 1997.

algorithm reaches steady state at about the same time as the Distributed and Fair algorithms, although its allocation is slightly less in this case.

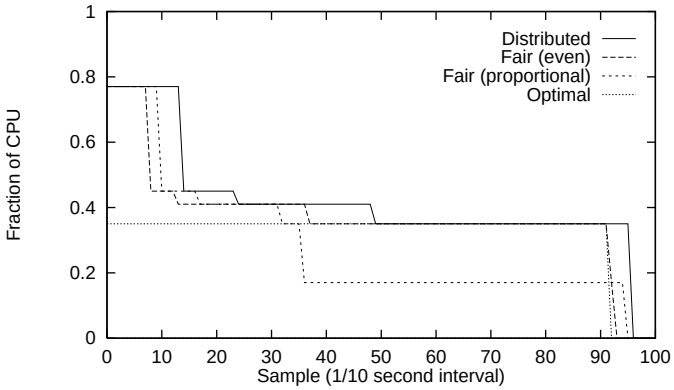

Figure 11: Application 2 with four algorithms (skip=2)

Figure 12 shows the summed CPU usage for the same four algorithms shown in Figure 11. 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

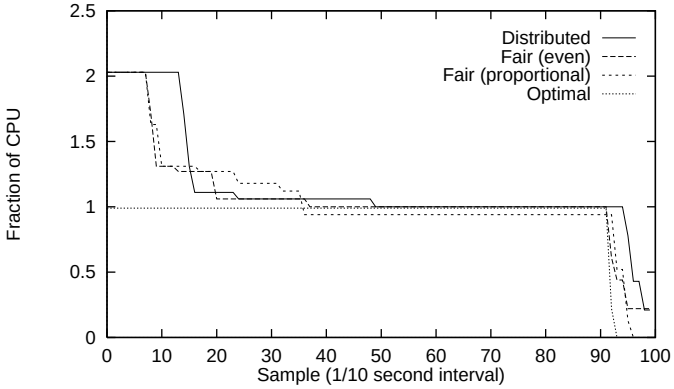

Figure 12: Sum with four algorithms (skip=2)

7. Conclusion

Soft real-time applications are an important, emerging class of applications and there is a class of operating systems evolving to support these applications. A basic premise of this work is that application writers should be prepared to participate in selecting an appropriate strategy for managing resources in the face of oversubscription. The DQM provides a link between the OS mechanism and applications with execution levels. The programmer need only provide a table of levels with resource requirements and benefit, then the DQM uses that information to optimize resource allocation based on cost-benefit analyses.

Flexible QoS systems are also being developed that change the resource allocations given to the applications, either autonomously, as is the case with the SMART scheduler, as a result of negotiation between the OS and the applications, as is the case with Rialto, or as the result of an explicit request by the applications, as is the case with Processor Capacity Reserves in RT Mach. These systems implicitly assume that applications can work properly in an environment where resource applications are dynamic (and perhaps even changing over some continuous range).

We have shown that it is possible to write soft real-time multimedia applications so as to work with varying levels of resource availability. In particular, we provided a specific example from our VPR application, a distributed virtual environment for collaboration that shows how CPU usage for rendering applications can vary by a factor of 4 over a range of 12 execution levels.

We have also developed a prototype middleware QoS resource manager that uses execution level information and dynamically-obtained system and application state information to adjust application execution levels on-the-fly. The DQM demonstrates the feasibility of dynamically adjusting application levels, even in operating systems providing no QoS-based scheduling features.

Several algorithms were examined and results presented showing the performance of these algorithms on a representative set of applications. These results suggest that the algorithm selection is important and may depend on the results desired. The Distributed algorithm is very efficient and performs reasonably well, reaching steady state quickly and resulting in allocations that are fairly uniform across the applications. The Fair algorithm with the even option produces results very similar to the Distributed algorithm. With the proportional algorithm, application benefit information is used to provide each application with its proportional share of the available CPU resources. Finally, the Optimal algorithm produces results that do no have to hunt for steady state.

Additional results were presented that show the decisions the various algorithms would make as available resources change. These results indicate that while the Optimal algorithm does maximize the benefit number, it suffers from instability when resources change. By contrast, the Fair algorithm with the proportional option was shown to exhibit reasonable performance in situations of changing resource availability.

Having provided some examples of multimedia applications that can be written with explicit execution levels, our results show that it is possible to create a middleware QoS resource manager that can manage the resource usage of such applications. The DQM and level-based applications provide a mechanism for implementing soft real-time execution on a generic operating system providing no QoS based scheduling primitives. The soft real-time policy implemented is application specific and will depend on the details of the algorithms developed for each execution level of an application.

