Flexible Soft Real-Time Processing in Middleware

Scott A. Brandt
Computer Science Department
Jack Baskin School of Engineering
University of California
1156 High Street
Santa Cruz, CA 95064
sbrandt@cse.ucsc.edu

Gary J. Nutt
Computer Science Department
Campus Box 430
University of Colorado
Boulder, CO 80309-0430
nutt@cs.colorado.edu

Abstract

As desktop computer computational power continues to increases dramatically, it is becoming commonplace to run a combination of deadline-sensitive applications. Despite the proliferation of computational power, the detailed nature of these applications causes new problems for the system resource allocation mechanisms. First, these applications are designed to meet their deadlines as long as nearly all the system's resources are available to them; once the system approaches saturation, the collective applications will fail to meet their deadlines. To aggravate the situation, conventional best effort managers will allocate resources to the competing applications based on a static form of equitability rather than addressing the dynamic relative benefit provided by each application. Second, the applications differ from conventional real-time applications: though members of this new class of desktop applications are sensitive to deadlines, their constraints are non-critical. They are also typically designed to use the full spectrum of functions provided by a general purpose system call interface rather than the more limited interface of a real-time OS. This paper describes a flexible soft real-time system design that addresses these two problems. The CPU scheduling strategy accommodates the community of applications by taking application benefit into account during times of heavy load. Applications are written to use the full interface of a general purpose system call interface, yet the system is able to schedule them according to their deadlines and resource reservation and availability. This paper describes the theoretical foundation of the approach, additional application responsibilities, the design of a middleware system to implement the approach, and then it presents an extensive set of experimental studies that demonstrate the behavior of the approach. The Dynamic QoS Manager middleware (implemented on top of two different versions of Unix), is shown to be highly effective system for supporting contemporary soft real-time desktop applications.

1. Introduction

As the computational power of desktop computers increases dramatically, it is becoming possible to run applications that were previously only feasible on dedicated systems. In particular, high-throughput applications such as video display and processing, real-time hardware control, internet teleconferencing, telerobotics, and virtual reality are being executed on desktop systems at rates formerly achievable only with special-purpose computers. These and other similar applications have relatively high throughput requirements with specific timing constraints on their processing. In particular, they have real-time deadlines. Thus the correctness of their output is determined in part by the time at which it is delivered; a result delivered after its deadline could be considered to be incorrect. These additional timing requirements place a special burden on the operating system; it must not only provide the requested resources to the applications, but must do so on application-specific timing schedules that allow the applications to meet their deadlines.

Traditional hard real-time systems are designed precisely to handle this type of computation. They differ from general purpose, best effort systems in that their purpose is to support applications with deadline requirements. To provide this service, hard real-time systems generally rely on *a priori* knowledge about the applications' worst case resource needs to determine a feasible schedule or a set of priorities that will guarantee that the deadlines will be met. To determine the worst case, it must be possible to bound the resource usage of an application prior to the time it begins execution. This means that the application cannot use system services that do not have strict bounds on the time to perform the service and the amount of resources used to provide the service, i.e., the system call interface for a general purpose operating system such as Unix cannot always make such worst case usage assurances. Some recent desktop operating systems (Windows NT, Solaris, etc.) incorporate preemptive priority-based real-time scheduling. This type of scheduling can support a small set of real-time applications with carefully measured and constrained resource usage and little or no dependence on other system services or interaction with other processes. Because the real-time processes are executed at a higher priority level than the rest of the processes in the system, all other processes, including system daemons, must execute only in the CPU cycles not used by the real-time processes. For this reason, when this scheduling class is available at all it may not be available to most users.

In contrast to traditional real-time applications, this emerging class of desktop applications does not need strict real-time performance guarantees. Instead, these applications need only a reasonable assurance that their resource needs will be met by the system. Failure to meet a deadline does not constitute system or application failure, it is simply less satisfactory to the user. Systems that support this type of application are referred to as *soft real-time systems*.

Operating system researchers have been studying techniques for softening the strict guarantees of real-time systems so as to allow more applications to run simultaneously, with the expectation that their average case performance will be acceptable and with the understanding that worst-case behavior may include some missed deadlines,

loss of data, or other similar consequences. Several researchers have focused on developing OS mechanisms that provide soft real-time application support by allowing applications to miss some or all of their deadlines (see Related Work). These systems generally rely on scheduling primitives that use application-supplied information concerning each application's CPU and timing requirements. When all of the application resource requirements cannot be met, these systems tend to reject additional applications or stop low-priority tasks that are already executing [10][25]. This method does not support a principle of graceful degradation wherein all user requests are honored and resources are allocated to the applications in an equitable manner. Other systems reduce the resources provided to each application based on formulas of resource need, application importance, etc. [11][18][21][28].These systems rely on the applications themselves to adjust their processing to fit within the resources they have been allocated. However, none of these latter systems provide a model for how the applications can reduce their resource requirements to function within a less-than-optimal allocation.

1.1. Approach

Soft real-time support can be provided to a community of cooperating applications by incorporating a *QoS manager* that optimizes resource allocation according to the global benefit of each application to the community (as well as the currently available resources). Each application provides the QoS manager with a model of its application benefit, processing time, and period, i.e., the model is a deadline-sensitive model [29]. Applications that provide relatively high benefit to the user at any given instant should receive correspondingly more resources than ones that provide lower benefit at that same instant After evaluating relative benefit and resource requirements, the QoS manager can define a strategy whereby applications with relatively high global benefit should be allocated a correspondingly larger proportion of system resources than applications with relatively low global benefit.

The QoS manager depends on an application agent (e.g., the application itself) to provide a specification of its benefit versus resource usage. Various researchers agree that *Quality of Service (QoS) levels* can be used to define such a specification. QoS Levels have evolved over a period of time with contribution by various researchers, though Tokuda and Kitayama published the seminal idea in 1993 [35]. In this study, Tokuda and Kitayama's QoS Levels are extended and then used both as part of a mechanism for providing soft real-time processing on top of a general-purpose operating system, and as a model for applications to dynamically specify their resource usage in soft real-time environments. The extended QoS Level model allows applications to define the importance of various patterns of missed deadlines and other real-time processing constraints. By allowing applications to determine which real-time processing constraints to change as they define their resource usage, our QoS Levels effectively separate soft real-time policy from soft real-time mechanism so that the system can provide the mechanism and the applications themselves define their own policies.

With QoS Levels, applications are developed with multiple modes in which they can execute. Each mode provides a different output quality and consumes a different amount of resources. An application's modes are ordered

by the output quality they provide and the resources they require. Given the levels for a set of applications, a resource manager running as middleware can dynamically allocate the available resources among the applications with the goal of maximizing overall user satisfaction.

We have developed the *Dynamic QoS Manager* (*DQM*) as a middleware mechanism that operates on the collective QoS Level specifications. It analyzes the collective optimization functions provided by the community of processes to determine its allocation strategy[1]. Once the DQM determines how resources should be allocated, it determines the level at which each application should operate in order to optimize the global benefit. In the DQM middleware solution, each application is informed of the level at which it should execute to maximize global benefit, but the DQM does not ensure that the application will actually execute at the recommended level. Thus the DQM requires that the applications be well-behaved to achieve maximized behavior (OS support is required to assure cooperation among ill-behaved applications, though our experiments observe situations in which some applications do not use QoS Levels -- see Section 6).

1.2. Issues with the Approach

Extended QoS Levels and the DQM are a proof of concept that a cooperating community of applications can use an OS with a best effort CPU scheduler, yet achieve an effective soft real-time strategy that is well-suited to a desktop computer with a repertoire of applications with processing deadlines. In designing and analyzing the approach, there were several critical issues that guided the research:

- Is QoS Level soft real-time feasible?
- What is the effect of different resource allocation algorithms?
- Is a complete middleware-only system possible?
- What performance can be expected from such a system?

This paper describes the approach, then addresses each of these issues to demonstrate the feasibility and utility of a middleware QoS manager for flexible soft real-time processing.

1.3. Outline of the Paper

Section 2 discusses related work in OS support for soft real-time processing and QoS Levels. Section 3, presents a detailed discussion of our notion of QoS Levels. Section 4 presents the details of the DQM. Section 5 presents results from investigations of the system operation with a variety of representative resource allocation algorithms. Section 6 discusses the issues in implementing a middleware-only QoS Level soft real-time system. Section 7 summarizes the work, presents the conclusions, and discusses future directions for this research.

---

1. We note that any system that uses QoS Levels or similar functions to specify global benefit introduces a new problem. To optimally determine how to allocate resources using such functions, the QoS manager must solve an NP-complete problem [16]. Elsewhere we elaborate on the implications of using QoS Levels [4].

2. Related Work

Jensen *et al.* proposed a soft real-time scheduling technique based on application benefit [19]. Each application specified a *benefit curve* that indicated the relative benefit to be obtained by scheduling the application at various times with respect to its deadlines. The goal was to schedule applications so as to maximize overall system benefit. While this approach is intuitively very appealing, it is computationally intractable. This work evidently led to QoS Levels as an alternative, discrete representation of benefit.

Tokuda and Kitayama [35] developed QoS Levels as a mechanism to be used with RT Mach [36] and processor capacity reserves [25]. There are several differences between their QoS Levels and those used in this study. The original QoS Levels characterize a particular level 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 QoS Levels are characterized by a single number representing average CPU utilization of the algorithm. Level changes involving their 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 and Kitayama also use simple priority to determine the QoS allocation for each application, which does not define the relative importance of the individual levels within an application. Finally, this work did not result in a working system.

Rajkumar *et al.* have developed a theoretical QoS model called Q-RAM [30][31] that is similar to the one presented here. Q-RAM uses continuous benefit functions to specify application benefit as a function of resource allocations. Rajkumar *et al.* envision Q-RAM simultaneously managing several applications, each with benefit functions on several different resources (such as CPU, memory, and network bandwidth). They have developed an algorithm for the case where the benefit functions are continuous and convex [30] (using the same algorithm as [16]), and an approximation for the case where the benefit functions have discontinuities and are almost convex [31]. Lee extended Rajkumar *et al.*'s work to address applications with discrete benefit functions [24], a model that is essentially the same as QoS Levels. This work is relevant as a theoretical model, but the results are largely untested in an actual soft real-time system.

Abdelzaher *et al.* use a similar notion to 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 do not provide information about how QoS Levels are determined, nor what to do if they are incorrect.

Fan describes an architecture similar to the DQM in which applications request a continuous range of QoS commitment from a centralized QoS Manager [11]. Based on 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. There is apparently no implementation of the system at this time.

Compton and Tennenhouse describe a system in which applications are shed when resource availability falls below a threshold point [10]. Their approach is to be explicitly guided by the user in selecting which application to eliminate. This approach is similar to QoS Levels in that the applications cooperate in their usage or non-usage of the resources, but by limiting the applications to two levels -- running or not running. This approach does not address deadlines.

Liu *et al.*, define imprecise computation [12] where each task has a required part and an optional part. 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. Tasks have 3 levels: running, running with more computational error, and not running. Applications do not miss deadlines, and there is no deadline miss detection or notification.

