Table-based QoS Control for Embedded Real-Time Systems

SUGAWARA Tomoyoshi TATSUKAWA Kosuke
C&C Media Research Laboratories, NEC Corporation
Kawasaki, JAPAN
{sugawara,tatsu}@ccm.cl.nec.co.jp

ABSTRACT

This paper proposes a new QoS control scheme that is suited for embedded real-time systems. Our scheme focused on real-time systems where both device control and multimedia processing are required. Such systems needs to keep timing constraints of control tasks while providing the highest possible quality of service(QoS) to multimedia processing tasks. Although many QoS control schemes are proposed and used in distributed multimedia systems, they are not suited for such real-time systems. Their QoS control policies cannot exactly keep the timing constraints of control tasks.

To overcome this problem we chose a scheme which uses a table describing resource requirements of all tasks. Resource allocations for tasks and total resource utilization in a system can be calculated from the table. Using our scheme, any QoS control policies, such as a fair-share policy or a priority based policy, can be implemented. In other words, it has become possible for the first time to allow the intention of system designers to be directly reflected on QoS control.

We have implemented a CPU time QoS control mechanism based on our proposed scheme and evaluated it on a μ-ITRON Ver.3.0 based real-time OS. The evaluation results demonstrate that the QoS control scheme can keep deadline misses low and CPU utilization high under an overloaded state. The results also show that overhead of the QoS control mechanism is small enough to support both multimedia and control applications.

Keywords

QoS control, embedded real-time system, resource allocation, device control, multimedia processing.

1 INTRODUCTION

Recently demands for multimedia services are increasing in embedded real-time systems, such as vehicle navigation systems. These real-time systems include both hard real-time tasks to control devices like CD-ROM or various sensors, and soft real-time tasks to process video and voice. Such systems must keep timing constraints of hard real-time tasks, while providing the highest possible quality of service(QoS) to multimedia tasks. However, there were no QoS control schemes for such embedded real-time systems to satisfy this requirement.

In distributed multimedia systems, many QoS control schemes [1][2][3][5][9] are proposed. These QoS control schemes, in order to efficiently use system resources, automatically reallocate resources among client programs in response to variations of server and network loads. However, they are not suited for embedded real-time systems. One reason is that the schemes are too complex to be adopted for embedded real-time systems, because distributed systems include an unspecified number of clients and it is difficult to determine resource requirements of all clients in advance. The other reason is that the schemes cannot accurately treat hard real-time tasks because the tasks always need to keep their resource and timing constraints and cannot accept the automatic resource reallocation.

This paper proposes a new QoS control scheme suited for embedded real-time systems. Our scheme uses a table for resource allocation. This table is called "QoS table." The QoS table contains resource requirements of all tasks in a system. It is used for resource allocation to tasks and estimation of overall resource requirement of the system. The QoS table allows the system designer to specify a QoS control policy.

The rest of this paper is organized as follows. In Section 2, we refer to related works. Section 3 provides our QoS control scheme and an implementation of QoS control mechanism for CPU time, and Section 4 demonstrates the effects of the QoS control mechanism. Finally, Section 5 concludes this paper.

2 RELATED WORK

Many QoS control schemes [1] [2] [3] [9] are proposed for distributed multimedia systems. The scheme aims at a system that contains multimedia applications and non-real-time applications. Their policies are to fairly or proportionally share resources among all applications and to protect resource allocated to one application from other applications. So that the QoS control schemes mainly check QoS violation and downgrade QoS of a violator.

In [5], Lee et al. proposed a practical QoS control scheme that uses resource reservation facilities[7] provided by the Real-Time Mach kernel. The proposed scheme allows applications to specify a quality adjustment policy, an admission control policy and an overrun control policy as well as resource reservation parameters which contains computation time and an interval of execution. Moreover, applications can specify quality adjustment priority that is associated with each resource reservation. The priority specifies the global priority at which quality of the application will be adjusted. However, a priority based scheme is not flexible because resource allocation of lower priority reservation will be always decreased before that of higher priority reservation.