The algorithms explored in this paper are not intended to be a complete set, but were selected because of their simplicity or their similarity to algorithms employed in soft real-time schedulers in some of the other systems that we examined. At this time it is not possible to determine which algorithm is the best, but our experiments show that different algorithms have different properties, some, such as utilization and benefit maximization, are obviously desirable while others, such as instability, are obviously not. Further

tralized algorithm makes decisions in an attempt to give all applications an equal share of the CPU. This algorithm generally produces results nearly identical to the Distributed algorithm, as it did with this set of applications. Figure 7 shows the results of running the applications with the Fair algorithm using the proportional option. This version of the algorithm attempts to distribute shares of the available CPU cycles to each application proportional to that application's benefit. Using the previous algorithms the CPU percentage used by all applications was approximately the same. With this algorithm, 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.

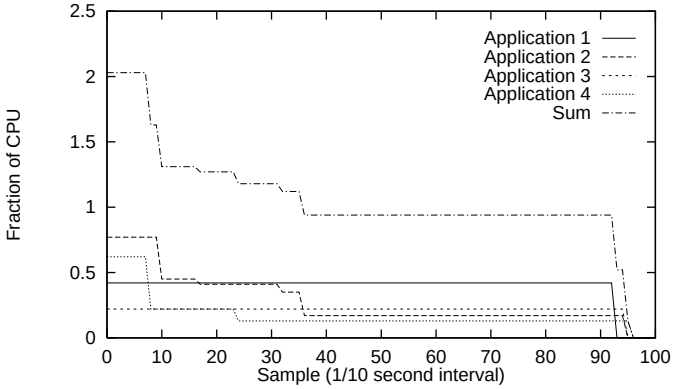

Figure 7: CPU Usage with Fair (proportional)

Figure 8 shows the CPU usage for the applications running with the Optimal algorithm. This algorithm reaches steady state operation immediately, as the applications enter the system at a level that uses no more than the available CPU cycles. This algorithm optimizes the CPU allocation so as to maximize the total benefit for the set of applications, producing an overall benefit number of 14.88 as compared with 13.02 for the other algorithms. Note also that because this algorithm optimizes for benefit and not necessarily for utilization as in the other algorithms shown, it can result in a more stable steady state, yielding no additional deadline misses and requiring no corrections. However, as we shall see, this algorithm is the least stable given changing CPU resources (such as those caused by other applications entering or leaving the system).

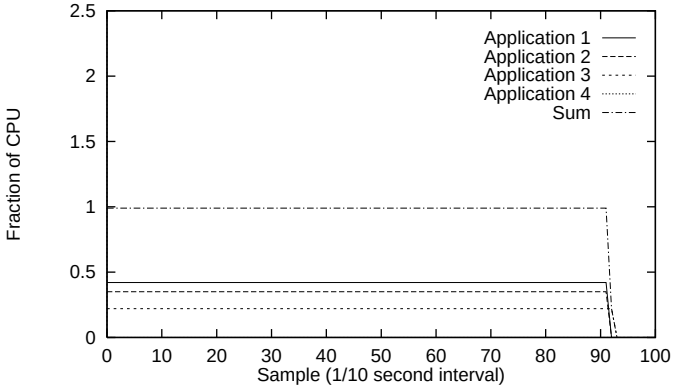

Figure 8: CPU Usage with Optimal

Simulation experiments using the Decider tool give us some insight into the stability of the various algorithms in the presence of changing CPU resources in the system. A Decider output with monotonically decreasing levels indicates smooth transitions from one CPU availability to another. By contrast, Decider output with wildly varying lines indicates an algorithm that will result in unstable application performance over time in the presence of changing CPU availability.

Figure 9 shows the Decider output for the Fair algorithm using option 2 (proportional). For this algorithm, 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 algorithm, as shown in Figure 10. In this case, while the sum moves smoothly from 1 to 0, the levels of the individual applications fluctuate wildly as the available CPU resources decrease. Application 2 gets the worst treatment, starting and stopping 3 times, while the other applications do a little better but still move up and down unacceptably.