Kravets *et al.* have developed a notion similar to QoS Levels for use with networked applications [23]. Their applications use a payoff function specifying the quality of the output as a function of network bandwidth received, and a dynamic resource manager allocates network resource to applications in an attempt to maximize overall benefit.

Steere *et al.* have developed a proportional resource allocator for scheduling threads that dynamically determines the proportion of the CPU to allocate to a given thread and an appropriate period for the thread based on the progress that the thread makes [33]. Real-time threads have static period and proportion while non-real-time threads have dynamic period and proportion. The dynamic period and proportion of the non-real-time threads are changed based on a heuristic goodness based on the system-determined progress of the application. Their notion of progress is based on built-in knowledge about certain classes of applications, including client-server, interactive, I/O intensive, and other. By extending the notion of period and proportion to non-real-time threads, this work provides a framework for integrating real-time and non-real-time processes. However, their work has many aspects that make it unsuitable for general soft real-time processing. First, their notion of progress is inadequately developed to work for general applications that do not fit within their classes. Second, they have no deadline monitoring capabilities that would allow for deadline misses to be addressed in the case of system overload with soft real-time applications. Third, they have no notion of importance or benefit that would allow the system to decide among competing soft real-time applications in cases of system overload. Finally, while they claim to use control theory in determining their allocations, their results show significant and problematic instability in the resource allocations provided to the applications, an issue that they have not yet addressed.

There is considerable complementary work in which the OS provides an explicit scheduling approach and a mechanism to enforce cooperation among the processes (e.g., see RT Mach [25], Rialto [21], SMART [28], MMOSS [11], and Epiq [26]). However, the DQM approach is a middleware approach that relies on QoS Levels, so it differs substantially from this work. Besides Abdelzaher *et al.* and Kravets *et al.*'s work (mentioned above) there is also other relevant QoS work in networks (see for example RSVP [3], DiffServ [27], work published in IWQoS (e.g. [13]), and work from the Internet2 workshops (e.g., [7][34])).

3. QoS Levels

Tokuda and Kitayama distinguish between static and dynamic schemes for specifying QoS Levels. In the static scheme, the QoS Level to be used is determined when an application starts, but in the dynamic scheme it can be adjusted as the application runs. A QoS Level (for continuous media) is expressed with temporal and spatial characteristics. For example an application might specify a range of temporal values relating to the frame display rate and a range of spatial values to describe image resolution.

Our QoS Levels are somewhat more general. The levels for each application are characterized by the application resource usage and benefit provided by each level, and the period for that level. They do not explicitly specify or otherwise limit the ways in which applications may modify their resource usage from level to level, allowing temporal, spatial, and any other kind of change in processing between levels. This distinction allows each application to employ a unique, application-specific, soft real-time policy that best reflects the needs of the application.

