A Dynamic Quality of Service Middleware Agent for Mediating Application Resource Usage

Scott Brandt, Gary Nutt
University of Colorado at Boulder
{sbrandt,nutt}@cs.colorado.edu

Toby Berk
Florida International University
berkt@cs.fiu.edu

James Mankovich
Hewlett Packard
jman@cs.colorado.edu

Abstract

High-bandwidth applications with time-dependent resource requirements demand certain resource level assurances in order to operate correctly. Quality of Service resource management techniques are being successfully developed that allow network systems to provide such assurances. These solutions generally assume that the operating system at either end of the network is capable of handling the throughput requirements of the applications. However, real operating systems have to manage many concurrent applications with varying resource requirements. Without specialized support, the operating system cannot guarantee the resources needed for any particular application. In support of these kinds of applications we have developed a middleware agent called a dynamic QoS manager (DQM) that mediates application resource usage so as to ensure that applications get the resources they need in order to provide adequate performance. The DQM employs a variety of algorithms to determine application resource allocations. Using application QoS levels, it provides for resource availability based algorithmic variation within applications and varying application periods. It also allows for inaccurate application resource usage estimates through a technique we have developed called dynamic estimate refinement. This paper discusses new developments in the design of the DQM and presents results showing DQM performance with both real and synthetic applications.

1 Introduction

High-bandwidth applications with time-dependent resource requirements such as continuous media demand certain resource level assurances in order to operate correctly. Quality of Service (QoS) resource management techniques are being successfully developed that allow network systems to provide such assurances. These solutions generally assume that the operating system at either end of the network is capable of handling the throughput requirements of the applications. However, real operating systems have to manage many concurrent applications with varying resource requirements. Without specialized support, the operating system cannot guarantee the resources needed for any particular application.

Applications with such time-dependent resource requirements are often referred to as *soft real-time*. 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 the strictest sense these guarantees are hard; an application is guaranteed to get all of the resources it has requested, assuming the request is satisfiable with the currently available (i.e. uncommitted) resources. Because an application must reserve the resources it needs at the time that it enters the system, the hard nature of the guarantees requires that the application

make a worst-case estimate of its resource needs. However, many such applications can and often do run perfectly well with significantly less resources.

The nature of soft real-time processing is such that the processes do not need such a strict guarantee, only a reasonable assurance that their resource needs will be largely met by the operating system. Operating system researchers have been developing techniques for softening the strict guarantees of such systems so as to allow more applications to run simultaneously, with the expectation that their average case performance will be acceptable and the understanding that worst-case behavior may result in some missed deadlines, loss of data, or other similar consequences. Since these are operating system mechanisms, and since they are based on worst-case estimates, an application is generally required to operate within its worst case estimate (i.e. if the operating system is to make assurances about the use of its resources, it must generally enforce the resource management policy that it adopts).

Operating systems designers have been creating mechanisms to support QoS-based soft real-time application execution. These mechanisms provide a variety of interfaces for determining the amount of resources that will be allocated to an application, allowing a process to a) *negotiate* with the operating system for a specific amount of resources as in RT Mach [19] and Rialto [13][14]; b) specify a *range* of resource allocations as in MMOSS [7]; or c) specify a measure of application *importance* that can be used to compute a fair resource allocation as in SMART [21][22]. These systems all provide a mechanism that can be used to dynamically reduce the resource allotment granted to the running applications. In creating these 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.

We are exploring a middleware solution which takes an approach in which applications cooperate with the operating system in their use of system resources. This approach is in contrast with the operating system approach in which utilization outside the worst case requires enforcement (as opposed to cooperation). This distinction, cooperation versus enforcement, is a philosophical difference in our approach when compared to existing operating system approaches to achieving soft real-time. The assumption of cooperation should be completely adequate for those circumstances in which all processes in the system are owned by a single user or manager (as in a multimedia server or proxy). In a more general system, some level of enforcement will likely prove to be required. One goal of this work is to determine the minimum set of real-time system services required to support soft real-time application execution. In pushing our cooperative approach we have already found that relatively few real-time services are really required.

In previous papers [3][11][24][25] we introduced our notion of QoS levels (called *execution levels)*, a method for managing soft real-time application execution in an environments with varying QoS allocations. With execution levels, each application is constructed using a set of strategies for achieving its goals, ordered by their relative resource usage and the relative quality of their output. We also demonstrated the feasibility of cooperative middleware based QoS management and discussed our prototype middleware execution level based QoS resource manager called the *Dynamic QoS Resource Manager* (DQM) and examined a set of representative QoS allocation algorithms within this context. In so doing, we have demonstrated that a real-time scheduler such as the one provided by RT Mach is not required to support soft real-time execution of QoS level-based applications.