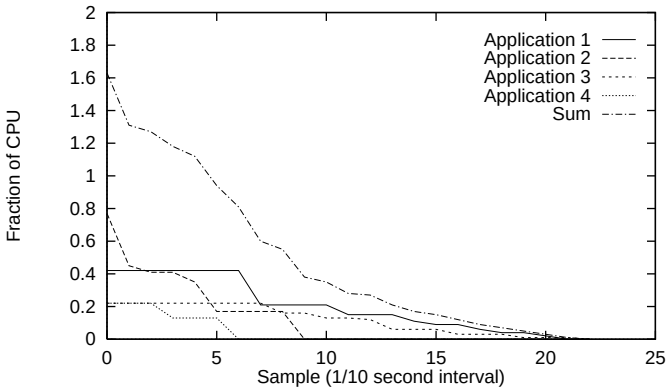

Figure 9: Decider output with Fair (proportional)

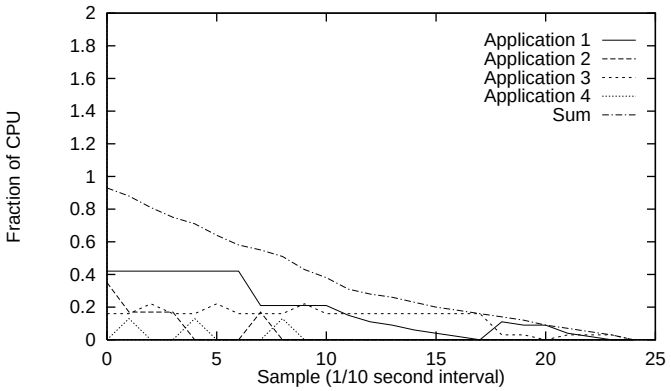

Figure 10: Decider output with Optimal

Figure 11 shows the plots for application 2 with 4 different algorithmic options. This graph summarizes the differences between the various algorithms. The Optimal algorithm 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) algorithms reach steady state at the same value, although they take different amounts of time to reach that state, the Distributed algorithm taking slightly longer. The Fair (proportional)

Application 1
Max Benefit: 8
Max CPU Usage: 0.42
Num 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
Max Benefit: 4
Max CPU Usage: 0.77
Num 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
Max Benefit: 5
Max CPU Usage: 0.22
Num 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
Max Benefit: 2
Max CPU Usage: 0.62
Num 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

Figure 3: Application set with execution levels

means that the level adjustments will be very sensitive to any missed deadlines, changing levels each time a deadline is missed. The execution levels can be seen to change rapidly at the beginning, because we are starting the system 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 2 at the 38th sample due to an additional missed deadline probably resulting from transient CPU load generated by some non-QoS application. 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 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 interval. In other words, we begin sampling before the applications have started executing, and continue sampling until after they have finished executing.

Figure 5 shows the CPU usage for the applications in the same experiment. Here we see that the total 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%.

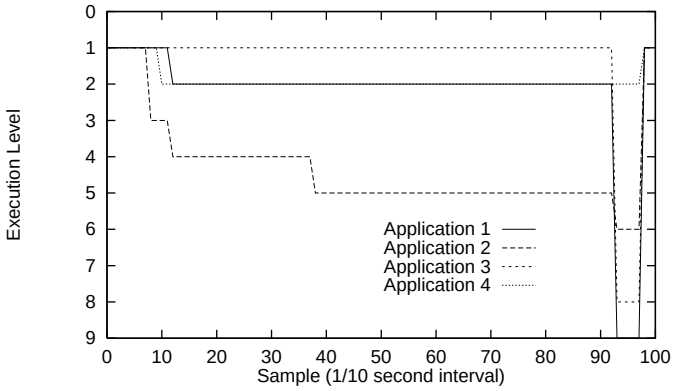

Figure 4: Execution Levels with Distributed (skip=0)

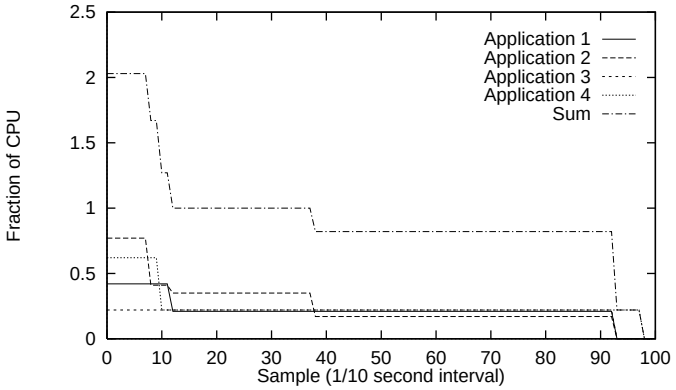

Figure 5: CPU Usage with Distributed (skip=0)

Figure 6 shows the CPU usage for the Distributed algorithm, this time when executed with a skip value of 2. Using a larger skip value desensitizes the algorithm to deadline misses such that a level adjustment is only made for every 3rd deadline miss, rather than for each one. This can result in a longer initial period before stability is reached, but will result in less overshoot as it gives the applications time to stabilize after level adjustments. As we see, stability isn't 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 5 during the level adjustment at sample 38.