Rosu et al.[8] proposed an adaptive resource allocation model for high-performance distributed real-time applications, such as radar systems. Each application task has a set of configurations that contain predetermined resource requirements. One of the configurations is selected to satisfy application's timing requirements. This model resembles our scheme in that resource requirements are predetermined and one of them is selected at run time. However, because evaluation function is used for selecting a configuration, it is difficult to reflect designer's intention to the selection.

3 TABLE-BASED QOS CONTROL

In this section, we introduce our QoS control scheme and describe the implementation of QoS control over CPU time.

3.1 QoS Control Scheme

Our QoS control scheme is based on the idea that quality level of an application task is discrete, not continuous, and the task has a different resource requirement for each quality level. In our QoS control scheme, the system maintains a single value which represents the quality level of the overall system. We call this value "QoS level." Resource allocation to each task is determined according to QoS level. QoS level is downgraded or upgraded in response to variations of resource utilization in a system.

We assume that resource requirements of a task is roughly predictable. In the design phase of a real-time system, the system designer makes a table that contains resource requirements of all tasks in the system. This table is called "QoS table". Each row in the QoS table corresponds to QoS level. Each field in the QoS table represents the resource requirement for QoS level.

In the QoS control scheme, predetermined upper limit and lower threshold of resource utilization trigger QoS control. When overall resource utilization in the system exceeds the limit, QoS level is downgraded and then resource allocations of some tasks are decreased according to the QoS table. On the other hand, when unused resource exceeds the threshold, QoS level is upgraded and resource allocations of some tasks are increased according to the QoS table.

Example of a QoS table is shown in Table 1. This QoS Table is designed for QoS control over CPU time. The left most column of the table represents QoS level values and the other columns represent resource requirements of tasks. In this case, the values of the QoS table represent CPU utilization. CPU utilization U is calculated as $U(\%) = 100 * C/T$. For a periodic task, T is the period and C is the computation time in the period. For a non-periodic task, T

QoS Level	Task A	Task B	Task C	Task D
0	↑	↑	↑	10%
1	20%	↑	↑	↑
2	↑	10%	↑	↑
3	↑	20%	0%	↑
4	↑	↑	↑	20%
5	30%	↑	↑	↑
6	50%	30%	10%	40%

Table 1: Example of QoS Table

is the interval time of execution and C is the computation time per execution. In this case, CPU utilization of Task A is 50% when QoS level is at 6, 30% when QoS level is between 2 and 5 and 20% when QoS level is at 0 or 1. Note that CPU utilization of Task C is 0% when QoS level is between 0 and 3, this means Task C cannot execute in these QoS levels.

We assume that all the tasks are scheduled using rate monotonic algorithm[6]. This means that tasks are scheduled by fixed-priority scheduling, all task are periodic and independent, and a task with shorter interval time is given higher priority of execution. In this case, all the task are schedulable if the total CPU utilization is less than a threshold. In other words, by reducing total CPU utilization under the threshold, all tasks can meets its deadline. In addition, to simplify admission control, we assume that the periods of tasks are multiples of the shortest period. In this case, we can use 100% as the threshold.

Let us describe detailed QoS control process using the sample QoS table. First, suppose tasks A, B and C are executing in the system. In this case, from the QoS table, total CPU utilization is $50 + 30 + 10 = 90\%$. Because the overall system CPU utilization is allowed up to 100%, these three tasks are acceptable with QoS level at 6. Next, if task D newly becomes runnable, total CPU utilization becomes 130% with QoS level at 6, and exceeds the limit. In this case, QoS level is downgraded to 4 at which total CPU utilization of the four tasks is 90%. As a result, CPU utilization of task A is reduced from 50% to 30% and task D is executed with 20% of CPU utilization.