In continuing our research in this area, we have extended the DQM in several ways that bring it significantly closer to our goal of a viable middleware QoS resource management agent. In our earlier work we used synthetic programs to experiment with the DQM, though now our tests are driven using working applications (two mpeg players). We have also enhanced the DQM: we added dynamic application period management capabilities and we have fully generalized the level management by adding level-raising capabilities. Once these capabilities were added, we discovered that middleware QoS management requires accurate application resource usage estimates. To deal with this problem we have developed a technique called *dynamic estimate refinement*. We ran into a number of other issues related to the placement of the DQM as a middleware application, including its lack of fine-grained application resource usage information and its lack of a QoS allocation enforcement mechanism. Taken together, these omissions result in significant differences in the way the DQM must function and in the resulting execution of the applications themselves, as compared with OS based QoS solutions.

This paper presents the results of our continued DQM development. We briefly review execution levels and the design of the DQM. We discuss the issues relating to implementing QoS as a middleware component and to the impact on its overall design. Finally, we present experimental results showing the operation of the DQM with the mpeg applications; the experiments highlight the need for dynamic estimate refinement. Section 1, this section, introduces the paper. Section 2 presents a survey of related

work in this area. Section 3 discusses execution levels and the design of the DQM. Section 4 discusses issues related to raising levels and Section 5 discusses dynamic estimate refinement. Section 6 presents the results of our experiments with real level-based applications, and Section 7 contains a summary of this work and the conclusions we have drawn.

2 Background

2.1 Soft real-time application models

Compton and Tennenhouse describe a system in which applications are shed when resource availability reduces below an acceptable point [6]. 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 [8][10] 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 job scheduling parameters are modified based on application-specific metrics such as input queue length [18]. This technique has also been applied to application execution [5] 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 [7]. 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.

2.2 Soft real-time scheduling

Jensen's work in Benefit-Based scheduling [12] is 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.

Nieh and Lam have developed another system based on the Fair Share scheduling algorithm [16] in which applications are allotted a portion of the CPU based upon their relative importance as measured against the other currently executing applications [21][22]. 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 closely related to those of Rialto [13][14]. 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. Our DQM relies solely on application-determined missed deadlines and OS reported idle time to inform it whether or not the system is over- or underloaded, demonstrating the feasibility of such a scheme on a general-purpose operating system. Second, our applications provide the DQM with explicit sets of execution 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.

Much of the recent research on RT Mach [15] is important for this project. In particular, Processor Capacity Reserves [19] 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. Other efforts at CMU are directed at providing end-to-end reservation services [17]. 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 [20]. 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.

2.3 QoS levels

Tokuda developed a system using QoS levels that are similar to ours [26]. His work was done on RT Mach and used Processor Capacity Reserves and real-time kernel threads to manage the execution of the real-time tasks. There are several differences between Tokuda's levels and our own. Tokuda characterizes the levels according to the type of difference between adjacent QoS levels, temporal or spatial, depending on whether the change was to the period of the application or the spatial resolution of the processed data. Our levels are characterized in the DQM by a single number representing average CPU utilization of the algorithm. Level changes involving Tokuda's temporal or spatial changes are automatically allowed for in this representation, as are changes to the algorithms which affect neither temporal nor spatial resolution. Tokuda also uses simply priority to determine the QoS allocation for each application and does not provide any indication of the relative importance of the individual levels within an application. Our per-application and per-level specifications of importance allow for a more accurate allocation of the available resources among the running applications. In addition to using a more general notion of levels, we have developed a middleware implementation that does not rely on the specialized scheduling and resource management provided by a real-time operating system such as RT Mach with Processor Capacity Reserves. This paper specifically addresses the issues that arise when running a middleware implementation without such specialized support.

Abdelzaher et al at the University of Michigan are using a very similar notion of QoS levels to support automated flight control processes distributed over a pool of processors [1]. They have also extended the concept to apply to network resources [2]. Both of their systems are built on top of RT Mach and rely on its real-time support and scheduling. They provide no information about how QoS levels are determined, nor what to do if they are incorrect.

3 Execution levels and the DQM

Our research has focused on supporting soft real-time processes on general-purpose operating systems. Most soft real-time systems soften the real-time behavior of the applications by moderating the percentage of missed deadlines or the amount by which deadlines are missed, with smaller amounts considered better. This is adequate for the class of soft real-time processes for which missed deadlines are acceptable, but not all such processes fall into this category. For example, desktop playback of a fixed-bandwidth network-based continuous media stream does not allow for all deadlines to be missed by a certain amount because eventually the OS will run out of buffer space to hold the queue of frames that is slowly backing up. In this case a preferable solution would be one such as dropping frames or reducing the amount of processing for each frame so that the hard deadlines (enforced by the arrival of new data) can still be met.

In order to give the policy decisions to the applications, we have developed the execution level application execution model. With execution levels, each application specifies a set of algorithmic modes in which it can execute, along with the computational requirements and the corresponding benefit of running in each mode. A middleware QoS manager provides the mechanism for managing soft real-time by managing the level of each running application through a variety of QoS allocation algorithms.