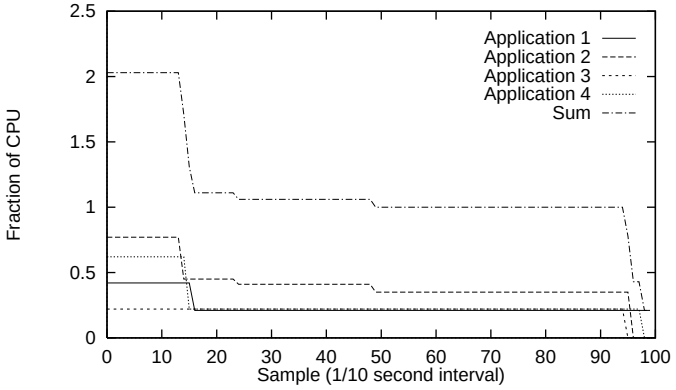

Figure 6: CPU Usage with Distributed (skip=2)

The results of running the applications with the Fair algorithm using the even option are not shown. This cen-

dramatically with a small change in resource availability. As a consequence, this algorithm can result in wildly-fluctuating execution levels for all applications in the system. As a result of these observations, a second option was implemented for this algorithm that restricts the change in level for each application to at most 1, optimizing within this narrow band of application change. While this option will not produce optimal results, we hypothesize that the results will be close to optimal and will result in greater overall user satisfaction as a consequence of the greater application stability.

The final algorithm, *Hybrid*, uses the Optimal algorithm 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 level changes. This algorithm uses a greedy approach in the decision-making, which is hoped will be near-optimal for small changes in resource availability.

For all of the algorithms it is possible to specify a *skip value*. A skip value of n changes the sensitivity of the algorithms so that, rather than responding to every missed deadline, they respond to every n^{th} missed deadline. With a skip value of 0, there is a tendency to overcorrect as a consequence of several missed deadlines occurring at or near the same time. A skip value of 2 or 3 appears to help the algorithms considerably inasmuch as it gives the DQM time to react to the CPU overload situation.

5. Experiment Design

The DQM prototype assumes that the applications know their resource requirements and cooperate to the extent that the execution level information that they present to the DQM is accurate and the applications actually run at the specified level. Under this set of assumptions, we believe that the results of our experiments can be generalized to systems that enforce QoS allocations inasmuch as we have trusted applications that run with exactly the allocation that would be imposed on them in such a system.

We used synthetic applications to ensure that they are well-behaved and to generate the data presented in this paper. The synthetic applications consume CPU cycles and attempt to meet deadlines in accordance with their specified execution levels, without performing useful work. When the DQM recommends a level of execution to an application, we are certain that the application will run at the specified level.

A separate tool, called CreateApps, is used to randomly generate sets of applications, each one exhibiting random total QoS requirement, absolute benefit, number of execution levels, and relative QoS requirements and benefit for each level. The applications generated with this tool exhibit a constant period of 0.1 second. While this does not reflect the complete variability of real applications, it simplifies the analysis of the resulting data. In all other respects we believe that these applications simulate the functioning of real level-based applications. Later versions of the system will include simulated applications with varying period.

For a given set of applications, data were 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 or 100 periods. Our results indicate that this is adequate for observing the performance of the algorithms in a steady state situation.

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 can be expected from each 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 we start running real 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 described in this paper were executed on a 200 Mhz Pentium Pro system running Solaris 2.6. While Solaris does provide some real-time scheduling classes using preemptive fixed priority scheduling, all applications and middleware were executed using the standard UNIX scheduling classes.

The tools described in this paper work with 1-9 applications each having between 2 and 9 levels. For the purposes of generating the data presented in the next section a single representative set of synthetic applications was used. This simplifies comparisons of the results of the different algorithms, but should in no way reflect on the generality of the system described. The application set used for the experiments is shown in Figure 3. This set has 4 applications, each having between 4 and 9 levels with associated benefit and CPU usage numbers. While these applications and levels do not correspond exactly to any real applications, we believe that the ranges of CPU usage and benefit values used adequately test the execution level model and probably vary at least as much as one would find in most real applications.

6. Results

Figure 4 shows the execution levels that result for the given application set when running the DQM with the Distributed algorithm with a skip value of 0. A skip value of 0

The SRL allows an application to specify maximum CPU requirements, maximum benefit, and a set of triples

<Level, Resource usage, Benefit>

As with priority specifications in many systems, 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 is show in Figure 2.

Figure 2 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%). The SRL also provides the application the ability to specify its period and, while running, to determine when deadlines have been missed and notify the DQM of such an event. Finally, the SRL dynamically receives information from the DQM about what level the application should be executing and sets a local execution level variable that the application uses to select the algorithm to execute during each period.