Another case, when tasks A, B and C are executing with QoS level at 6, suppose task A cannot meet its deadline. In this case, though a calculative total CPU utilization is 90%, the actual CPU utilization probably exceed a system limit. In order to decrease CPU utilization, QoS level is downgraded until task A can meet its deadline. QoS level is first downgraded to 4, and next it is downgraded to 3 if deadline misses still occurs.

On the other hand, when idle CPU time expands, QoS level is upgraded to increase overall CPU utilization.

In this example, the QoS table contained only tasks, but it can also contains interrupt handlers.

This scheme is simple and flexible. Any QoS control policies, such as fair-share policy or priority based policy, can be implemented using the scheme. Other examples of QoS table are shown in Table 2 and Table 3. Table 2 represents a QoS table that implements fair-share policy by allocating same resources to all tasks at any QoS level. This policy can be used when all tasks are multimedia processing tasks. On the other hand, Table 3 represents a QoS table that implements priority-based policy. This policy can be used when the priority of a task is important.

QoS Level	Task A	Task B	Task C	Task D
0	20%	20%	20%	20%
1	25%	25%	25%	25%
2	30%	30%	30%	30%
3	40%	40%	40%	40%
4	50%	50%	50%	50%
5	60%	60%	60%	60%
6	100%	100%	100%	100%

Table 2: QoS Table with Fair-share Policy

QoS Level	Task A	Task B	Task C	Task D
0	↑	20%	↑	↑
1	↑	↑	10%	↑
2	↑	↑	20%	↑
3	↑	↑	↑	0%
4	↑	↑	↑	10%
5	↑	↑	↑	20%
6	50%	30%	30%	30%

Table 3: QoS Table with Priority Based Policy

Because a QoS table can contain any type of resource requirement, the QoS control scheme can be adopted to any type of resources, CPU time, file I/O, network, and memory.

3.2 QoS Control Mechanism for CPU Time

We implemented a QoS control mechanism that treats CPU time using our scheme. The QoS control mechanism is composed of the following components. The structure of the QoS control mechanism is depicted in Figure 1.

QoS Control Module A module to control QoS level and resource allocation. This module also maintains the QoS table.

QoS Handler A handler function resides in an application task and it is called to change behavior of the task when the resource allocation has changed.

Deadline Monitoring Module A module to detect deadline misses. This module notifies the QoS control module that deadline misses have occurred in order to downgrade QoS level.

Waste Detection Module A module to measure free CPU time. If free CPU time exceeds a predetermined threshold, this module notifies the QoS control module in order to upgrade QoS level.

The QoS control mechanism was implemented on the NEC RX830, μ-ITRON Ver.3.0[4] based real-time kernel.

3.3 QoS Control Module

The QoS control module is a central components of our QoS control mechanism. This module changes CPU time allocation to tasks and control QoS level of the system. The module does the following work.

- Downgrading QoS level when the system is in overloaded state.

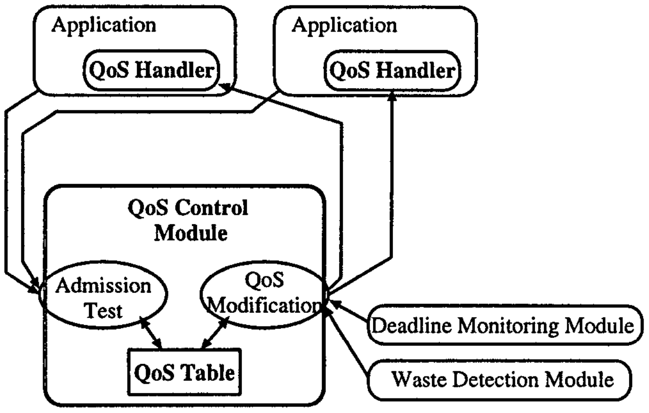

Figure 1: The Components of the QoS Control Mechanism

- Upgrading QoS level when the system has more idle CPU time than a predetermined threshold.
- By an admission test, examining that all runnable tasks are acceptable in new QoS level
- Finding tasks where CPU utilization changes by adjusting QoS level.
- Notifying QoS modification to tasks.