Specifically, 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. 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 characterized by three numbers, relative CPU usage, relative benefit, and period. Relative CPU usage specifies the fraction of the maximum CPU usage required by the execution level, relative benefit specifies the fraction of the maximum benefit provided by the execution level, and period specifies the amount of time to allocate for each iteration of the algorithm. 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. 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 specific execution levels and corresponding CPU usage and benefit are highly application dependent. The levels themselves will be determined by the goals of the application developer. They will incorporate algorithmic trade-offs that represent a range of output qualities. Traditionally, real-time and multimedia application developers make these trade-offs as a matter of routine, selecting the highest quality algorithm that fits within their known computational constraints. With execution levels the developer can incorporate multiple algorithmic options and allow the system to dynamically determine which one is most appropriate at run-time. The CPU usage numbers are directly measured and represent the average CPU usage for each level of an application. The benefit numbers are determined by the application developer. The details of determining appropriate benefit numbers goes beyond the scope of this research. However, a very large body of research has been conducted in other domains to determine acceptable video frame rates, audio jitter and delay, etc. The results of this research apply directly to the problem of determining the relative benefits of different execution levels.

In order to examine QoS-based soft real-time processing with the execution level model, we have developed a prototype system consisting of a middleware QoS manager 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 several different QoS decision algorithms for dynamically adjusting levels among a set of running applications. Like the flexible QoS systems cited above, the current implementation of our DQM works solely with the CPU resource. However, we believe that the concepts described in this paper can be extended in a straightforward manner to encompass other resources such as network bandwidth and memory

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

<Level, Resource usage, Benefit, Period>

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

Max Benefit: 6
Max CPU Usage: 0.75
Num Levels: 6

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

Figure 1: Execution levels with CPU usage, Benefit and Period

lower execution levels are represented with larger numbers. For example, an application might provide information such as in Figure 1.

Figure 1 indicates that the maximum amount of CPU that the application will require is 75% of the CPU, when running at it's 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 provides the application with the ability to specify the period to be used for each level 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.

The DQM dynamically determines levels for the running applications based on the available resources and the specified benefit of the application. It changes the level of each running application until all applications run without missing deadlines, the system utilization is above some predetermined minimum, and stability has been reached. Resource availability (or the lack thereof) is determined in a few different ways. CPU overload is determined by the occurrence 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 examining CPU idle time. System idle time can be determined in several ways including via the operating system, through the UNIX /proc file system, by measuring the CPU usage of a low priority application, and by taking the complement of the CPU usage measurements (or estimates) of the running applications. If the operating system provides idle time information, this information is the most reliable.

Currently the DQM implements several algorithms with various strengths and weaknesses. A previous paper introduced these algorithms and provided a limited com-

parison of their capabilities [3]. For the purposes of this paper we will use one representative algorithm, *Proportional*, which uses the CPU usage/benefit ratio to determine which application's level to modify in the case of over- or underload. In situations of system overload, it lowers the level of the application with the highest CPU usage/benefit ratio. Similarly, in the event of system underutilization, it raises the level of the application with the lowest CPU usage/benefit ratio. In effect, this algorithms changes the execution level of the application that is furthest from it's fair proportional allocation of the CPU.

4 Measurement issues with raising levels

The DQM uses a static threshold level of measured system idle time to trigger the level-raising algorithm. In order to make good level-raising decisions, reliable resource usage and idle time information is needed. OS-based QoS systems have direct access to this information and make good use of it. Middleware solutions such as ours must make use of the information provided to user-space applications. Both Linux and Solaris (our two OS platforms) provide a /proc filesystem with information about the CPU usage of running applications in user and system space. This information is used by the DQM to trigger the level-raising algorithms. However, there is a problem with this method; the measurements of CPU usage and idle time vary from measurement interval to measurement interval.

Figure 2 shows the execution levels for an experimental execution of the DQM with two synthetic applications run on Linux[1] (with level raising disabled). These applications both have a period of 1/10 of a second, and a maximum usage of 70% of the CPU. The number of levels and relative cpu usage and benefit for each level was randomly generated. The two applications in this example have 8 and 6 levels, respectively. Note that the applications run for 3 seconds (90 periods), then stop.