Max Benefit: 6
Max CPU Usage: 0.75
Num 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

Figure 2: Execution Levels with CPU Usage and Benefit

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. The SRL linked into each application notifies the DQM each time an application misses a deadline. CPU underutilization is determined by examine at CPU idle time. In the prototype 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 so as to 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.

The DQM also reads application execution statistics including CPU usage in order to determine if application estimates are correct. Currently, this information is simply recorded for later analysis, but subsequent implementations of the DQM will use this information as well.

A major issue in the design of the DQM is determining the policy for selecting the execution level for each application. Currently the DQM implements four algorithms: Distributed, Fair, Optimal, and Hybrid, a variant of the Optimal algorithm. The algorithms all determine what levels will be selected for each application given the current system resources. These algorithms were selected either because they appear to be obvious solutions to the problem or because they model the solutions provided in other systems.

The *Distributed algorithm* is the simplest policy and is primarily intended to serve as a baseline against which to compare the other algorithms. When an application misses a deadline, it autonomously selects the next lower level. A variation of this algorithm allows applications to raise their level when they have successfully met N consecutive deadlines, where N is application-specific. This algorithm is completely decentralized and does not use the DQM at all. This algorithm could be used in conjunction with the RT Mach reserves scheduling system inasmuch as it does not assume any centralized decision-making or level management but simply allows each application to adjust to the resources that it has available. Were this algorithm to make any sense in an RT Mach system, the admission criteria would have to be changed to include a negotiation whereby an application's execution level is determined before it enters the system. Similarly, this algorithm could be used by applications in the MMOSS [4] and SMART systems to dynamically adjust to the resources that they have been granted.

The *Fair algorithm* is the simplest centralized algorithm that we have implemented. It has two options. In the event of a deadline miss, the *even* option simply 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 algorithm 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. In effect, this algorithms changes the execution level of the application that is furthest from its fair proportional percentage of the CPU. This algorithm approximates the scheduling used in the SMART system.

The *Optimal algorithm* is based loosely on Jensen et al.'s benefit-based scheduling idea[9]. Whereas Jensen et al. attempted to maximize the user benefit by using application deadline benefit curves to maximize benefit for each scheduling decision, Optimal 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. The current implementation uses a brute force exponential-time calculation for determining the optimal allocation. Our experimental results show that the computation time resulting from this approach is trivial for the number of applications and levels we have used. However, should the computation time become significant, it can be reduced using standard dynamic programming techniques.

As expected, this algorithm performs extremely well for initial QoS allocations, but we were surprised to discover that it reacts very poorly to changing resource availability. In particular, the QoS allocations change

Rendering	Lights	Polygons	Frames/ Second	Msecs/ Frame	% of Max CPU Usage
Smooth	On	2X	3.19	313.5	100.0
Flat	On	2X	3.34	299.4	95.5
Wireframe	On	2X	4.45	224.7	71.7
Smooth	Off	2X	4.76	210.1	67.0
Flat	On	2X	5.15	194.2	61.9
Smooth	On	1X	5.87	170.4	54.3
Flat	On	1X	6.09	164.2	52.4
Wireframe	Off	2X	7.7	129.9	41.4
Smooth	Off	1X	7.97	125.5	40.0
Flat	Off	1X	8.63	115.9	37.0
Wireframe	On	1X	8.94	111.9	35.7
Wireframe	Off	1X	12.74	78.5	25.0

Figure 1: CPU Requirements for the VPR application

As an example, consider an analysis of execution levels in our prototype distributed virtual environment, the Virtual Planning Room (VPR) [20][21]. The VPR supports synchronous and asynchronous distributed collaboration, thus it provides an example of a complex multimedia-based application exhibiting natural execution levels. Figure 1 shows the CPU requirements for rendering in a VPR application using different rendering parameters. The table illustrates how an application (a simple moving object in the VPR) can easily change its required processing time over a 4:1 range in 12 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, time per frame, and CPU usage as a fraction of the CPU usage of the highest level. If we were to add benefit to this table, it would reflect the relative quality of the rendered objects at each level

In benchmarking OpenGL applications, the OpenGL Performance Characterization Organization (www.specbench.org/gpc/opc.static/) shows applications that exhibit 10 performance levels with CPU requirements varying by as much as a factor of 10.

Contemporary applications need general flexibility in determining how real-time constraints should be softened. Besides graphic rendering, the application writer may wish to extend the period of an application, implicitly acknowledging the softness of the deadlines themselves. In the video example at the beginning of this section, we assumed that it is not possible to modify the rate at which video frames are entering the system; the application developer might instead choose to display every other frame. This has the effect of doubling the period of the soft real-time application (by ignoring half of the incoming frame data). This solution imposes a discrete step in the execution level resource usage, an important aspect of levels and a motivating factor in their development. In particular, any resource allocation between the one required by the full frame rate level and the half frame rate level would result in wasted resources inasmuch as they are reserved for this application but unused. The execution level application model tells the resource allocator exactly what resources are required for each level so that only the required amounts will be reserved, leaving unusable resource available for other applications.