The QoS control modules is woken up by the following events.

- Notification from the deadline monitoring module.
- Notification from the waste detection module.
- Starting a task execution.
- Terminating a task execution.

Currently, the QoS control module is implemented as a task, not as part of the kernel. The QoS control task has higher priority than any application tasks, because the QoS control task must be executed in overloaded state.

3.4 Deadline Monitoring Module

CPU overload can be detected by monitoring deadline misses, because response time of a task becomes worse in overloaded state. The deadline monitoring module checks whether deadline misses have occurred and then notifies the QoS control module of the occurrence of deadline misses. The QoS control module will downgrade QoS level to reduce total CPU utilization.

Deadlines are specified by application tasks as relative time from the beginning of the periods of the tasks, or allowable elapsed time for a set of computations.

The deadline monitoring module can be implemented either as part of kernel functions or a separated module from the kernel. We implemented the deadline monitoring module using the cyclic handler function of μ-ITRON Ver.3.0 that enables a periodic execution of user-defined functions.

3.5 Waste Detection Module

When a system has enough free CPU time, it should be utilized to provide better QoS for tasks. In order to achieve this, the waste detection module measures idle CPU time over the system. If the idle CPU time exceeds a predetermined threshold, the module notifies the QoS control module that the system has enough free CPU time to exploit. The QoS control module will upgrade QoS level to increase the total system CPU utilization.

We implemented the waste detection module as a task with the lowest priority. The task repeatedly execute a loop and measures elapsed time per fixed number of times. Free CPU utilization can be calculated by dividing ideal elapsed time by the measured elapsed time, where the ideal elapsed time is elapsed time when no other tasks and only the loop is executed. For example, if ideal elapsed time is 10ms and measured elapsed time is 100ms, then idle CPU utilization is 10%.

3.6 QoS Handler

A QoS handler is called when the QoS control module changes CPU time allocation to a task. The task must change the amount of computation by using an alternate algorithm or changing the rate at which a task is executed. The handler is part of an application task and written by a developer of the application. A QoS handler is registered to the QoS control module before a task starts its execution. The QoS handler is called with new QoS level of the system as its argument.

4 EVALUATION RESULTS

In this section, we present evaluation results of our QoS control scheme. We have measured the number of deadline misses and overall system CPU utilization with and without the QoS control mechanism for CPU time. The overhead against the number of tasks is also presented.

4.1 Evaluation System

We implemented and executed our QoS control programs and evaluation programs on an evaluation board for RX830. The specification is as below.

```
Name: RTE-V831-PC(Midas Lab. Corp., Japan)
CPU: NEC V831(including V830 core)
     100MHz 118MIPS
Bus clock rate: 33MHz
Cache: 4KB(Instruction), 4KB(Data)
DRAM: 8MB
EPROM: 128KB
Flash ROM: 8MB
```

All programs are linked into one object and it is downloaded from front-end PC through a ROM emulator. The elapsed time and CPU utilization are measured by performance analyzer running on the PC.

4.2 Effect of QoS Control

First, we show effect of our QoS control mechanism. The purpose of CPU time QoS control is reducing overall CPU utilization in system overloaded state as quickly as possible, while keeping overall CPU utilization high. We measured the number of deadline misses to validate that the QoS control mechanism could cope with system overloaded state. We also measured total CPU utilization of the system to check that QoS control mechanism could efficiently exploit CPU time.

In this evaluation, one high priority task(HP task) representing a control task, four low priority tasks(LP tasks) representing multimedia processing tasks and a clock interrupt handler are always executed. Then, an additional load is added to the system. We have caused overloaded state by using the following three different types of load.

1. Interrupt handling
2. HP task
3. LP task

CPU time allocations for LP tasks are changed by QoS control, while those of HP tasks and an interrupt handler does not vary during execution. All tasks and interrupt handlers are executed periodically by means of the cyclic handler function of RX830.