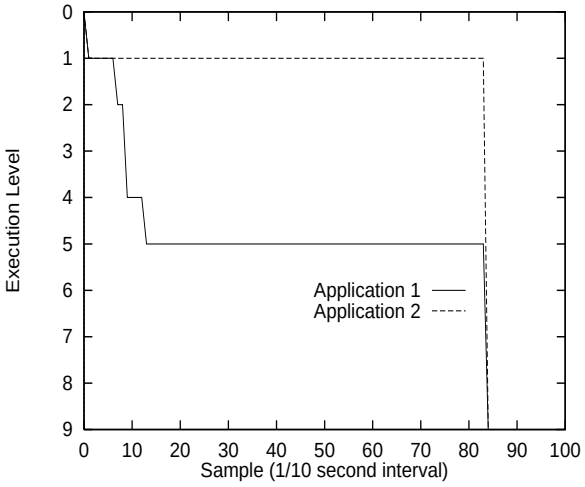

Figure 2: Execution levels for 2 synthetic applications

As can be seen, the execution level of application 1 drops from 1 to 5 between iterations 0 and 13, then remains steady at this level. The estimated CPU usage for application 1 at level 5 is 19%, and the estimated CPU usage for application 2 at level 1 is 70%. This should yield a consistent idle time of 11%. The measurements are made by a separate process that runs at intervals of 1/10 of a second and measures the level and current CPU usage of each application, and system idle time.

Figure 3 shows the measured CPU usage for these same 2 applications, as well as the measured idle time. The measurements can be seen to fluctuate wildly between samples 0 and 13. This is caused by missed deadlines. Even after the levels (and therefore, theoretically, the CPU usage) have stabilized, the measured CPU usage and idle time continues to fluctuate by 10% of the total CPU cycles. However, it should be noted that the sum of the measured CPU usage and the idle time is 100% for every measurement iteration. Measurements on Solaris show similar results.

The observed fluctuation is caused by two factors. The first is phasing of the scheduling of the measurements with the scheduling of each iteration of the applications. This is somewhat avoidable with respect to the CPU usage measurements if we have the SRL measure each application's CPU usage at the end of each period, but is unavoid-

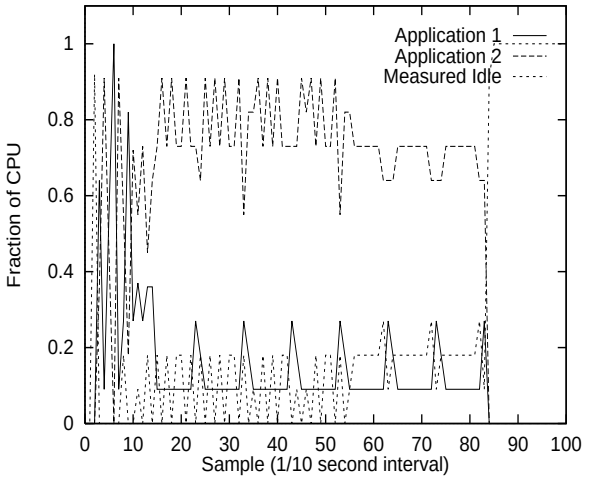

Figure 3: CPU usage for 2 synthetic applications (1)

1. All of the experiments described in this paper were executed on a 200 Mhz Pentium Pro system running Linux 2.0.30. All applications and middleware were executed using the standard Linux scheduler.

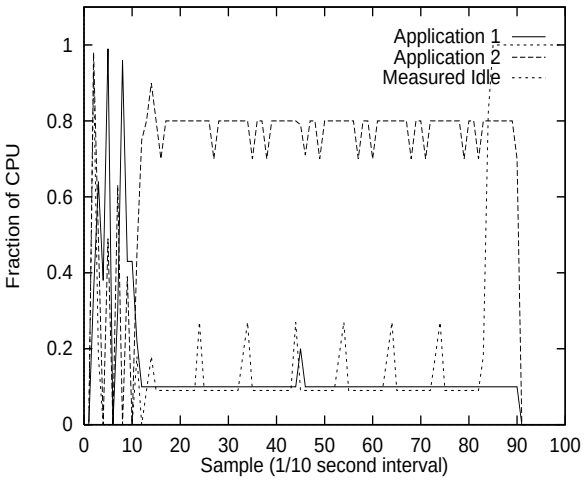

Figure 4: CPU usage for 2 synthetic applications (2)

able with respect to idle time measurement because idle time is a measure of what the applications have not used, and hence there is no perfect time to measure it.

Figure 4 shows the CPU usage and idle time for the same set of applications running with the same algorithm, but this time with the CPU usage measured by the SRL at the end of each iteration of the algorithms (i.e. exactly once per period). The variations seen in this graph are due to the resolution of the CPU usage information provided by the operating system. Both Linux and Solaris provide CPU usage information in hundredths of a second. Reading the usage every tenth of a second gives a granularity of 10%. Note that the idle time is measured slightly less frequently than the CPU usage in this execution, so the graphed peaks in idle time don't match up exactly with the graphed dips in measured CPU usage.

These variations in measured idle time present a problem for middleware determining of when to raise application levels. In particular, the target utilization has to leave at least 10% overhead to account for the measurement inaccuracies.