However, if there is a reason to keep the period the same, e.g. a desire to keep the frame rate constant in the above example, then the execution level model allows the application developer to soften other parameters. For example, the application developer may instead choose to reduce the processing time required during each period by changing the algorithm being used. This has the desired result of reducing the average resource usage of the application while maintaining the desired period. In the video playback example this could be accomplished by subsampling and displaying a smaller image, reducing the color depth of the image, etc. In a graphics application this could be implemented by changing the Level of Display (LOD). Different LODs can dramatically change the amount of resources required to render an image by changing lighting, resolution, shading, rendering mode etc.

These examples demonstrate that, with relatively few parameters, it is possible to modify a multimedia application using levels such that its resource requirements (CPU, in this case) can vary significantly while still maintaining a satisfactory, although reduced, level of performance. We believe that these examples are representative of a large class of soft real-time multimedia applications that can, with relatively little effort, be written with execution levels. Given an infrastructure that can select among these levels, the applications will operate correctly and provide the best performance they can with the resources they are granted.

We conclude that it is possible to write soft real-time multimedia applications in such a way as to at least make it possible to operate in a flexible QoS-based environment. The remaining challenge is to show that these levels can be used by a resource management subsystem in a meaningful way to improve application performance without requiring a specialized real-time QoS based operating system.

4. DQM: A Dynamic QoS Resource Manager

In order to examine the execution level model, we have built a prototype consisting of a middleware application called a *Dynamic QoS Resource Manager* (*DQM*) and a library of DQM interface and soft real-time support functions called the Soft Real-Time Resource Library (SRL). This prototype system has allowed us to experiment with different *policies* (algorithms) for dynamically adjusting levels among a set of running applications with varying numbers of levels with varying resource requirements. Like the flexible QoS systems cited above, the current implementation of our DQM works solely with the CPU resource. We believe that the concepts described in this paper can be extended to encompass other resources such as network bandwidth and memory.

levels with corresponding resource requirements and expected benefit, thus allowing the DQM to make resource decisions that more closely reflect the actual operation and associated resource needs of the applications.

Another body of work that is very relevant to this one is Processor Capacity Reserves [16]. Processor Capacity Reserves can be used by applications to reserve a particular portion of the CPU. Applications are free to increase their portion of the CPU, given available capacity. Real-time processes and non-real-time processes are treated uniformly, because applications merely request their desired CPU portion.

Jensen's work in Benefit-Based scheduling [9] is also relevant to this project. Jensen proposed soft real-time scheduling based on application benefit. Applications would specify a *benefit curve* that indicates the relative benefit to be obtained by scheduling the application at various times with respect to its deadlines. Jensen's goal was to schedule the applications so as to maximize overall system benefit.

In addition, much of the recent research on RT Mach [12] is important for this project. At CMU, efforts are directed at providing end-to-end reservation services [14]. The Keio-Multimedia Platform at the Japan Advanced Institute of Science and Technology (JAIST) is extending RT Mach to support QoS for continuous media streams[17]. Our overall project approaches operating system support for multimedia applications from a different perspective than these projects---rather than determining how to map QoS parameters (such as frame rate for video and sample rate for audio) into operating system mechanisms, we attempt to create an architecture in which there can be mediation between applications, enforcement of applications' registered resource usage, and high resource utilization.

3. Execution Levels

A common assumption in soft real-time scheduling is that real-time applications can be softened simply by missing some or all deadlines -- either by completing the execution late or by aborting the execution entirely when the deadline would be missed. However, this is not the ideal model for all applications. Consider a simple soft real-time multimedia example: desktop video playback. If the video frames are being read from a local disk, then it is a simple matter to delay the read of subsequent frames to account for the delay in displaying previous frames. However, consider the display of a video stream that is being received from a remote system or a physical device such as a camera. In this case, the video source may not support dynamically changing frame rates. Without special operating system support or detailed resource awareness by the applications, displaying frames at less then the incoming frame rate will eventually result in frame buffer overflow or related problems. While this it not a catastrophic failure, it is a failure that goes beyond simply delaying or missing deadlines. A better solution would allow the application itself to respond to the changing resource availability by modifying its algorithm and/or period so as to adapt to the current resource availability.