There is a fixed format for the QoS Level specification used by the DQM. Each application provides a benefit table specifying its QoS Level information in the form of a *maximum CPU requirement*, *maximum benefit*, and a set of quadruples of the form <*Level*, *Resource usage*, *Benefit*, *Period*>. Level 1 represents the highest level and provides the maximum benefit using the maximum amount of resources, and lower QoS Levels are represented with larger numbers.

```
Max Benefit: 6
Max CPU Usage: 0.75
Num Levels: 6
Level  CPU    Benefit   Period(µs)
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: QoS Levels with CPU Usage, Benefit and Period

A representative benefit table is shown in Figure 1. Note that although maximum benefit is specified by the user at runtime, it is included in this table for simplicity. The rest of the information is statically specified by the application developer at the time that the application is created.

Figure 1 indicates that the maximum amount of CPU that this 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 allo-

cation, but that if the level of allocation is reduced to 40%, the quality of the result will be substantially less (25%). In this example, the period of the application is held constant over the set of levels, but in general the period can remain the same, increase, or even decrease.

3.1. User Responsibility

QoS Levels provide a means for the user to play a role in the system policy. As with all general-purpose operating systems, the user is free to initiate any application at any time. This is a characteristic that distinguishes soft real-time desktop systems from most hard real-time systems. Because there is no admission phase, the system has no *a priori* information about the applications that will be executed. Consequently, it is possible that the applications themselves will be best-effort applications that do not comply with the QoS Level model and the system (as implemented) can make no guarantees about the application performance. Of course, if the user only initiates compliant applications, then the system can provide very good soft real-time performance. In the presence of non-compliant applications, the DQM will adjust the levels of the compliant soft real-time applications as necessary to keep them running without missing deadlines within the available resources (see Section 6). Once the non-compliant applications have left the system, the compliant applications will return to stable performance at the appropriate levels.

In starting the applications, the user can specify the maximum benefit for the application based on his/her preferences i.e., application A is more important than application B. This is analogous to priority or importance in other systems and, together with the relative benefit information in the QoS Level specification, is used by the DQM in deciding how to allocate system resources. The user also specifies any application parameters at run-time, and interacts with the application as it executes. Finally, the user is the recipient of the application output and for whom the benefit is calculated.

3.2. Applications

Applications are written using the QoS Level model; each application is developed with a set of levels at which it can operate, each level having a particular set of resource requirements, a corresponding benefit, and an associated period. Each application is accompanied by a specification of the levels and their resource needs, benefit, and period. The QoS Levels chosen for a particular application are highly application-specific and reflect the particular soft real-time policy that the developer wishes to employ. A given application can soften one constraint multiple times to create a set of levels, or soften several simultaneously in different ways for the different levels. The only restriction is that the levels be ordered according to the resource used and the benefit provided. The levels can be implemented as parameters on the algorithms that implement the functionality of the application, or can consist of calls to completely different algorithms. These choices are up to the application developer and will reflect the details of the desired soft real-time policy. In the simplest case, where period changes and everything else is held constant, the application performs the same computation at each level, but a DQM component manages the dead-

line timing. In a slightly more general case (e.g., with a change in the resolution of a transmitted image), the change will be reflected in one or more parameter values used by the algorithm. In the most general case where completely different algorithms are employed for the different levels, a global switch statement at the top of the application will select the appropriate function to call based on the current level.

The determination of appropriate CPU usage numbers for the various levels of an application is relatively straightforward inasmuch as it can be measured directly by running the application with only a small amount of instrumentation added. By running the application for some large number of periods at each level, and measuring the time required, an average resource usage number for each level can be determined. By measuring over a large number of periods, the measurement can be made reasonably accurate. As is elaborated in Section 6, there are several situations that can arise in general purpose environments in which these measurements can still be inaccurate. Interprocess interference (e.g. competition for unaccounted for resources) can cause an application to take longer than expected. Variations in the required resources as, for example, from data dependencies in the algorithm, can also result in inaccuracies in the measurements. Additionally, running the application on a different architecture from the one on which it was measured will result in inaccuracies in these measurements. Finally, errors in the measurement process itself can exist. Nevertheless, we have found that it is possible to make reasonably accurate measurements of the resource usage of the applications. In Section 6 we discuss robust strategies for managing inaccurate resource usage estimates from these and other sources.

Determination of the relative benefit numbers for each level of each application is not as straightforward. The relative benefit numbers for each application should reflect the reduced output quality from running at a given level relative to the quality from running at the highest level. Briefly, the determination of relative quality can be made in several ways. For common soft real-time applications such as desktop audio and video, there is a large volume of research data from the telecommunications and television industries relating the perceived quality of outputs relative to the data bandwidth, rate, size, color, etc. see, for example, [32]. Alternatively, the users themselves could specify their perceived relative benefits interactively based on their current satisfaction. It is reasonable to expect that the determination of benefit is an area where additional research effort will be spent once the mechanisms used to implement soft real-time have become sufficiently standardized.

4. DQM System

The entire system consists of a middleware application referred to as the *DQM* and a library of DQM interface and soft real-time support functions called the *Soft Real-Time Resource Library* (*SRL*) [15]. The DQM application is a fully functional soft real-time system that has allowed us to experiment with different algorithms for dynamically adjusting levels among a set of running applications with varying numbers of levels with varying resource requirements and to explore problems that arise in developing a middleware QoS manager. Like most existing soft real-time systems, the DQM manages only the CPU resource.

The current implementation of the system has been implemented on top of Linux and Solaris. Consequently, the specific details of the implementation are dependent upon the facilities and capabilities of those Unix platforms. However, in general the facilities on which it depends are provided by any general-purpose operating system.

Figure 2 shows the components of the system and the information that is passed among them (the user and applications were described in Section 3). The user is responsible for starting the applications (e.g. via the command line), specifying the applications' absolute benefit, i.e. the relative importance of each application; the user is the ultimate recipient of the systems' output. The applications perform their work according to the algorithm(s) their designers chose. The SRL (linked with each application) provides facilities to allow the application to communicate with the DQM. The SRL sends the application QoS Level information to the DQM, monitors the application performance, and reports application state and missed deadlines to the DQM. The SRL also reports the current level, as selected by the DQM, to the applications. The DQM monitors system and application state via the operating system and the information from the SRLs, and chooses appropriate levels for the applications. All components are independent of the operating system and execute as user-level processes. The following sections discuss the SRL and DQM in more detail.

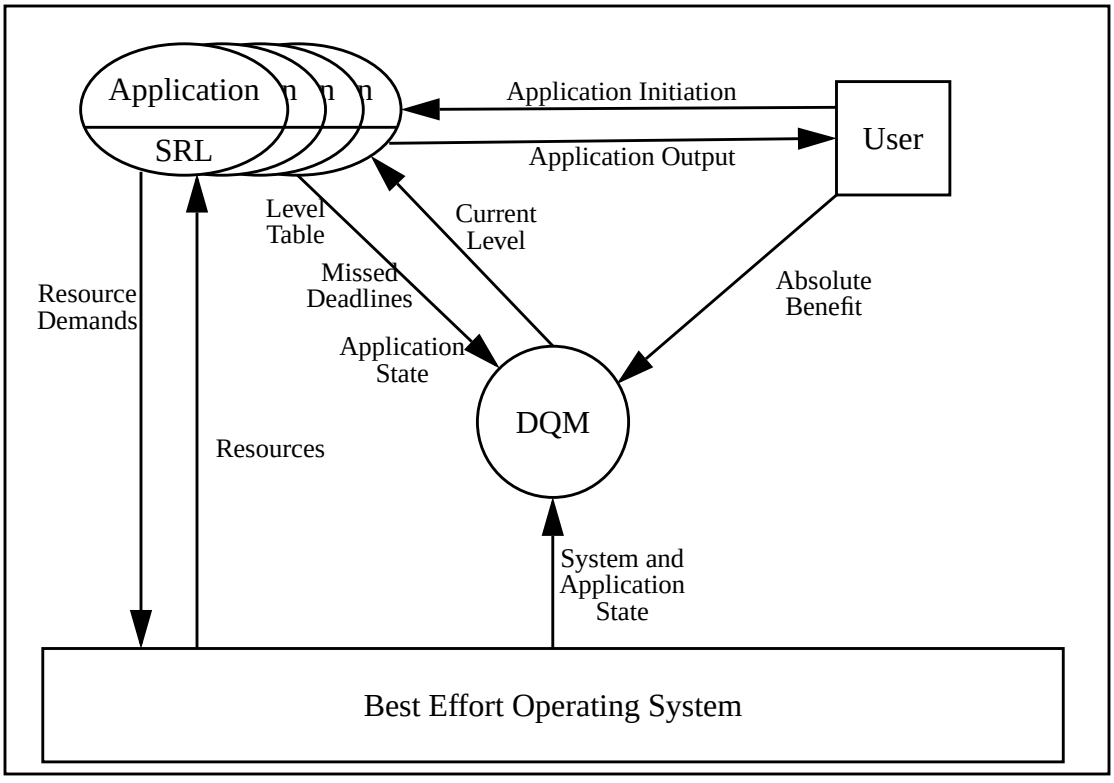


Figure 2: The DQM System

4.1. The SRL

The SRL is a library of functions linked to each application that provides three simple functions to be called by applications: `dqm_init()` to initialize the computation, `dqm_loop()` called at the beginning of each period, and `dqm_exit()` called when the application terminates. The three functions perform many operations in the system: They notify the DQM of application entrance and termination, initialize important system data structures with application benefit table information, manage and monitor application timing and periods, measure application resource usage (when necessary), detect missed deadlines and notify the DQM of the misses, report the DQM determined level to the applications or modify application level directly (in the case of the Distributed algorithm (discussed in Section 5), and record application QoS statistics.

The `dqm_init()` function reads any relevant parameters, attaches to shared memory used to communicate with the DQM, attaches to the DQM's deadline miss semaphore (used to signal that a deadline has been missed by an application), creates its own deadline miss semaphore (used in the local determination of missed deadlines), then notifies the DQM (through a shared memory location) that the application has begun running.

The `dqm_loop()` function does the bulk of the SRL work. The SRL manages application timing using alarms, signals, and semaphores. For each application, a deadline handler increments the local deadline miss semaphore. A timer alarm is set to go off exactly once per period. When the alarm goes off, the deadline handler function is called, incrementing the deadline miss semaphore. Each time that dqm_loop() is called by the application, the deadline miss semaphore is checked. If the semaphore has value 0 or 1, then we completed processing before the expiration of the deadline (or just after the expiration of the deadline). In that case, the function waits until the semaphore has value 1 and then returns the current level to the application, allowing it to proceed. If the semaphore has value greater than 1 then we know that the application took more than its allotted time for that period and a deadline has been missed. In that case, the DQM missed deadline semaphore is incremented and the application is blocked until the next time that the alarm goes off, and then proceeds as usual.

The SRL missed deadline detection has a feature that increases the stability of the applications and the system: average deadline miss detection. The average deadline miss detection is part of a broader goal of allowing an application to specify when the performance is outside of its acceptable range and the DQM should be notified. Currently only a single policy is implemented, but it is envisioned that future systems will allow applications to specify or even implement their own policies. The current policy works by only signalling a missed deadline when an application has become 1 period out of phase, rather than every time an application takes slightly longer than 1 period for 1 iteration. By signalling a missed deadline only when the value of the local deadline miss semaphore is 2 or more, we allow some deadlines to be missed (as is permissible in soft real-time applications), but signal the DQM when any period is missed by too much (currently more than 1 period), or the cumulative deadline miss amount adds up to more than 1 period. Thus if an application has varying resource usage and alternates between