4.2.1 The Influence of Interrupts

In embedded realtime systems, frequent interrupts can easily interfere with task execution. The first results show the influence of interrupts.

We assume non-periodic interrupts in this case. Overloaded state caused by such interrupts cannot be predicted because occurrence of interrupts cannot be expected beforehand. The QoS control mechanism does not function until a deadline miss is detected. We measured three types of results using no QoS control: Guaranteed, Middle and Best effort.

Guaranteed The CPU time necessary for interrupt handling is allocated in advance, remaining CPU time is shared among tasks. All tasks can be successfully executed, if additional interrupts occur.

Best effort All CPU time is shared to tasks without consideration to interrupts. Some tasks can fail, if additional interrupts occur.

Middle Half of CPU time necessary for interrupt handling is allocated in advance, remaining CPU time is allocated to tasks. Some tasks can fail, if additional interrupts occur.

Table 4 shows task attributes and contents of QoS table. Task attributes include period of execution, relative deadline from the beginning of a period and CPU utilization. For LP tasks, three types of attributes are defined. The three types of attributes, Guaranteed, Middle and Best effort, respectively corresponds to the three case when no QoS control is used. In this experiment, the additional interrupts require 35% of CPU utilization for a duration of 500ms. The values in these tables are not derived from a particular application, but learned from our experience. For example, an interval time and a CPU utilization of LP tasks are associated with multimedia processing that display twenty frames of images per one second.

Table 5 shows the results under the influence of interrupts. The first column shows CPU utilization in non-overloaded state and second column shows CPU utilization in overloaded state. The last column shows the number of deadline

	Period	Deadline	CPU Utilization
HP Task	10ms	–	15%
Timer Interrupt	1ms	–	3%
LP Task 0-3(Guaranteed)	50ms	49ms	11%
LP Task 0-3(Middle)	50ms	49ms	15%
LP Task 0-3(Best effort)	50ms	49ms	20%
Additional Interrupt	1ms	–	35%

QoS Level	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HP Task	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
Timer Interrupt	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
LP Task 0	5	5	5	5	10	10	10	10	15	15	15	15	20	20	20	20
LP Task 1	0	5	5	5	5	10	10	10	10	15	15	15	15	20	20	20
LP Task 2	0	0	5	5	5	5	10	10	10	10	15	15	15	15	20	20
LP Task 3	0	0	0	5	5	5	5	10	10	10	10	15	15	15	15	20
Additional Interrupt	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35

Table 4: Task attributes and QoS Table (The interrupt case)

	QoS Control	Guaranteed	Middle	Best effort
CPU Util. (Non-overload) (%)	93.9	58.9	74.9	94.9
CPU Util. (Overload) (%)	91.4	93.9	94.9	89.9
DL Misses (LP Task 0/1/2/3)	0/0/1/4	0/0/0/0	0/0/0/10	0/0/10/10

Table 5: Results under the Influence of Interrupts

misses of LP tasks. Notation "$n_1/n_2/n_3/n_4$" represents that n_1, n_2, n_3 and n_4 deadline misses occurred in LP task 1, LP task 2, LP task 3 and LP task 4 respectively.

In the three cases using no QoS control, total CPU utilization of the system are nearly at the limit while interrupts are occurring. However, they are much different in completion of task execution. In the best effort case, LP tasks 2 and 3 always failed to complete during the overloaded state. Because CPU time was consumed nearly at its limit even while the interrupts are not occurring, so the CPU time for tasks 2 and 3 were taken away by the interrupts.

In the middle case, only task 3 could not meet its deadline ten times, because 20% of free CPU time, that corresponds to CPU utilization of only one LP task, remained while the interrupt load does not exist. On the other hand, deadline misses does not occur in the guaranteed case because the CPU time for interrupts is reserved in advance.

In contrast, CPU utilization of three cases are apparently different while interrupts are not occurring. About 40% of CPU time is wasted in the guaranteed case, while almost all CPU time is exploited in the best effort case.