We propose an application execution model where applications are written with multiple explicit *execution levels*. Each execution level executes a distinct algorithm to carry out the work of the application. Each application is characterized by two numbers, the *maximum CPU usage* and the *maximum benefit*. The maximum CPU usage is the fraction of the CPU required to execute the application at its most intensive resource level. This number is obtained by dividing the cycles/second required by the algorithm by the cycles/second provided by the CPU. In some cases this number may be greater than 1 if the timely execution of the algorithm would take more cycles than the CPU can provide. Similarly, the maximum benefit of the application is a user-specified indication of the benefit provided by executing the application when running at its highest performance level -- analogous to application priority, importance, or utility in other systems. Each execution level is also characterized by two numbers, CPU usage and benefit, where CPU usage specifies the fraction of the maximum CPU usage required by the execution level and benefit specifies the fraction of the maximum benefit provided by the execution level. Relative values are specified because the maximum CPU usage number depends on the system on which the application is being executed and the maximum benefit will be user-specified, but the relationship between the levels is expected to be constant in most cases and specifiable at application development time.

Thus, each application implements a set of algorithms represented by

{<Level, Resource, Benefit>}

where

$Level_i > Level_j \Rightarrow Resource_i > Resource_j$ AND $Benefit_i > Benefit_j$

Implicitly we assume that while all of the algorithms correctly implement the desired application, the benefit of the result degrades with a decrease in execution level. An application can be executed at any of the levels, using corresponding resources with corresponding benefit. The rate at which the benefit degrades is an application specification.

Execution levels represent natural functionality steps in the application. Multimedia applications commonly use execution levels as a matter of standard practice, although operating systems do not normally provide any support for doing so. In effect, execution levels separate soft real-time mechanism from soft real-time policy. Levels and the software to dynamically select among them is the soft real-time mechanism. The details of how the application developer chooses to implement each level is entirely application-specific. In particular, the application developer can choose to implement the levels so as to support a spectrum of soft real-time policies. The soft real-time systems cited above have some of the mechanism for supporting this approach, but they do not provide the part of the mechanism to perform an analysis and then select execution levels based on that analysis. The DQM mechanism performs that function.

other factors besides the frequency of missed deadlines into account, to flesh out the DQM mechanism, and to experiment with this framework to understand the viability of the approach under various policies. To focus on execution levels and the DQM as a fundamental mechanism, our experimentation is currently being conducted using the Solaris operating system. This means that we have not yet exploited the soft real-time aspects that are unique to the target operating systems. Because of the lack of such mechanism in Solaris, our applications are written to be well-behaved, rather than relying on the resource management and enforcement mechanisms that exist in RT Mach, Rialto, and SMART.

Our work demonstrates that it is feasible and perhaps even natural to write multimedia applications with explicit execution levels. The results of our experiments show that given a set of level-based applications, it is possible to create a DQM that dynamically adjusts application execution levels, even in the absence of any underlying QoS or other soft real-time scheduling mechanisms. We present several different DQM decision algorithms that demonstrate the range possibilities inherent in this model and begin to show how level-based applications might work under the various QoS based scheduling mechanisms provided in the research operating systems mentioned above.

Section 2 presents a survey of related work in this area. Section 3 introduces execution levels, and Section 4 describes the DQM. Section 5 discusses a set of experiments we have used to study execution our mechanism, Section 6 presents results of these experiments, and Section 7 contains a summary of this work and the conclusions we have drawn.

2. Background

QoS and soft real-time have generally evolved from traditional real-time systems. Rigorous QoS guarantees require that an application specify its worst-case resource requirements before being admitted to the system; admission is based upon whether or not the resources can be guaranteed. We rely on applications to be able to produce acceptable, but reduced, performance at various levels of resource allocation. Given adequate OS support, such applications could dynamically adjust their processing based on resource availability. Several research programs have looked at various ways to soften QoS requirements.

Compton and Tennenhouse describe a system in which applications are shed when resource availability reduces below an acceptable point[3]. They argue that applications should cooperatively, dynamically reduce resource requirements. Their approach is to be explicitly guided by the user in selecting which application to eliminate.

Research in imprecise computation at the University of Illinois[5][7] examined the idea of having two parts to each task: a required part and an optional part where the optional part refines the computation performed in the required part, reducing the computational error. A modified task scheduler was used to allocate extra CPU capacity towards the optional parts in such a way as to reduce overall computational error.

Massalin and Pu developed the notion of *software feedback*, wherein scheduling parameters are modified based on application specific metrics such as input queue length[15]. This technique has also been applied to application execution[2] such that an application may dynamically modify its processing based upon its performance. The application execution model they describe is strictly best-effort and decentralized, and does not incorporate the notion of any actual QoS guarantees in the operating system environment.