slightly longer than 1 period and slightly less than 1 period, the average deadline miss amount will be zero and no deadline miss will be signalled.

A future version of the system will incorporate a more general miss notification mechanism. This will allow the application developer to specify an application-specific policy with respect to deadline miss detection and notification. For example, various applications may require more stringent deadline miss detection, i.e. never miss a deadline. Alternatively some applications may allow for a more relaxed deadline miss profile, i.e. make at least every other deadline, or don't miss more than 1 out of every 3 deadlines. Such a mechanism will further support the notion of user-specified soft real-time policy.

As mentioned above, `dqm_loop()` returns the current level of the application each time that it is called. It also checks for level changes and each time a change in period accompanies a change in level, the period for the aforementioned timer alarm is changed accordingly. It should be noted that the applications and the DQM operate asynchronously. The applications only change level at the top of the main loop, upon receiving a different level from the `dqm_loop()` function. Each time that an application algorithm is called to run at a particular level, it is guaranteed to finish before the level change is made effective.

There is one situation where the SRL itself will change the level of an application; when the Distributed algorithm is running. This decentralized algorithm allows each application to change its own level upon missing a deadline, and to raise its own level periodically to test if there are available resources for it to use.

Finally, the SRL can measure the resource usage of the application each time that `dqm_loop()` is called and record the information in the output data structure (described in [4]). The resource usage of the applications can either be measured once per period by the SRL, or once per sampling interval by the DQM. The resource usage is determined by reading the appropriate field in the `/proc` pseudo-file for the process executing the application. Having the SRL measure the application resource usage once per period results in better individual measurements that exactly reflect the per-period resource usage of the application at that level. Unfortunately, the measurements from the individual SRLs cannot be summed to get a systemwide resource usage measurement because the measurements are taken at different times.

4.2. The DQM

The DQM is the centralized resource management mechanism for the system and executes as a separate user-level process. It dynamically determines a level for each of the running applications based on the available resources and the user-specified benefit of the applications, and 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. In accomplishing these goals, the DQM does several things: it initializes the shared memory data structures, it monitors application resource usage and system resource availability, it monitors appli-

cation deadline misses, it adjusts application deadlines as necessary, and it records relevant information in the output data structure and writes it out at system termination.

The DQM must be started before any applications are initiated in order to initialize the shared data structures. After initializing the data structures, the DQM starts two child processes that do the bulk of the work of the DQM; the QoS Manager and the Sampler. The QoS Manager is responsible for monitoring deadline misses and, when necessary, lowering the overall resource usage of the system by lowering the level of one or more applications. In so doing, it may also raise the level of one or more (but not all) applications.The Sampler is responsible for monitoring the unused resources in the system and, when possible, raising the level of one or more applications. In so doing it may also lower the level of one or more (but not all) applications. These two operations are largely separate and while the two processes are executing concurrently, the two mechanisms will not be acting simultaneously. Either the system is in a situation where the applications are trying to use more resources than are available and the QoS Manager will be operating to reduce the allocated resources, or the system is running stably without missed deadlines. If the system is running stably without missed deadlines, it may be the case that there are excess unallocated resources and the Sampler may attempt to increase the allocated resources. Consequently (except in the most degenerate situation where the resource usage of the applications is grossly underestimated and the Sampler is using the estimates themselves to determine the presence of unallocated resources), the QoS Manager and the Sampler will never be acting simultaneously.

The QoS Manager monitors deadline misses by waiting on the DQM deadline miss semaphore. When the semaphore is non-zero, it calls one of the resource allocation algorithms to assign a new set of levels with lower total resource usage than the current allocation. Because a bad resource allocation that causes the applications to attempt to use more than the available resources may result in several deadline misses from one or more applications before the QoS Manager has time to respond by changing the levels, the QoS Manager employs a *skip value* (set by a runtime parameter) to determine the number of deadline misses to receive before changing the levels of the applications. This skip value builds some hysteresis into the system, allowing the QoS Manager time to respond to problems and allowing the system to stabilize after the levels of the running applications have been changed before the QoS Manager again responds to missed deadline signals. In the experiments we have conducted, a skip value of 3 has been found to work well (see Section 5). In general, the skip value could easily be replaced by a more general function specifying the acceptable number of missed deadlines. For instance, levels could be changed any time the percentage of missed deadlines exceeds a specified number.

The Sampler periodically monitors system resource availability and determines CPU underutilization by examining system idle time. The periodicity is managed with timer alarms and semaphores in a manner analogous to the SRL deadline management. A timer alarm is set and a handler is specified. The handler increments a semaphore each time that the alarm goes off. The Sampler waits on the semaphore and executes once each time that it is set.

System idle time can be determined in several ways including via the OS, through the `/proc` file system, by measuring the CPU usage of a low priority application, and by taking the complement of the sum of the CPU usage measurements (or estimates) of the running applications. The particular method used is dependent on the information available in the host OS. If the OS provides idle time information, this information is the most reliable. As mentioned above, taking the complement of the sum of the individual resource usage estimates for the applications can lead to some problems. Additionally, measuring the resource usage of a low priority application can also lead to some problems. In an experiment with Solaris 2.7, a lowest priority compute-intensive process running concurrently with a high priority compute-intensive process was observed to consume about 2 percent of the CPU until some moderate amount of time after it was started, at which time its resource usage jumped to about 24 percent of the CPU. While the cause of this phenomenon is unknown, it demonstrates the unreliability of using such a method to determine system idle time. Of our two research platforms, Linux and Solaris, system idle time is directly available from Linux, and not directly available from Solaris.

To complement the hysteresis provided by the skip value used by the QoS Manager, the Sampler uses an idle time threshold to determine when adequate idle time is available to increase the amount of allocated resources. This threshold prevents the Sampler from trying to reallocate the resources in response to minor changes in resource availability. The threshold is dynamic in the sense that if no allocation is found higher than the current allocation that can fit within the available resources when the threshold value is met, the threshold is raised so that a greater amount of resources must be available before the Sampler will again try to change levels. This avoids repeatedly attempting to change levels for a given resource availability when it has been determined that no suitable higher allocation exists.

When the DQM is set (via parameters) to record application statistics, the Sampler does so during its periodic execution. This information is written to the output data structure and dumped to a file when the DQM terminates.

A major issue in the design of the DQM is how the QoS Manager and the Sampler actually select the level at which each application should execute: Approximation algorithms are used.; we have implemented and examined a variety of different algorithms and studied this issue extensively (Section 5).

Early in the development of the DQM, the first major issue -- feasibility --was addressed by considering two questions with respect to the DQM:

- Can soft real-time applications execute without missing deadlines when running on a best-effort operating system?
- Can applications utilize the available CPU resources effectively.

A set of simple DQM experiments was run on a 200 Mhz Pentium Pro PC running the Solaris operating system [6] (the experiments were repeated with the Linux operating system, with virtually identical results [4]). While Solaris provides some real-time scheduling classes using preemptive fixed priority scheduling, all applications and

middleware were executed using the standard UNIX scheduling classes. In [15] -- and all the results in the remainder of the paper -- the approach is shown to be both feasible and useful. System tests on this preliminary version of the DQM demonstrated that with a load that would require about twice the CPU capacity of the system, the DQM adjusted the QoS Levels so that applications began operating at reduced levels where the total load stabilized at 82% of the CPU time. As shown in Section 6, given suitable levels in the applications, the utilization can be increased to nearly 100%.

5. Resource Allocation Algorithms

The second issue in the design of the DQM was to explore policies for selecting the QoS Level for each application. Four representative allocation algorithms were initially implemented in the DQM: Distributed, Even, Proportional, and Optimal. 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 other algorithms. When an application misses a deadline, it autonomously selects its 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. It could be used in conjunction with the RT Mach Reserves mechanism [25] inasmuch as it does not assume any centralized decision-making or level management but allows each application to adjust to the resources that it has available[1]. Similarly, this algorithm could be used by applications in the MMOSS [11] and SMART [28] systems to dynamically adjust to the resources that they have been granted.
- The *Even* algorithm is the simplest centralized algorithm that we have implemented. In the event of a deadline miss, the Even algorithm reduces the level of the application that is currently using the highest fraction of CPU time. 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* algorithm uses the benefit parameter and raises or lowers the level of the application with the highest or lowest benefit/CPU ratio. This algorithm approximates the scheduling used in the SMART system [28].

1. To use this algorithm in an RT Mach system, the admission criteria would have to be changed to include a negotiation whereby an application's QoS Level is determined before it enters the system.

- The *Optimal* algorithm is loosely based on Jensen *et al.*'s benefit-based scheduling [19]. 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 and application-specified maximum CPU usage, as well as the relative CPU usage and relative benefit information specified for each level to determine a QoS allocation of CPU resources that maximizes total user benefit. Total user benefit is the sum of the individual benefits of the running applications, as determined by the relative benefit of the level at which they are currently executing and the maximum benefit specified by the user. The current implementation of Optimal uses an exponential-time search of the solution space to determine the optimal allocation. The Optimal algorithm performs extremely well for initial QoS allocations, but it reacts very poorly to changing resource availability. In particular, the allocations can change dramatically with a small change in resource availability, resulting in wildly-fluctuating QoS Levels for all applications in the system. To reduce this effect, an 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.