5 Dynamic Estimate Refinement

Another problem with raising levels occurs when the application resource usage estimates are inaccurate. With underestimates, the system resource utilization will be less than could be achieve with these applications. With overestimates, the system simply may not stabilize at all. This can occur when the estimates indicate that a level change is feasible, but the actual resource usage is such that it is not. In this case, the level will be raised to take advantage of the available resources, then the application will use more resources than it requested. Then, one more applications will miss deadlines, causing application levels to be lowered and the cycle will repeat.

There are several reasons why the estimates might be inaccurate. One reason is that the measurements used to calculate the resource usage were simply faulty. A second possible cause of errors is that the measurements were made on a different machine or architecture than the one on which the application is being run. The estimates could also be inaccurate because of an inherent difficulty in measuring the CPU usage of a particular application, either because of measurement error as discussed above or because of variations in the actual amount of work done per iteration. A final cause for inaccuracies in the CPU usage estimates is interference between running applications, e.g. through sharing a system server or resource other than the CPU.

In order to deal with these issues, we have developed a technique called *dynamic estimate refinement*, in which the CPU usage estimates are continuously adjusted using measurements of the actual amount of CPU time used for each level of each application. This allows the system to adjust to the current execution parameters of each application. The current implementation uses a weighted average of the previous estimate and the current measurement, as follows:

> current estimate = (previous estimate * weight + measurement) / (weight + 1)

This calculation is executed every 1/10 of a second, each time the CPU usage is measured. The estimate is initialized to the values supplied by the application. Smaller weight factors result in quickly adjusted estimates, but also display some sensitivity to transient changes in measured CPU usage. Larger weight factors slow down the adjustment process, but lead to more stable estimates due to the relative insensitivity to transient changes in the measured CPU usage. As with idle time measurement, the accuracy of the measurement is limited by the resolution of the CPU usage information provided by the operating system.

Figure 5 shows the execution levels for the same two synthetic applications seen above, this time with estimates that are 30% lower than the actual CPU usage of the applications. As can be seen, the level of application 1 changes almost continuously as the DQM tries to adjust the levels to use the available resources efficiently. Note that the applications use level 0 to indicate that they are not running, as seen at the beginning and end of the graphed levels for each application. Figure 6 shows the estimated and measured CPU usage for the same execution. The underestimated CPU usage results in the worst case situation described above.

Figure 7 shows the results with dynamic estimate refinement. In this case, there is some level changing at the