On the other hand, in the case of using QoS control, deadline miss occurred once in task 2 and four times in task 3, while the total CPU utilization is from 91.4% to 93.4%. The results demonstrates that our QoS control mechanism can efficiently utilize CPU time while reducing deadline misses.

In order to explain the cause of four deadline misses in task 3, we investigated the transition of QoS level when interrupt load is added and deleted. The result is shown as the timing chart of calculative total CPU utilization in Figure 2. In this figure, the vertical line represents the calculative CPU utilization and the horizontal line represents elapsed time after the interrupt load was inserted. The calculative total CPU utilization is calculated from QoS level and the

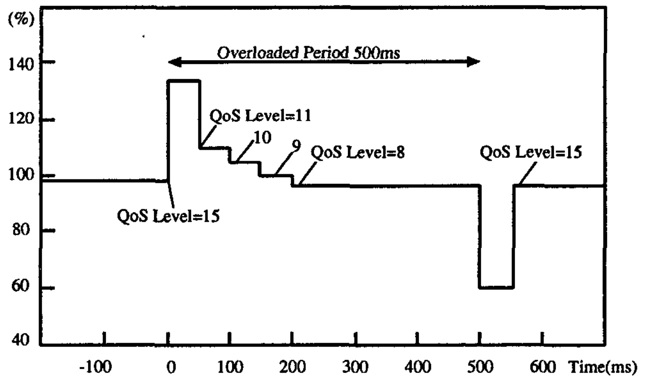

Figure 2: Transition of the calculative total CPU utilization

CPU utilization specified in Table 4.

As mentioned above, when a deadline miss occurs, the QoS control module downgrades QoS level. In order to estimate excess CPU utilization, the module uses CPU utilization of tasks that cannot start execution as a hint. If there are no such tasks, the module can only downgrade the QoS level one by one. As a result, the module can reduce QoS level from 15 to 11 in the first 50ms, but, after that, can downgrade only one level per 50ms. Therefore it takes 200ms to reach appropriate QoS level.

4.2.2 *Influence of a High Priority Task*

The next result shows the influence caused by addition of a HP task. The HP task that is newly added represents a control task with hard real-time constraint. In this case, the system call starts up the additional task. At this point, the QoS control module can check whether the total CPU utilization of the system becomes more than its limit.

The results using QoS control are compared to the results using no QoS control. We have measured result for the three cases, "Guaranteed", "Middle" and "Best effort". In this experiment, the additional HP task requires 35% of CPU utilization for a duration of 500ms.

The task attributes and QoS table are described in Table 6.

Table 7 shows the results. The results using no QoS control are similar to that of the interrupt case. This means that a high priority task has almost the same effect as interrupt handlers for a low priority task by preempting CPU time of a low priority task.

In the case of using QoS control, the results is better than the interrupt case; the deadline misses never occurred, and the total CPU utilization is 94.9%. The reason for the improvement is because the QoS control module can calculate the excess CPU utilization caused by a high priority task in advance.

In the current implementation, when starting a new task, the QoS control module adjusts QoS level to keep the total CPU utilization less than its limit. Therefore no deadline misses occur. When stopping a running task, the module adjusts QoS level so that the total CPU utilization is as high as possible below its limit. Therefore the total CPU utilization in the system does not decrease after stopping tasks.

However, this is an ideal case, because all tasks obey their predetermined computation time. In actual applications, the computation time of tasks are variable, so that it may be difficult to obtain a result like this experiment.

4.2.3 Influence of a Low Priority Task

The last result shows the influence caused by addition of a LP task. The result of using QoS control is compared with two cases using no QoS control. The two cases are "Guaranteed" and "Best effort". In this experiment, the additional LP task requires 16% of CPU utilization in the guaranteed case, 20% of CPU utilization in the best effort case and from 0 to 20% of CPU utilization in the case with QoS control. It runs for 500ms. The "Middle" case are not shown in this evaluation, because the result was almost the same as the "Best effort" case.