For all of the algorithms it is possible to specify a skip value (see Section 4). 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. Experiments using a skip value of 2 or 3 provide a damping effect that gives the DQM time to react to CPU overload situations, i.e. situations in which the total requested CPU allocation exceeds the available CPU cycles.

For a given set of applications, the data presented in this section were generated by running the applications and the DQM (with level lowering only) and recording 90 samples of the current level, requested CPU allocation 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 9 seconds or 90 periods. By varying the runtime over several experiments we determined that 90 periods is adequate for observing the performance of the algorithms in a steady state situation.

To compare algorithms, a single representative set of synthetic applications was used. This simplifies comparisons of the results of the different algorithms, though it does not reflect on the generality of the system (we also experimented with the system using actual applications -- see Section 6). The application set (see Figure 3) 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 actual applications, the ranges of CPU usage and benefit values used test the QoS Level model and vary at least as much as one would find in most actual applications. For this set of experiments, application period was fixed at 1/10 of a second for all QoS Levels of all applications.

Figure 4(a) shows the QoS Levels at which the four application run using the Distributed algorithm with a skip value of 0, with the DQM executing on top of Solaris. The QoS Levels 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 3 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

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

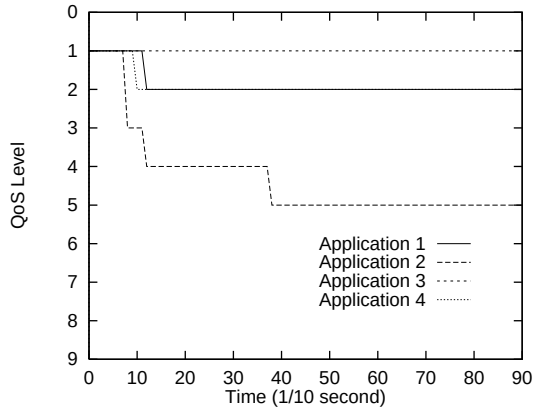
Application 3
Max Benefit: 5
Max CPU Usage: 0.22
Num Levels: 8

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

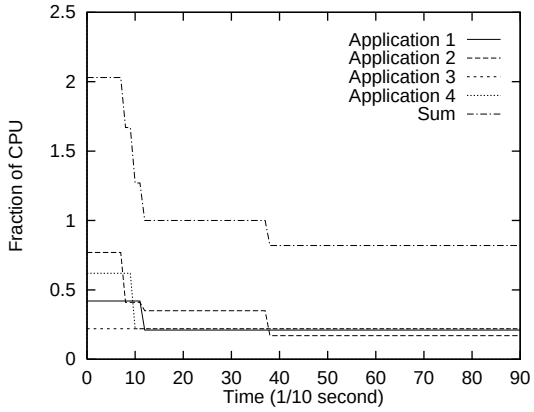
Application 4
Max Benefit: 2
Max CPU Usage: 0.62
Num Levels: 4

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 QoS Levels



(a) QoS Levels



(b) Requested CPU Allocation

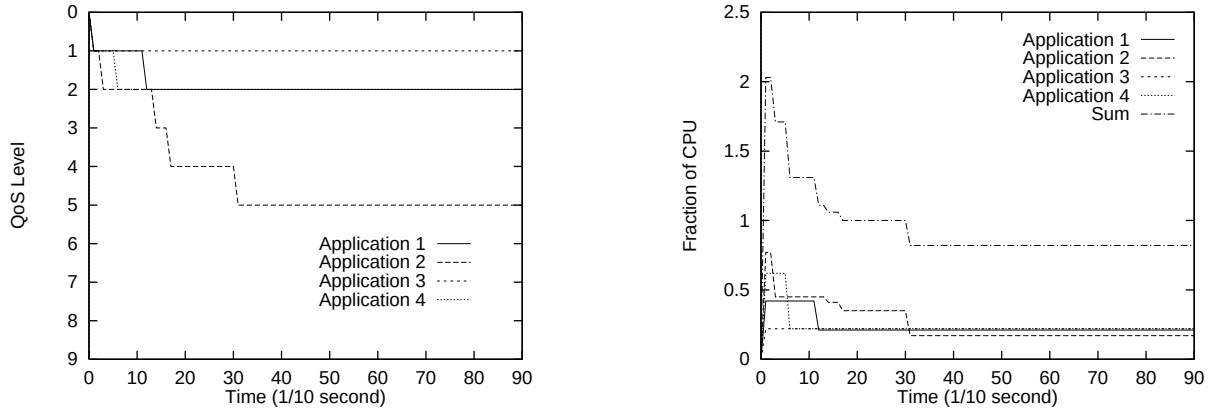
Figure 4: Distributed Algorithm (skip=0)

to each missed deadline in lowering its level, regardless of the transient nature of the overload situation. The lack of changes at the very beginning of each graph are due to the fact that we begin sampling just before the applications have started executing in order to record everything that occurs.

Figure 4(b) shows the requested CPU allocation for the applications in the same experiment. Here we see that the total requested CPU allocation (designated Sum) starts out at approximately twice the available CPU, and then drops down to about 100% as the applications are adjusted to stable levels. Note also the adjustment at sample 38 (described above for Figure 4(a)), lowering the total requested CPU allocation to approximately 80%. Because these experiments are conducted with level-lowering only, the system does not adjust the application levels upward to take advantage of the excess available resources.

Figure 5(a) shows the QoS Levels for the same application set, resource allocation algorithm, and skip value running on the Linux operating system. Figure 5(b) shows the requested CPU allocation for this experiment. While the graphs show almost exactly the same performance, there is an important difference. The synthetic applications performed differently on the Linux operating system, consuming almost twice as many CPU cycles. This highlights the need to modify the maximum CPU usage numbers for the applications for each platform on which they will be executed, to develop a platform independent specification of application resource need, or to develop a mechanism to dynamically determine these numbers at runtime (see Section 6 for a discussion of dynamic estimate refinement, a mechanism that we developed to accomplish this).

To compare the results with the Solaris experiments, the synthetic applications were modified to consume the same percentage of the CPU, i.e. so that an estimated application load of 100% on Linux equals an estimated application load 100% on Solaris. As can be seen from the graphs, the results are almost identical to those obtained for the Solaris system. This similarity was observed for all of the experiments and so the rest of this section will deal only with the Solaris results. Complete graphs of the Linux experiments are given in [4].



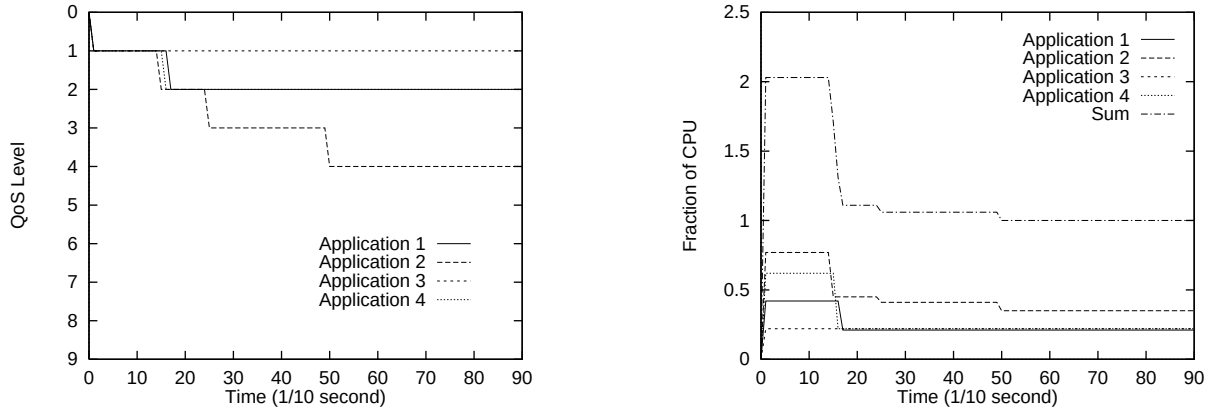
(a) QoS Levels (b) Requested CPU Allocation

Figure 5: Distributed Algorithm (skip=0) on Linux

Figure 6(a) show the QoS Levels and Figure 6(b) shows the requested CPU allocation for the Distributed algorithm 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 3 deadline miss, rather than for each one. A larger skip value can result in a longer initial time before stability is reached, but will result in less overshoot, i.e. adjustment to levels lower than are supportable within the available resources, because it gives the applications time to stabilize after level adjustments.

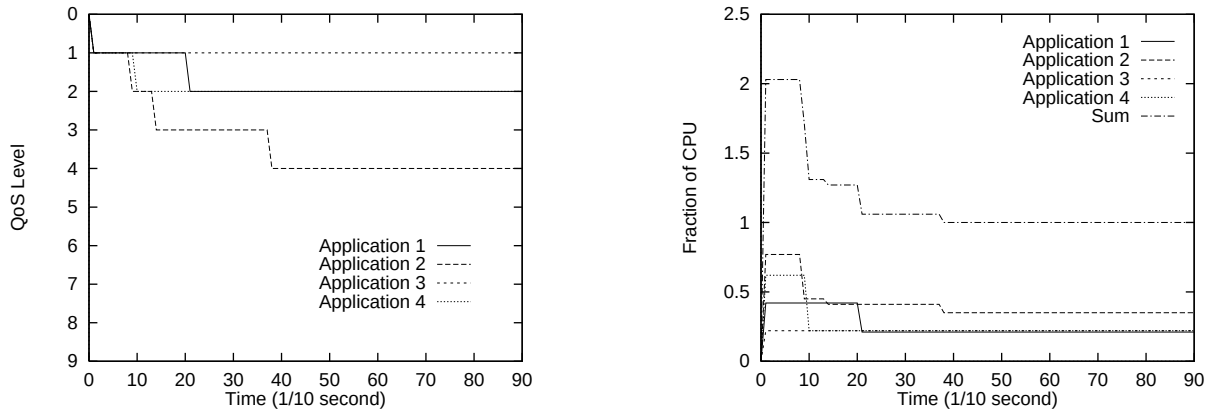
Overshoot can result when one or more applications miss multiple deadlines as a result of a situation of overload before the system has a chance to change the levels to fix the problem, and the system reacts to each missed deadline as if it was the result of a separate overload situation and thereby lowers the levels more than necessary to correct the problem. Now, stability is not reached until about sample 16, and there are two small adjustments at samples 24 and 49. However, the overall requested CPU allocation stays very close to 100% for the duration of the experiment with essentially no overshoot as is observed in Figure 4 during the level adjustment at sample 38. Because the system does not overshoot, utilization is higher and application 2 is able to remain at level 4 rather than unnecessarily dropping down to level 5.

Figure 7(a) show the QoS Levels and Figure 7(b) shows the requested CPU allocation for the Even algorithm with a skip value of 2. This centralized 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. This is due to the fact that, given applications with equal period as in this example, the application with the highest requested CPU allocation, i.e. the one that would be selected by the Even algorithm to have its usage lowered if a deadline were missed, is also the one most likely to miss a deadline and therefore lower its own usage in the Distributed algorithm.



(a) QoS Levels (b) Requested CPU allocation

Figure 6: Distributed Algorithm (skip=2)

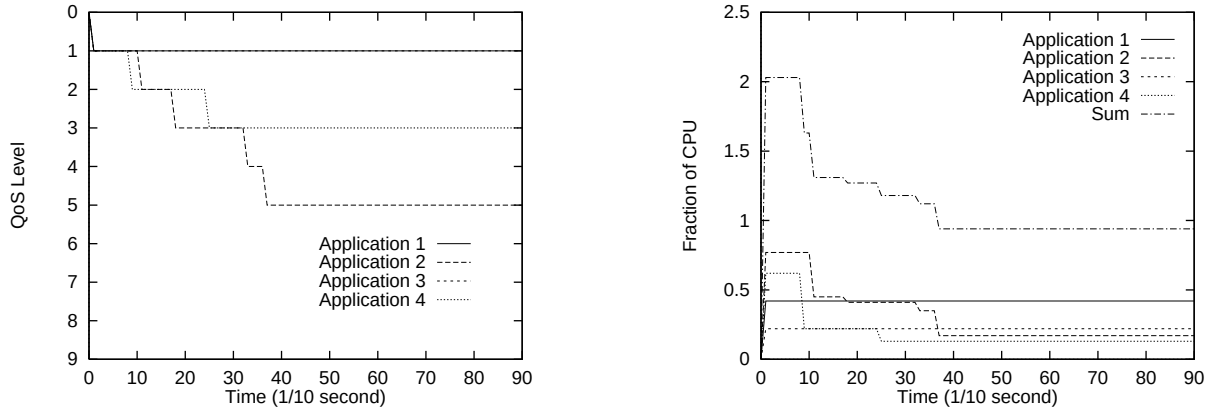


(a) QoS Levels (b) Requested CPU Allocation

Figure 7: Even Algorithm (skip=2)

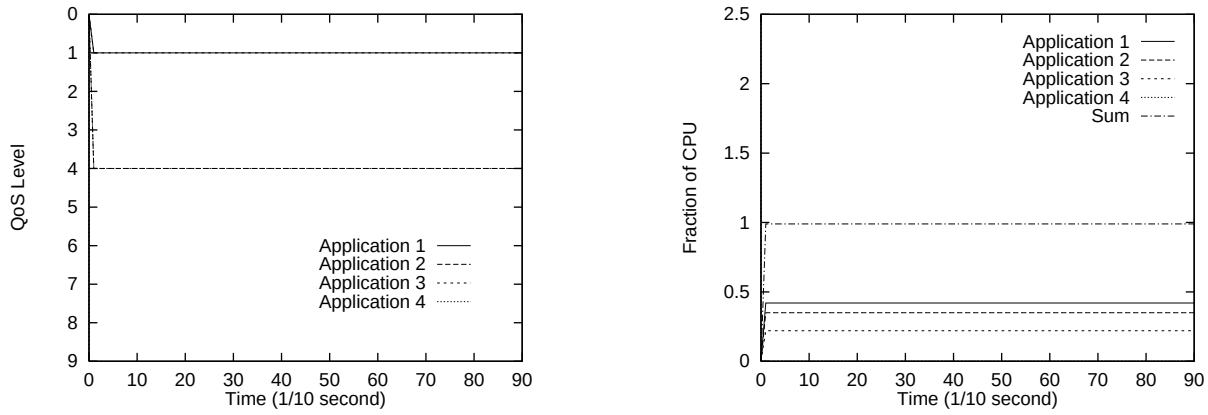
Figure 8(a) shows the QoS Levels and Figure 8(b) shows the requested CPU allocation for the Proportional algorithm with a skip value of 2. This algorithm attempts to distribute shares of the available CPU cycles to each application proportional to that application's benefit. Using the previous algorithms the CPU percentage used by all applications was approximately the same. With this algorithm, the requested CPU allocation/benefit ratio is approximately the same for all applications. In fact, the ratio is as close to equal as can be reached given the QoS Levels defined for each applications. The resulting QoS Levels are therefore different from those in the previous algorithms although the overall requested CPU allocation remains at about 100%.

Figure 9(a) shows the QoS Levels and Figure 9(b) shows the requested CPU allocation for the applications running with the Optimal algorithm with a skip value of 2. 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 a total bene-



(a) QoS Levels (b) Requested CPU Allocation

Figure 8: Proportional Algorithm (skip=2)

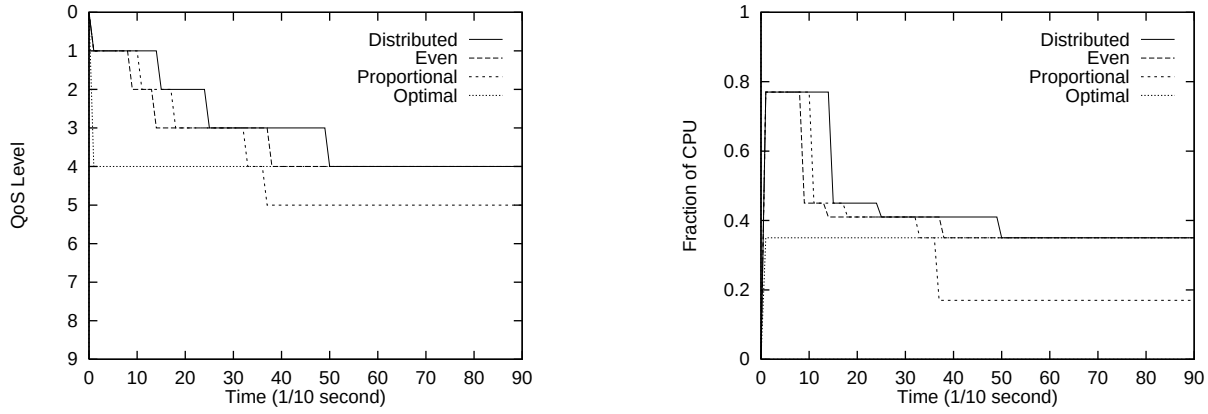


(a) QoS Levels (b) Requested CPU Allocation

Figure 9: Optimal Algorithm (skip=2)

fit 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, it can result in a lower total CPU utilization and can thereby result in a more stable steady state, yielding no additional deadline misses and requiring no corrections. However, this algorithm is the least stable given changing CPU resources (such as those caused by other applications entering or leaving the system).

Figure 10(a) shows the QoS Levels and Figure 10(b) shows the requested CPU allocation for application 2 under the four algorithms graphed separately above. These graphs summarize 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 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



(a) QoS Levels (b) Requested CPU Allocation

Figure 10: Application 2 with All Four Algorithms (skip=2)

longer. The Proportional algorithm reaches steady state at about the same time as the Distributed and Even algorithms and its allocation to application 2 is less in this example.

Figure 11 shows the sum of requested CPU allocation for the same four algorithms shown in Figure 10. 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. The Optimal algorithm reaches steady state immediately while the others take between 3.8 and 5 seconds to stabilize.

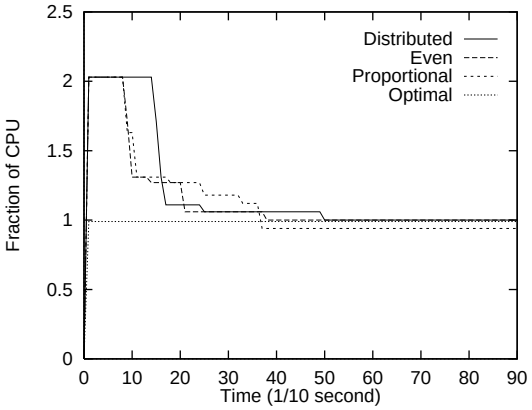


Figure 11: Summed Requested CPU Allocation with All Four Algorithms (skip=2)

6. Issues with Middleware Implementation

The algorithm comparison results were obtained with algorithms that lowered the level of applications in order to lower overall CPU usage, but did not raise the level of any application to take advantage of excess available resources. Because soft real-time applications can be expected to execute in a dynamic environment where application are entering and leaving the system, thereby changing the available resources up and down, the system must be able to raise the levels of various applications at various times to take advantage of the available resources. Furthermore, even when overall resource usage must be lowered, an optimal allocation might require that some application levels be raises as others are lowered.

The ability to raise the levels of applications introduces several new issues [5]. The DQM uses a static threshold level of measured system idle time to trigger the level-raising algorithm. In order to make accurate level-raising decisions, the resource usage of the applications and the amount of unused resources (idle time in the case of CPU usage) must be accurately known. OS-based QoS systems have direct access to this information and make good use of it. Middleware solutions must make use of the information provided to user-space applications. Both Linux and Solaris 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 level-raising by the various resource allocation algorithms. However, in our experiments with level-raising, significant instability was observed in the application levels. Several specific issues were identified and addressed. The details of these issues and solutions are provided in this section. All of the experiments described in this section 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. In order to simplify the discussion, all of the experiments presented in this section used a single representative resource allocation algorithm, Proportional.

6.1. Issue 1: Inaccurate Resource Usage Measurements

As mentioned above, the `/proc` filesystem can be used to determine the resource usage of the applications, which is used by the DQM to determine whether or not an application can raise its level to take advantage of some excess available resources. However, a problem was found with this method of determining application resource usage; the measurements of CPU usage and idle time vary significantly from measurement interval to measurement interval.

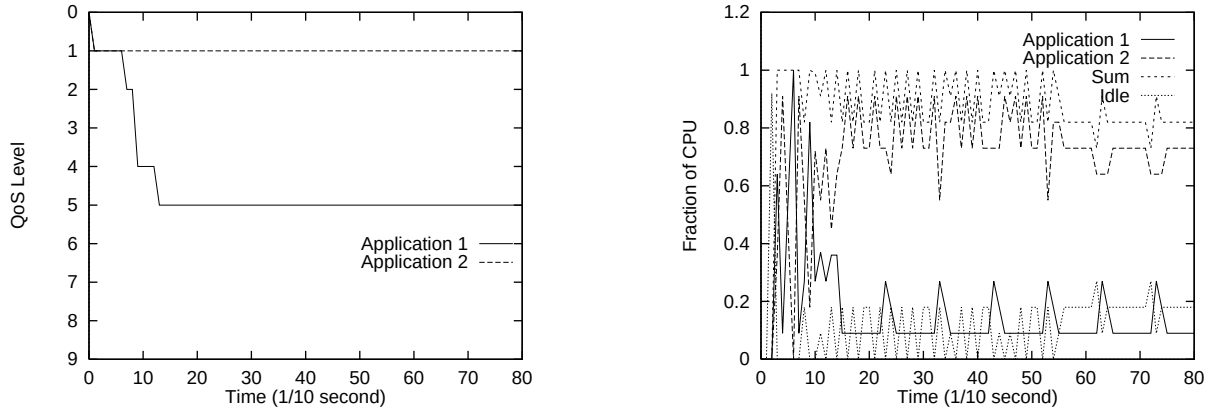
Figure 13(a) shows the QoS Levels for an experimental execution of the DQM with two synthetic applications run on top of the Linux operating system with level raising disabled. The benefit table information for these applications is given in Figure 12. These applications both have a period of 1/10 of 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.