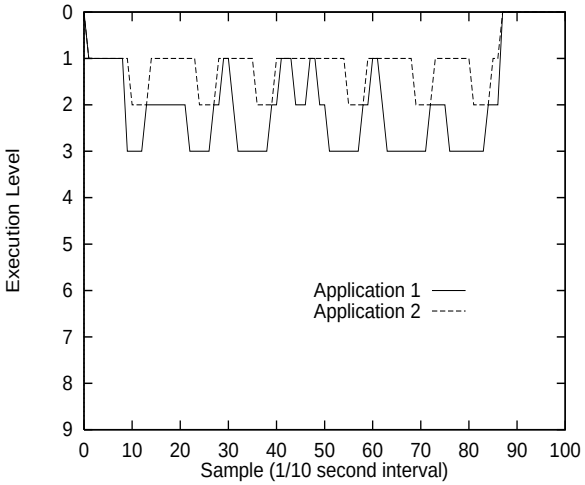

Figure 5: Execution levels with underestimates

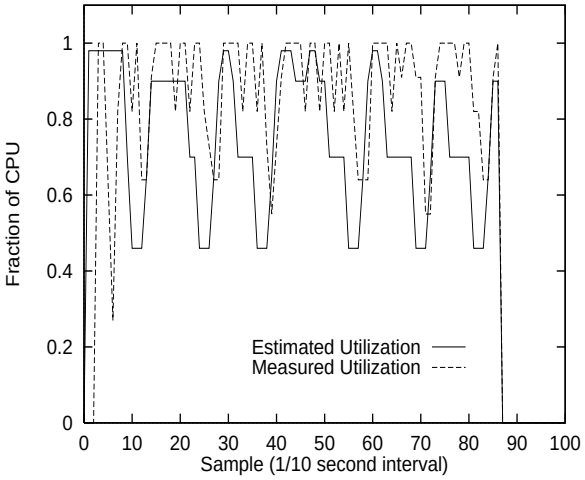

Figure 6: CPU utilization with underestimates

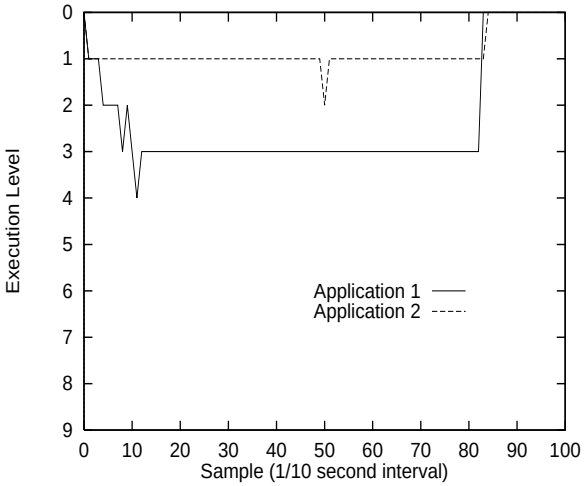

Figure 7: Execution levels with underestimates using dynamic estimate refinement

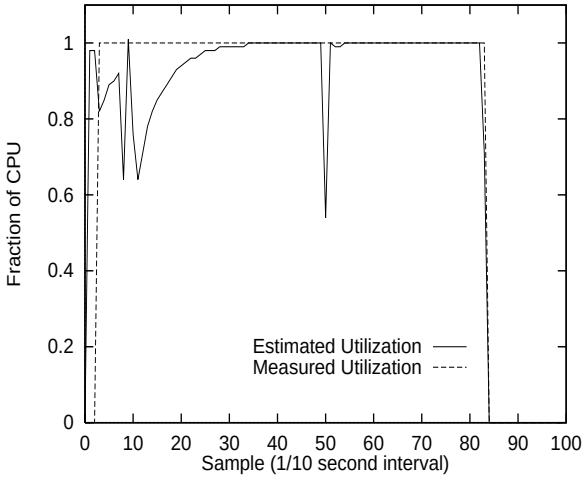

Figure 8: CPU Utilization with underestimates using dynamic estimate refinement

beginning, but the levels quickly stabilize as the estimates are corrected. There is a glitch at sample 50, where the level of application 2 drops to 2 in response to a deadline miss, but it is immediately corrected by the level-raising algorithm. Figure 8 shows the estimated and measured CPU usage for this same run. As can be seen, the incorrect estimates cause some discrepancy between the two numbers, but this is quickly corrected as the applications execute. By sample 33, the estimated and measured CPU usage numbers match exactly.

The graphs in this section were generated with a static raise threshold of 15% and a dynamic estimate refinement weight of 5.

6 Results with real level-based applications

In order to further test the operation of the DQM we developed two level-based soft real-time applications. Both are mpeg players, but they differ in the way that their real-time behavior has been softened. The first, mpeg_size, dynamically adjusts the size of the image displayed on the screen. Since the amount of work is related to how much time is spent drawing the pixels on the screen, this results in a reasonable range of CPU usage numbers over the different levels.

The second application, mpeg_rate, changes the frame rate of the displayed image from 0 frames/second to 10 frames/second. This particular application required no algorithmic changes other than the inclusion of the three SRL functions: dqm_init(), called once at the beginning of the application, dqm_loop(), called each time through the main loop, and dqm_exit(), called at application exit. Figure 9 shows the execution levels for these two applications.

Application 1 - mpeg_size
Max Benefit: 9
Max CPU Usage: 0.89
Num Levels: 8

Level	CPU	Benefit	Period(ms)
1	1.00	1.00	100000
2	0.86	0.90	100000
3	0.73	0.80	100000
4	0.63	0.73	100000
5	0.54	0.65	100000
6	0.46	0.46	100000
7	0.40	0.40	100000
8	0.00	0.00	100000

Application 2 - mpeg_rate
Max Benefit: 9
Max CPU Usage: 0.49
Num Levels: 6

Level	CPU	Benefit	Period(ms)
1	1.00	1.00	100000
2	0.80	0.90	125000
3	0.60	0.75	166666
4	0.40	0.50	250000
5	0.20	0.20	500000
6	0.00	0.00	100000

Figure 9: Execution levels for mpeg_size and mpeg_rate

There are many other ways in which the processing of the mpeg player could have been softened. One obvious method for changing levels is to dynamically change the resolution of the displayed image, displaying a full image each time but using varying size superpixels rather than displaying the full resolution of the image. Depending on the details of the graphics system this technique may or may not result in a large variation in processing time over the various levels. The other obvious technique is to drop frames to achieve a lower CPU usage. Like mpeg_rate, this version would clearly result in the desired variation in CPU usage over the various levels. The choice in which levelization to use is dependent on the goals of the user. If the user wants to see every frame of the image, but doesn't care about the rate at which they are displayed, then mpeg_rate is the one to use. If the user wants the display to run at approximately the rate of the original video, but is willing to sacrifice display resolution or be satisfied with some dropped frames, then these two techniques are good choices.