The task attributes and QoS table are described in Table 8.

Table 9 shows the result. In the guaranteed case, no deadline misses occurred, but CPU utilization is less than 80%. In the best effort case, the additional LP task always fails to complete.

In the case of using QoS control, only one deadline miss occurred and the total CPU utilization is from 90.9 to 94.9. The deadline miss is caused by slight excess of total CPU utilization due to the overhead of QoS control mechanism and other system operations, such as context switch. Because the deadline miss caused downgrade of QoS level, the total CPU utilization in overloaded state is lower than in non-overloaded state. We ensured that QoS level is at the unexpected low level just after the QoS control, but it returned to the expected level within 300ms.

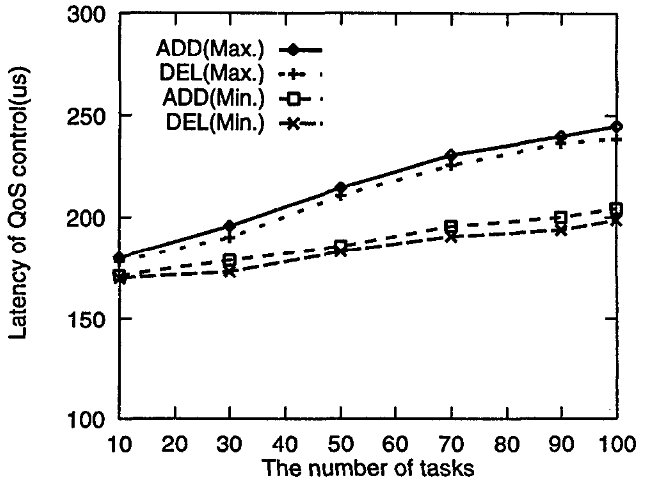

Figure 3: QoS Control Overhead v.s. Number of Tasks

4.3 Overhead of QoS Control

We also measured the overhead imposed by the QoS control mechanism for various number of tasks.

Figure 3 shows processing time within the QoS control module for various number of tasks. This graph contains the result of two types of QoS control operations, ADD and DEL. The QoS control module performed the following works when these operations called.

- Added an entry to QoS table or deleted an entry from QoS table.
- Downgraded or upgraded QoS level by one.
- Performed an admission test at new QoS level.
- Found one task that can be changed resource allocation and decreased or increased its resource allocation
- Called QoS handler to change QoS of the task.

The minimum and maximum latencies of both operations are plotted. In order to remove the effect of the CPU cache, the cache is flushed before each operation.

The result shows the latency is less than 250us in the worst case. This means that the impact of QoS control on performance is quite small, because overhead of the QoS control mechanism is much less than a period of a typical multimedia processing task (about 25ms), and the overhead is also less than response time of a control task, such as CD-ROM control and sensor monitoring (more than 10ms). Therefore the QoS control mechanism can efficiently support real-time systems that contains multimedia processing tasks and device control tasks.

The result also shows that all latencies increased in proportion to the number of tasks, and the maximum latency increases rapidly than the the minimum latency. The reason for the difference is search operations to find eligible tasks. In the current implementation, the QoS control module searches a bitmap created from the QoS table in order to find the task whose resource allocation can be modified. Since this bitmap size increases in proportion to the number of tasks and QoS levels, maximum search time also increases.

	Period	Deadline	CPU Utilization
HP Task	10ms	–	15%
Timer Interrupt	1ms	–	3%
LP Task 0-3(Guaranteed)	50ms	49ms	11%
LP Task 0-3(Middle)	50ms	49ms	15%
LP Task 0-3(Best effort)	50ms	49ms	20%
Additional HP Task	10ms	–	35%

QoS Level	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HP Task	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
Timer Interrupt	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
LP Task 0	5	5	5	5	10	10	10	10	15	15	15	15	20	20	20	20
LP Task 1	0	5	5	5	5	10	10	10	10	15	15	15	15	20	20	20
LP Task 2	0	0	5	5	5	5	10	10	10	10	15	15	15	15	20	20
LP Task 3	0	0	0	5	5	5	5	10	10	10	10	15	15	15	15	20
Additional HP task	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35