Fan investigates an architecture similar to ours in which applications request a continuous range of QoS commitment[4]. Based upon the current state of the system, the QoS Manager may increase or decrease an application's current resource allocation within this prenegotiated range. Such a system suffers from instability due to the fact that the ranges are continuous and continuously being adjusted, and it lacks a strong mechanism for deciding which applications' allocations to modify and when. It also assumes that any application can be written in such a way as to work reasonably with any resource allocation within a particular range. This assumption is not consistent with the design of the majority of real-time applications.

Nieh and Lam have developed another system based on the Fair Share scheduling algorithm[13] in which applications are allotted a portion of the CPU based upon their relative importance as measured against the other currently executing applications[18][19]. This allotment changes dynamically depending upon the current requirements of all of the currently executing processes and their relative *importances*. Like Fan's system, Nieh and Lam have based their system on the assumption that soft real-time applications can provide reasonable performance with any resource allocation that the operating system decides to give them.

Our goals and approach are most closely related to Rialto [10][11]. A major goal of Rialto was to investigate programming abstractions that allow multiple, independent real-time applications to dynamically co-exist and share resources on a hardware platform. They intended to have a system resource planner reason about and participate in overall resource allocations between applications. The major difference between Rialto and this work is in how we deal with system overload. Rialto has a QoS-based scheduler that dynamically allocates system resources (in particular, the CPU) based on prenegotiated QoS guarantees. These guarantees may be renegotiated, and are explicitly enforced by the scheduler. Furthermore, it is up to the applications to decide how to execute in such a way as to effectively utilize the resources that they have been granted. 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, demonstrating the feasibility of such a scheme on a general-purpose operating system. Second, our applications provide the DQM with explicit sets of execution

Soft Real-time Application Execution with Dynamic Quality of Service Assurance

Scott Brandt, Gary Nutt
University of Colorado at Boulder

Toby Berk
Florida International University

Marty Humphrey
University of Colorado at Denver

ABSTRACT: There is an emerging set of research operating systems that provide specialized support for continuous media and other soft real-time applications. A number of these systems provide QoS scheduling abstractions, some of which may dynamically change the QoS allocations to applications during application execution. The tools and environments that allow application developers to take advantage of these abstractions generally do not exist. This paper describes a dynamic QoS resource manager (DQM) middleware application that abstracts these new OS interfaces so that they are easily used in contemporary application environments.

1. Introduction

Multimedia and other modern applications have created a need for a new class of operating systems support, often referred to as *soft real-time* support. Soft real-time is based on the need for execution deadlines, though the requirements differ substantially from the more traditional (hard) real-time requirements: In a soft real-time application, failure to meet a deadline is not necessarily considered to be a failure of the application or system. The judgement of acceptability of failure in the case of a missed deadline is determined on a case-by-case basis by the application, not by a simple static principle.

Quality of Service (QoS) techniques, originally developed in the context of network bandwidth utilization and packet loss management, have been successfully applied to the domain of soft real-time processing. A QoS system provides a *guarantee* that a certain amount (or quality) of resources will be available when they are needed. In a soft real-time environment, the application needs a *reasonable assurance* that resources will be available on request, rather than an absolute guarantee. In both QoS and hard real-time environments, the system makes strict guarantees of service, and requires that the 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.

Operating systems designers have been creating designs and interfaces to support soft real-time operation. These new OS interfaces allow a process to either a) *negotiate* with the operating system for a specific amount of resources as in RT Mach [16] and Rialto [10][11]; b) specify a *range* of resource allocations as in MMOSS [4]; or c) specify a measure of application *importance* that can be used to compute a fair resource allocation as in SMART [18][19]. These systems all provide a mechanism that can be used to reduce the resource allotment granted to the running applications. Since their average case resource requirements may be significantly lower than the worst-case estimates, resources can be allocated more aggressively, allowing a greater number of soft real-time applications to perform at acceptable levels.

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 [16]. 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).

This paper defines a specific framework in which applications can be 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. Higher quality strategies have higher resource requirements, i.e., the order of solution quality and the order of resource requirements is the same.

Within this framework, the *Dynamic QoS Resource Manager* (DQM) is a policy-independent mechanism that uses resource allocation information from the operating system and execution level information from the community of applications to balance the load, user benefit from running the applications at various levels, and available resources across the collection of applications. In the long term, we see the DQM and execution levels as being a natural environment in which one can leverage the mechanisms provided by systems such as RT Mach, Rialto, and SMART. Our long term goal is to develop such mechanisms -- ones that support soft real-time applications on appropriate operating system facilities -- to complement the OS work. The shorter term goal in this paper is to extend our preliminary work [8] on execution levels to take