Figure 10 shows the execution levels that result for these two applications without dynamic estimate refinement and Figure 11 shows the execution levels that result for these two application with dynamic estimate refinement. There are a number of level-changes that occur without dynamic estimate refinement, probably due to

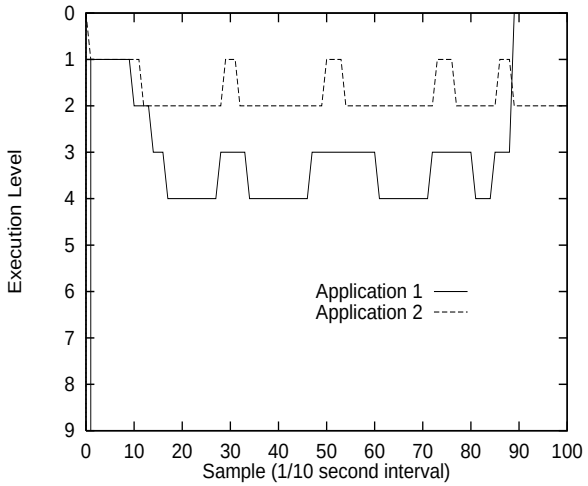

Figure 10: Execution levels for real applications without dynamic estimate refinement

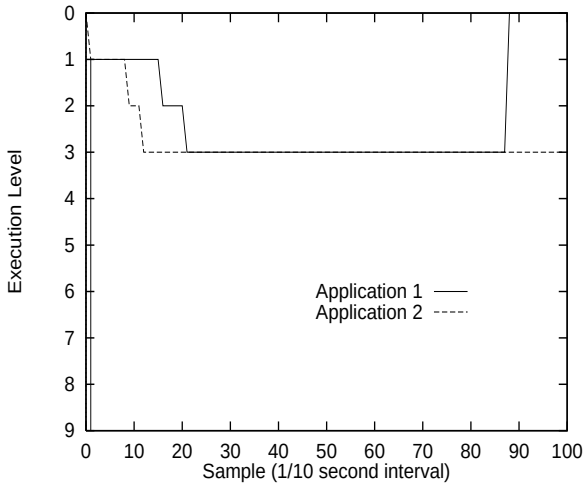

Figure 11: Execution levels for real applications with dynamic estimate refinement

some inaccuracies in the CPU usage estimates. The results with dynamic estimate refinement are significantly better.

Figure 12 shows the execution levels for the same two applications over 10000 samples (just over 15 minutes). There is a small amount of fluctuation in the levels, presumably in response to transient loads in the system, but the levels are generally stable.

7 Conclusion

Soft real-time multimedia applications require certain resource guarantees from the underlying system to be able to provide adequate and stable service. Without such guarantees, application performance varies significantly, and in certain situations applications may fail to run at all. Sev-

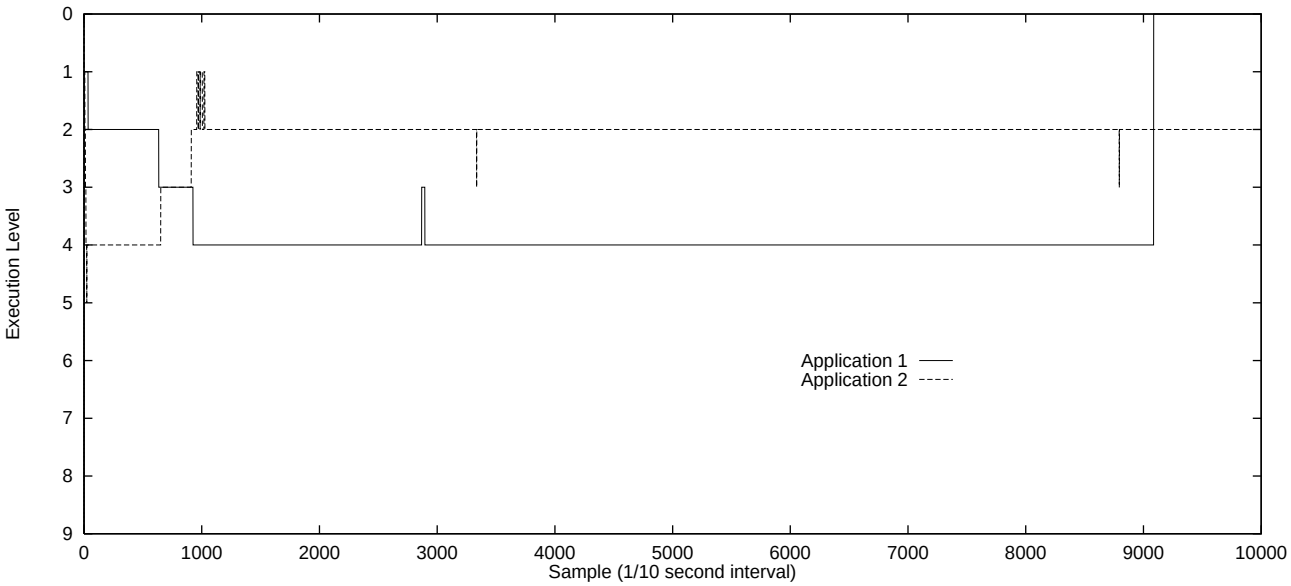

Figure 12: Execution levels for real applications with dynamic estimate refinement over 10000 samples

eral researchers OS based solutions provide (and enforce) QoS resource allocations.