```
Application 1
Max Benefit: 4
Max CPU Usage: 0.70
Num Levels: 8
Level   CPU     Benefit   Period(µs)
1       1.00    1.00      100000
2       0.74    0.92      100000
3       0.60    0.49      100000
4       0.55    0.34      100000
5       0.27    0.23      100000
6       0.12    0.11      100000
7       0.10    0.10      100000
8       0.05    0.05      100000

Application 2
Max Benefit: 5
Max CPU Usage: 0.70
Num Levels: 6
Level   CPU     Benefit   Period(µs)
1       1.00    1.00      100000
2       0.85    0.89      100000
3       0.62    0.75      100000
4       0.51    0.44      100000
5       0.33    0.27      100000
6       0.10    0.10      100000
```

Figure 12: Benefit Table for Two Synthetic Applications

As can be seen, the QoS 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



(a) QoS Levels (b) Measured CPU Usage

Figure 13: Problems with Inaccurate Resource Usage Measurements

application 2 at level 1 is 70%. This should yield a consistent idle time of 11%. The measurements are made by the Sampler running at intervals of 1/10 of a second and measuring the level and CPU usage of each application, and system idle time.

Figure 13(b) shows the measured CPU usage for these same 2 applications, the sum of the CPU usage, and 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 as much as 20% of the total CPU cycles and the sum of the CPU usage measurements fluctuates by as much as 35% of the available 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. This fluctuation makes the determination of when to raise levels almost impossible except in cases of very large amounts of excess available CPU cycles, approaching 50% of the total.

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 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 unavoidable 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 14 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 measurement granularity of 10%. Specifically, by reading the CPU usage every tenth of a second, i.e. every ten hundredths of a second, the measured usage of each application is an integral number of hundredths of a second between 0 and 10. If the actual usage is, for example, 75% of the available cycles, then when measured every tenth of a second this will alternately be reported as 7 and 8 hundredths of a second, as is seen for application 2 in Figure 14. Note that the idle time (measured by the Sampler) is measured slightly less frequently than the CPU usage so the graphed peaks in idle time don't match up exactly with the graphed dips in measured CPU usage.

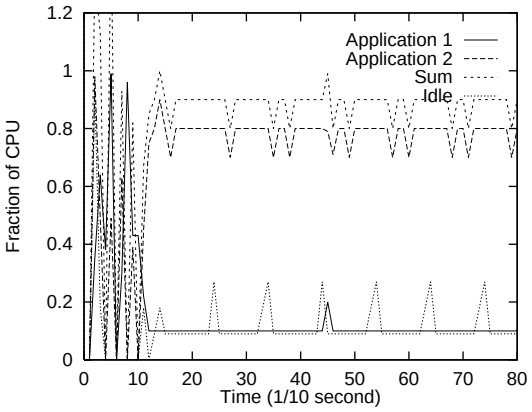


Figure 14: Measured CPU Usage with Synchronized Measurements

These remaining variations in measured idle time present still pose some problems for the middleware determination of when to raise application levels. In particular, the target utilization has to leave at least 10% overhead to

account for the measurement inaccuracies. However, this is significantly better than the 35-40% overhead that would be needed without the modifications.

6.2. Issue 2: Inaccurate Resource Usage Estimates

Another problem with raising levels occurs when the application resource usage estimates are inaccurate, in which case instability problems can occur. With underestimates the system may fail to stabilize at all. Specifically, the system may believe it can raise an application level but, because the application will use more resources than it indicated, such a change is not actually feasible. As a result, the system will make the change, one or more applications will miss deadlines, the system will lower the level of some application, and the cycle will repeat. With overestimates the system may stabilize, but at a lower utilization and benefit than is actually achievable.

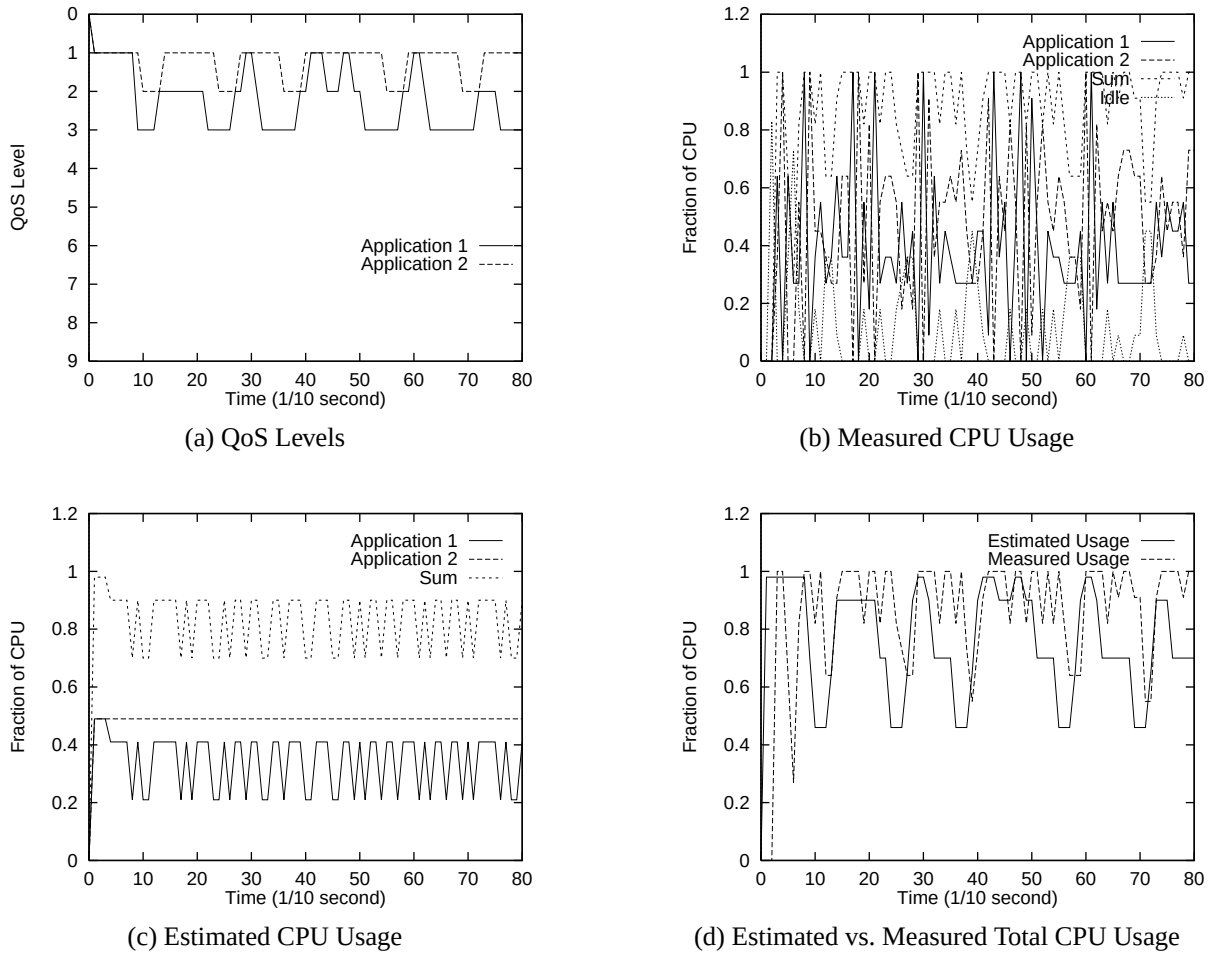
There are several reasons why the estimates might be inaccurate. One reason is that the measurements used to calculate the resource usage were simply faulty. A second possible cause of errors is that the measurements were made on a different machine or architecture than the one on which the application 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 + current measurement) / (weight + 1)

This calculation is executed 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 15(a) shows the QoS Levels for the same two synthetic applications described in Figure 12, 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 frequently as the DQM tries to adjust the levels to use the available resources efficiently. Figure 15(b) shows the measured CPU usage and Figure 15(c) shows the dynamically estimated CPU usage for the



(a) QoS Levels

(b) Measured CPU Usage

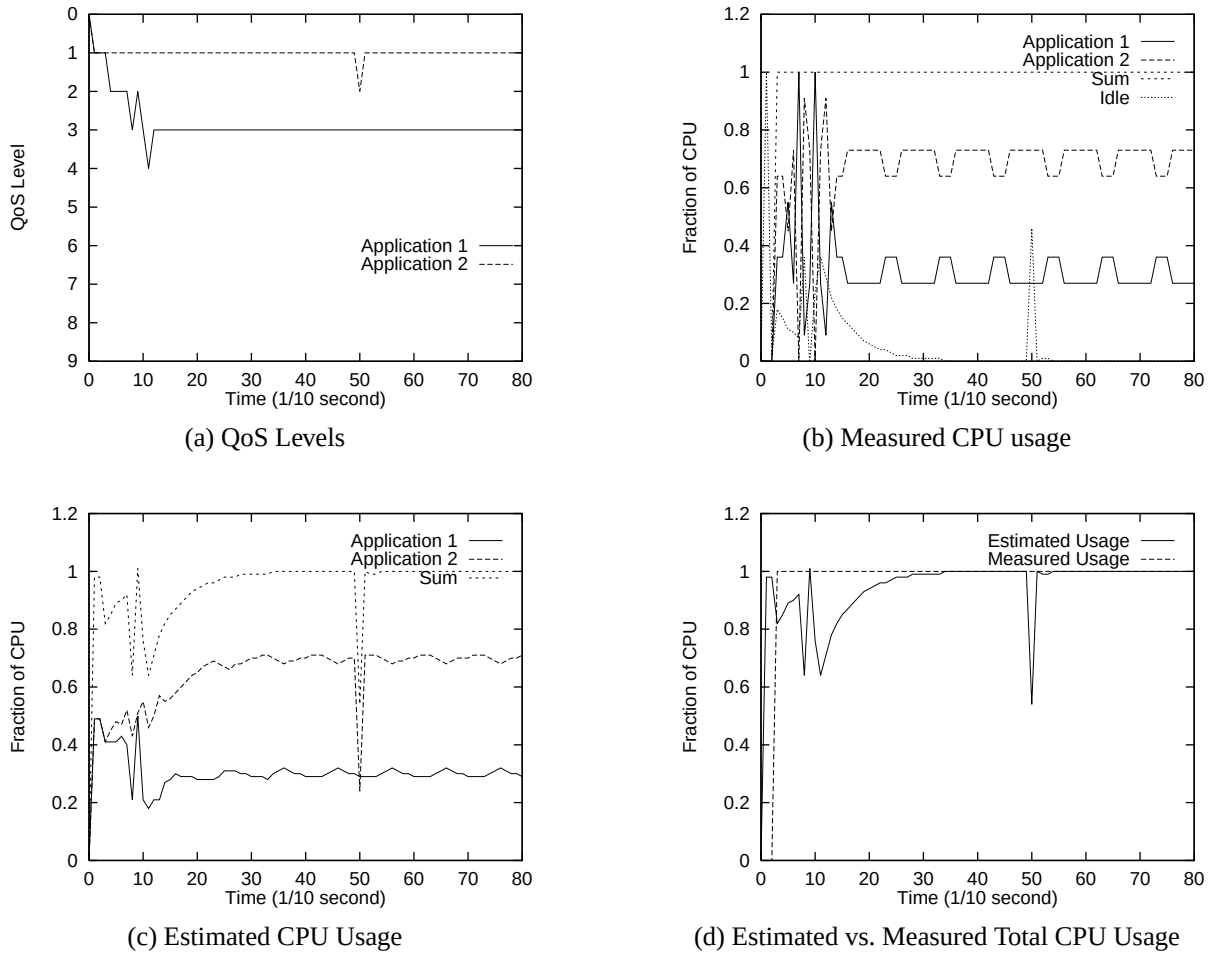
(c) Estimated CPU Usage

(d) Estimated vs. Measured Total CPU Usage

Figure 15: Problems with Inaccurate Resource Usage Estimates

same experiment. Finally, Figure 15(d) shows the estimated and measured total CPU usage for the same experiment. The estimated and measured CPU usage fail to match, resulting in the worst case situation described above.

Figure 16(a) shows the QoS Levels with dynamic estimate refinement. In this case, there is some level changing at the beginning, but the levels quickly stabilize as the estimates are corrected. There is a level change at sample 50, where the level of application 2 drops to 2, in response to an anomaly in the idle time measurement, but it is immediately corrected by the level-raising algorithm. Figure 16(b) shows the measured CPU usage for the same experiment. Note the anomalous idle time measurement at sample 50. Figure 16(c) shows the dynamically adjusted estimated CPU usage for the same experiment. Finally, Figure 16(d) shows a comparison of the estimated and measured total 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.



(a) QoS Levels

(b) Measured CPU usage

(c) Estimated CPU Usage

(d) Estimated vs. Measured Total CPU Usage

Figure 16: Inaccurate Resource Usage Estimates with Dynamic Estimate Refinement

6.3. Issue 3: Non-Compliant Applications

Concurrently executing real-time or non-real-time applications that fail to comply with the QoS Level model can result in instability in the execution of the soft real-time applications. In a general-purpose environment, the system will attempt to execute the soft real-time applications as well as possible within the available resources. To the extent that this is possible, the system will do the best it can, changing application levels as necessary (see Section 19 in Section 6.4 for an example).

In certain cases, this situation can be avoided. In dedicated systems, such as a web proxy, the applications to be executed can be limited to only those known to be compliant. In cases where this is not possible and stable performance is required, a resource usage enforcement mechanism such as that provided by RT Mach [25] and Rialto [21] would be useful.

6.4. Results with Representative QoS Level Soft Real-Time Applications

In order to confirm our results the DQM we developed two representative QoS Level 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.

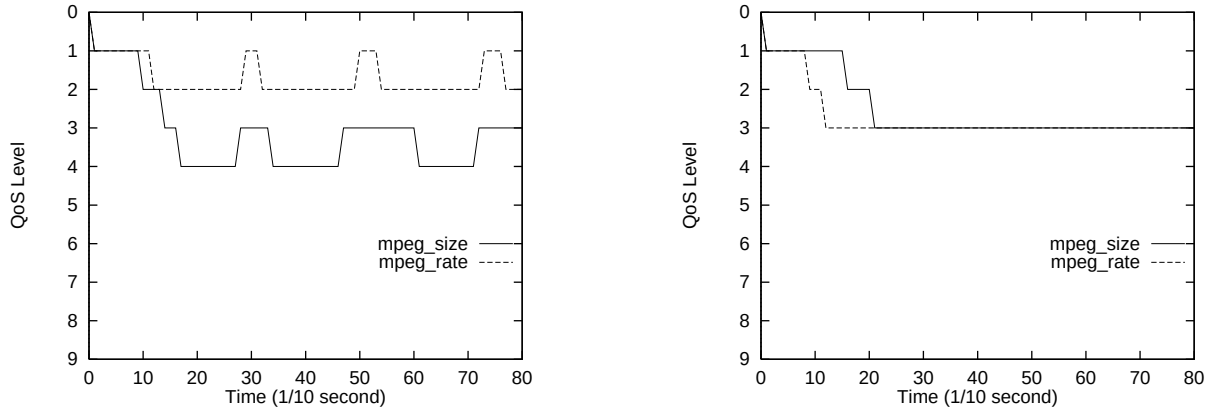
The second application, mpeg_rate, changes the frame rate of the displayed image from 0 frames/second to 10 frames/second in 2 frame/second increments. This particular application required no algorithmic changes other than the inclusion of the three SRL functions: `dqm_init()`, called once at the beginning of the application, `dqm_loop()`, called each time through the main loop, and `dqm_exit()`, called at application exit. Figure 17 shows the QoS Levels for these two applications.