We have developed a QoS based middleware resource management agent called the DQM. The DQM is based on the notion of application specified execution levels that reflect algorithmic modes in which the applications can execute. The DQM uses the execution level information and current system state to dynamically determine appropriate application QoS allocations for the running applications. In this paper we have presented new developments in the design of the DQM, together with significant new results. We have expanded the DQM to work with dynamic application periods and included capabilities for raising application levels. We have also introduced dynamic estimate refinement, a technique for dealing with inaccurate application CPU usage estimates, a significant problem for the level raising algorithms. Finally, we presented two level-based applications, and provided results showing these applications running with the DQM.

In developing the DQM, we have encountered several important issues related to the DQM's role as a middleware component. The biggest issue is that unlike OS-based solutions, a user-space middleware component cannot enforce the QoS allocations. This requires that the applications cooperate, using only the resources that they have been allocated. Rogue applications cannot be managed by the DQM, but we have successfully managed applications whose estimates are incorrect using dynamic estimate refinement. A side effect of the lack of enforcement is that the effect of applications whose usage exceeds their estimates is not confined to themselves. OS based solutions simply cause an application whose usage exceeds its estimate to miss a deadline. Without enforcement, applications whose usage exceeds their estimate may cause a different application to miss its deadline. The centralized nature of the DQM allows global resource level decisions to be made any time any application misses a deadline. Once the estimates have been corrected and levels adjusted appropriately, this is less of a problem. A final issue that we have had to deal with is the lack of detailed CPU usage and idle time information. By averaging the information over longer periods of time, we can get reasonable estimates of resources usage, and by building in a small amount of slack into the idle estimates, we have overcome this problem as much as possible.

8 Acknowledgments

Scott Brandt and Gary Nutt were partially supported by NSF grant number IRI-9307619.

9 References

[1] T.Abdelzaher, E. Atkins, and K. Shin, "QoS Negotiation in Real-Time Systems and its Application to Automated Flight Control," *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, Jun. 1997.

[2] T. Abdelzaher and K. Shin, "End-host Architecture for QoS-Adaptive Communication", *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, Jun. 1998.

[3] S. Brandt, G. Nutt, T. Berk, and M. Humphrey, "Soft Real-Time Application Execution with Dynamic Quality of Service Assurance", *Proceedings of the 6th IEEE/IFIP International Workshop on Quality of Service*, May 1998.

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

[5] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole, "A Distributed Real-Time MPEG Video Audio Player", *Proceedings of the 5th International Workshop on Network and Operating System Support of Digital Audio and Video*, Apr. 1995.

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

[7] 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, Aug. 1995.

[8] 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.

[9] H. Fujita, T. Nakajima and H. Tezuka, "A Processor Reservation System Supporting Dynamic QoS Control", *Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*, Oct. 1995.

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

[11] M. Humphrey, T. Berk, S. Brandt, G. Nutt, "The DQM Architecture: Middleware for Application-centered QoS Resource Management", *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, pp. 97-104, Dec. 1997.

[12] E. Jensen and C. Locke and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems", *Proceedings of the 6th IEEE Real-Time Systems Symposium*, pp. 112-122, Dec. 1985.

[13] M. Jones, J. Barbera III, and A. Forin, "An Overview of the Rialto Real-Time Architecture", *Proceedings of the 7th ACM SIGOPS European Workshop*, pp. 249-256, Sep. 1996.

[14] 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*, Oct. 1997.

[15] 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, Apr. 1995.

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

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

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

[19] 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.

[20] T. Nakajima and H. Tezuka, "A Continuous Media Application Supporting Dynamic QoS Control on Real-Time Mach", *Proceedings of the 2nd ACM International Conference on Multimedia*, pp. 289-297, 1994.

[21] 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*, Oct. 1997.

[22] J. Nieh and M. Lam, "Integrated Processor Scheduling for Multimedia", *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, Apr. 1995.

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

[24] G. Nutt, T. Berk, S. Brandt, M. Humphrey, and S. Siewert, "Resource Management of a Virtual Planning Room", *Proceedings of the 3rd International Workshop on Multimedia Information Systems*, Sep. 1997.

[25] G. Nutt, S. Brandt, A. Griff, S. Siewert, T. Berk, and M. Humphrey, "Dynamically Negotiated Resource Management for Data Intensive Application Suites", *IEEE Transactions on Knowledge and Data Engineering*, to appear.

[26] H. Tokuda and T. Kitayama, "Dynamic QoS Control based on Real-Time Threads", *Proceedings of the 3rd International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Nov. 1993.