```
Application 1 - mpeg_size
Max Benefit: 9
Max CPU Usage: 0.89
Num Levels: 8
Level  CPU    Benefit   Period(µs)
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(µs)
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 17: Benefit Table for mpeg_size and mpeg_rate

Figure 18(a) shows the QoS Levels that result for these two applications without dynamic estimate refinement and Figure 18(b) shows the QoS Levels that result for these two applications with dynamic estimate refinement. There are a number of level-changes that occur without



(a) Without Dynamic Estimate Refinement

(b) With Dynamic Estimate Refinement

Figure 18: QoS Levels for mpeg_size and mpeg_rate

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

Figure 19: shows the same two applications, this time over a duration of approximately 15 minutes. It shows the performance of the system over a long duration, and in the presence of non-compliant applications. Several level changes are seen after the initial period of instability. These are caused by the execution of non-compliant applications (e.g., system daemons) running concurrently. In each case, the system adjusts the levels of the applications appropriately to account for the resources used by the non-compliant applications, then adjusts them back when the resources are again available.

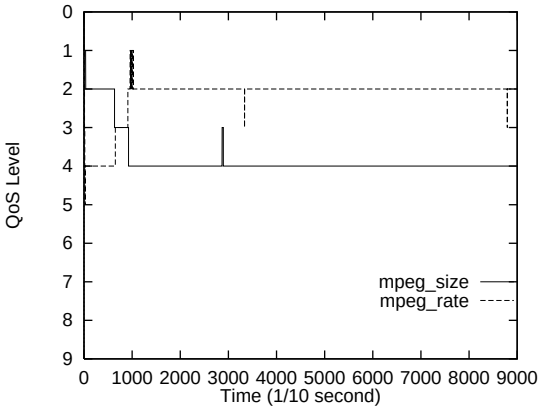
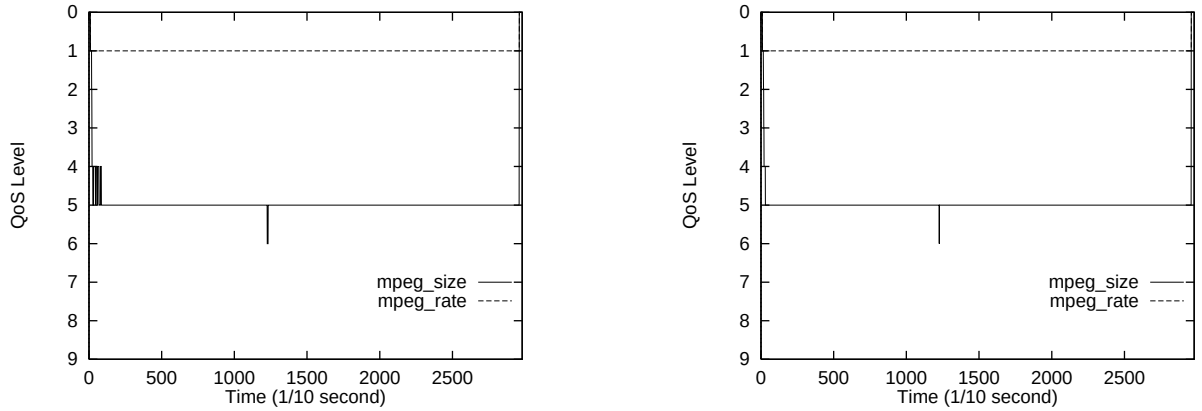


Figure 19: The Effect of Non-Compliant Applications

6.5. Dynamic Estimate Refinement Without Initial Estimates

Because dynamic estimate refinement was so successful in managing inaccuracies in the resource usage estimates of the applications, we did some additional experimentation to determine its applicability to situations where the applications lack initial estimates altogether. This is a highly desirable characteristic for any system that is to be used in a general processing environment.

Figure 20(a) shows an experiment with the two applications discussed above, with zero as the initial CPU usage estimate for all levels of both applications. The duration of the experiment is 3000 samples (5 minutes). Using dynamic estimate refinement, the levels take some time to stabilize, reflecting the time required for all of the estimates for the running levels to converge to the actual values. This is complicated somewhat by the fact that the system can only measure (and therefore adjust) the currently executing level of the application. Upon determining



(a) Using Zero as the Initial Estimate for All Levels

(b) Using Benefit as the Initial Estimate for All Levels

Figure 20: QoS Levels for Dynamic Estimate Refinement with Approximate Initial CPU Usage Estimates

that there are not enough resources for the current level, the DQM changes the level of the application, and then refines the estimate for the new level. This repeats until all applications have settled on levels at which they can execute without missing deadlines. In this example, it takes 85 samples (8.5 seconds, or approximately the entire duration of the previous examples) for the estimates to converge and the levels to stabilize.

Figure 20(b) shows the same experiment, but using relative benefit as the initial CPU usage estimates. While this estimation method is unlikely to be exactly correct, it is likely to result in estimates that are far closer to correct that simply assuming all levels use no CPU at all. Furthermore, since benefit and CPU usage are both monotonically decreasing, and the benefit to the user is always somewhat proportional to the CPU cycles allocated to the application, benefit is a reasonable, although very rough, estimate in the case where no initial CPU usage estimates are available. Consequently, in this example the estimates converge and the system stabilizes in 30 samples (3 seconds), significantly better than was achieved using zero as the initial estimate for all levels.

7. Summary and Conclusion

This work presented in this paper represents several significant contributions to the state-of-the-art in soft real-time processing:

- **Development of the QoS Level model of soft real-time**. Starting with the basic idea proposed by Tokuda and Kitayama [35], QoS Levels were evolved and significant issues examined and resolved
- **Separation of soft real-time policy from soft real-time mechanism**. In evolving QoS Levels, this work demonstrates the feasibility and utility of a policy-independent soft real-time mechanism.
- **Demonstration of the feasibility of cooperative middleware-only soft real-time**. This work demonstrates that middleware-only soft real-time systems built on top of best-effort operating systems are feasible and examines some of the important issues that arise in such a system.
- **Development of Dynamic Estimate Refinement**. This study introduces dynamic estimate refinement to deal with inaccurate resource usage estimates that can arise from inaccurate measurements, changing resource requirements, different run-time architectures, and interapplication competition for unaccounted for resources.
- **Development of a robust soft real-time system**. This research demonstrates the feasibility of running soft real-time applications without detailed *a priori* application knowledge. In particular, the system runs well without any *a priori* knowledge about the application resource usage.
- **Experimental validation of all concepts**. The ideas presented in this paper are validated in the context of a soft real-time system. Both actual and synthetic soft real-time applications were used to exercise the system and results were gathered by executing the system.

This work has extended and exploited the notion of QoS Level soft real-time, and in so doing has resulted in the development of a new system for soft real-time system developers. QoS Level soft real-time has been demon-

strated to be a feasible and natural method for supporting soft real-time processing, even in middleware-only implementations. This is especially important in dynamic soft real-time systems where the application mix is not known beforehand. Unlike hard real-time systems where the application mix and all of the resource needs of the applications are known at development time, desktop soft real-time systems must function in environments where nothing is known about the applications until runtime. QoS Levels provide one mechanism for allowing the system to learn about the applications at runtime and make reasonable resource allocation decisions based on that information.

QoS Levels also separate the soft real-time mechanism from the soft real-time policy. By requiring only that the application developer characterize the resource usage at each level and imposing no other constraints on the algorithm employed by the application at each level, QoS Levels place no limitations on the way in which the applications actually implement the levels and change their resource usage between levels. This is an important distinction from other implemented soft real-time systems, all of which either leave the issue unexamined, or impose a particular soft real-time policy.

This work has also demonstrated that QoS Level soft real-time is feasible in a middleware-only implementation. This new result is highly significant inasmuch as it was previously generally believed that best-effort scheduling is inadequate to support soft real-time processing [28]. However, this study has demonstrated that by having the applications cooperatively reduce their resource usage, it is possible to have multiple soft real-time applications executing concurrently on a best-effort operating systems without missing deadlines. Furthermore, this work has demonstrated that is possible to do so with very high utilization, approaching 100%.

Our work on different resource usage algorithms has shown the feasibility of a variety of different resource usage algorithms. It has demonstrated that centralized resource management algorithms perform somewhat better than distributed algorithms, by about 15% with respect to benefit, but that all algorithms result in high resource utilization.

Dynamic estimate refinement has been shown to be highly effective in dealing with inaccurate resource usage estimates, and even in situations lacking initial resource usage estimates altogether -- a previously unpublished result. By dynamically measuring the resource usage of the applications and correcting them with simple weighted averaging techniques, dynamic estimate refinement quickly corrects inaccurate resource usage estimates and dramatically improves application stability and overall system CPU utilization, and thereby also increases overall system benefit.

In addition to dynamic estimate refinement, this work presented a variety of techniques for managing issues that arise in the context of a non-soft real-time platform. Specifically, techniques were presented for dealing with inaccurate resource usage measurement issues, including lowering target utilization, synchronizing resource usage

measurement with application periods, and averaging the measurements over multiple periods to get higher accuracy.

A discussion of the limitations of the middleware approach was also presented. One of the primary goals of this research was to discover the OS requirements of such an approach by pushing the middleware implementation until hard limitations were discovered that could not be solved without adding additional services to the operating system. Resource usage measurement inaccuracies were somewhat difficult to surmount, and finer granularity of measurement would certainly have helped, but using the techniques discussed above, this limitation was overcome without having to change the OS. The one limitation that is insurmountable in a middleware-only implementation on a general-purpose OS is dealing completely with applications that do not comply with the QoS Level model. These can be best-effort applications unaware of the existence of the middleware soft real-time system, broken applications that appear to comply but actually do not (some of which can be handled with dynamic estimate refinement), or rogue applications that intentionally consume more resources than they have requested or been granted. While these issues can be dealt with effectively in a dedicated system (such as an soft real-time web proxy), they are very difficult to manage in a general-purpose environment. The simplest mechanism for dealing with such applications is a resource usage enforcer. Several systems such as RT-Mach with Reserves [25] and Rialto [21] provide mechanisms for enforcing the resource usage of applications, and the system presented herein should run well on top of such a system.

All of the results presented in this paper were gathered from real systems and validated with real soft real-time applications. These results are reliable and not subject to the problems that sometimes arise with untested theoretical results. Most of the experiments were conducted on Solaris and Linux, two different implementations of UNIX. Other than performance differences between the two OS platforms and minor API differences requiring minor rewriting of some of the software, no differences were noted in the results gathered on the two systems.

7.1. Future Work

This work has spawned many research ideas: There is ample reason to believe that the system will work well on top of an OS the enforces resource usage, e.g., RT Mach or Rialto. One avenue for future work is to port the system to such a platform, noting the changes that will have to take place to operate in such an environment, and run the system. Providing a dynamic QoS manager for such an environment will be a significant result in and of itself, and a comparison of the performance of applications and allocation algorithms in such an environment with the results generated on best-effort systems would be valuable contribution to the field.

This work showed some of the strengths and weakness of various allocation algorithms including distributed versus centralized resource management, brute force optimal vs. approximation algorithms, etc., but is not conclusive as to what is the best possible resource allocation algorithm. Since the general resource allocation algorithm is

known to be NP-complete, we derived metrics for comparing algorithm performance in other work [4]. However, this work pointed out the need for additional refinement and analysis of allocation algorithms.

Given the high degree of similarity between soft real-time and networked QoS, there is a number of obvious applications of this work to networking. Other researchers are exploring similar approaches to networked (e.g., [23] and [26]). Thus, there is every reason to believe that the application of the techniques, algorithms, and theory developed in this context could be applied to the domain of networked QoS.

Finally, this work has dealt with a single resource, CPU usage. A fully general solution to this problem must necessarily take into account all relevant resources simultaneously. This includes CPU and network bandwidth, and may also include memory, secondary storage resources, video processing, etc. Resource allocation theory provides a general model for multi-resource problems, but these algorithms are even more complicated than the single-resource versions and are even more infeasible for run-time resource allocation decision-making in real systems. Simple multi-resource decision-making algorithms will have to be developed before this will be feasible.

8. 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*, June 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*, June 1998.

[3] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification, *RFC 2205*, September 1997.

[4] S. Brandt, "Dynamic Soft Real-Time Processing with QoS Level Resource Management", Ph.D. Thesis, University of Colorado, July 1999.

[5] S. Brandt, G. Nutt, T. Berk, and J. Mankovich, "A Dynamic Quality of Service Middleware Agent for Mediating Application Resource Usage", *Proceedings of the 19th IEEE Real-Time Systems Symposium*, December 1998.

[6] 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.

[7] S. Brandt, G. Nutt, and K. Klingenstein, "A Discrete and Dynamic Approach to Application/Operating System QoS Resource Management", *Proceedings of the 1st Internet2 Joint Application/Engineering QoS Workshop*, May 1998.

[8] K. Chen and P. Muhlethaler, "A Scheduling Algorithm for Tasks Described by Time Value Function", *Real-Time Systems*, Vol. 10, 1996.

[9] H. Chu and K. Nahrstedt, "A Soft Real Time Scheduling Server in UNIX Operating System", *Proceedings of the European Workshop on Interactive Distributed Multimedia Systems and Telecommunication Ser-*

vices, September 1997.

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

[11] 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*, August 1995.

[12] 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, February 1997.

[13] D. Ferrari, A. Gupta, and G. Ventre, "Distributed Advance Reservations of Real-Time Connections", *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, April 1995.

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

[15] 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*, December 1997.

[16] T. Ibaraki and N. Katoh, *Resource Allocation Problems, Algorithmic Approaches*, The MIT Press, Cambridge, Massachusetts, 1988.

[17] K. Jeffay, F. Smith, A. Moorthy, and J. Anderson, "Proportional Share Scheduling of Operating System Services for Real-Time Applications", *Proceedings of the 19th Real-Time Systems Symposium*, December 1998.

[18] K. Jeffay and D. Bennett, "A Rate-Based Execution Abstraction for Multimedia Computing", *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, April 1995.

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

[20] C. Jones and D. Atkinson, "Development of Opinion-Based Audiovisual Quality Models for Desktop Video-Teleconferencing", *Proceedings of the 6th International Workshop on Quality of Service*, May 1998.

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

[22] J. Kay and P. Lauder, "A Fair Share Scheduler", *Communications of the ACM*, Vol. 31, No. 1, January 1988.

[23] R. Kravets, K. Calvert, and K. Schwan, "Payoff Adaptation of Communication for Distributed Interactive Applications", *Journal for High Speed Networking: Special Issue on Multimedia Networking*, to appear.

[24] C. Lee and D. Siewiorek, "An Approach for Quality of Service Management", CMU Technical Report

#CMU-CS-98-165, October 1998.

[25] 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*, May 1994.

[26] K. Nahrstedt, H. Chu, S. Narayan, "QoS-aware Resource Management for Distributed Multimedia Applications", *Journal on High-Speed Networking*, 1998.

[27] K. Nichols, V. Jacobson, and L. Zhang, "A Two-bit Differentiated Services Architecture for the Internet", Internet Draft, December 1997.

[28] 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.

[29] 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.

[30] R. Rajkumar, C. Lee, J. Lehoczky and D. Siewiorek, "A Resource Allocation Model for QoS Management", *Proceedings of the 18th IEEE Real-Time Systems Symposium*, December 1997.

[31] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "Practical Solutions for QoS-Based Resource Allocations", *Proceedings of the 19th IEEE Real-Time Systems Symposium*, December 1998.

[32] S. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing, 1997.

[33] D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, "A Feedback-Driven Proportion Allocator for Real-Rate Scheduling", *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, 1999.

[34] B. Teitelbaum, J. Sikora, and T. Hanss, "Quality of Service for Internet2", *Proceedings of the 1st Internet2 Joint Application/Engineering Workshop*, May 1998.

[35] 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*, November 1993.

[36] H. Tokuda, T. Nakajima and P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System", *Proceedings of USENIX Mach Workshop*, October 1990.