Table 6: Task attributes and QoS table(The High Priority Task Case)

	QoS Control	Guaranteed	Middle	Best effort
CPU Util. (Non-overload) (%)	94.9	58.9	74.9	94.9
CPU Util. (Overload) (%)	94.9	93.9	94.9	89.9
DL Misses (LP Task 0/1/2/3)	0/0/0/0	0/0/0/0	0/0/0/10	0/0/10/10

Table 7: Results under Influence of the High Priority Task

	Period	Deadline	CPU Utilization
HP Task	10ms	–	15%
Timer Interrupt	1ms	–	3%
LP Task 0-3(Guaranteed)	50ms	49ms	16%
LP Task 0-3(Best effort)	50ms	49ms	20%
Additional LP Task(Guaranteed)	50ms	49ms	16%
Additional LP Task(Best effort)	50ms	49ms	20%

QoS Level	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HP Task	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
Timer Interrupt	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
LP Task 0	5	5	5	5	10	10	10	10	15	15	15	15	20	20	20	20
LP Task 1	0	5	5	5	5	10	10	10	10	15	15	15	15	20	20	20
LP Task 2	0	0	5	5	5	5	10	10	10	10	15	15	15	15	20	20
LP Task 3	0	0	0	5	5	5	5	10	10	10	10	15	15	15	15	20
Additional LP Task	0	0	0	5	5	5	5	10	10	10	10	15	15	15	15	20

Table 8: Attributes of Tasks and QoS Table(The Low Priority Task Case)

	QoS Control	Guaranteed	Best effort
CPU Util. (Non-overload) (%)	94.9	78.9	94.9
CPU Util. (Overload) (%)	90.9	94.9	94.9
DL Misses (LP Task 0/1/2/3/4)	0/0/0/0/1	0/0/0/0/0	0/0/0/0/10

Table 9: Results under the Influence of a Low Priority Task

5 CONCLUSION

We proposed a new QoS control scheme suited for embedded real-time system. The scheme uses a QoS table, that contains resource requirements of tasks, in order to determine resource allocation to the tasks. We have implemented a CPU time QoS control mechanism using our proposed scheme and evaluated it on an evaluation board with μ-ITRON Ver.3.0 based real-time OS. The evaluation results shows that the QoS control mechanism achieves an efficient CPU utilization at more than 90%, while reducing deadline misses to a few times, in system overloaded state. The results also shows that the latencies of QoS control operations are small enough for multimedia applications.

In future, we will try to apply the QoS control scheme to other resources, such as memory, file I/O or network.

ACKNOWLEDGEMENTS

We thank Hirokazu Tamura, Noriaki Takakura and Kazuya Hashimoto for providing test and evaluation environment to this research. We are grateful to Yousuke Takano and Toshiyuki Nakata for comments and reviews of this paper.

References

[1] G. Coulson and G. Blair. Architectural principles and techniques for distributed multimedia application support in operating systems. *ACM Operating Systems Review*, 29(4):17–24, October 1995.

[2] H. Fujita, T. Nakajima, and H. Tezuka. A processor reservation system supporting dynamic qos control. In *International Workshop on Real-Time Computing Systems and Applications*, October 1995.

[3] R. Gopalakrishnan and G. Parulkar. A real-time upcall facility for protocol processing with qos guarantees. In *ACM Symposium on Operating Systems Principles*, December 1995.

[4] ITRON Technical Committee. The ITRON specifications. http://www.ertl.ics.tut.ac.jp/ITRON/eng-spec.html.

[5] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic qos in real-time mach. In *International Symposium on Multimedia Systems*, 1996.

[6] C. L. Liu and J. W. Layland. Scheduling algorithms for multi-programing in a hard-real-time emvironment. *ACM*, 20(1), 1973.

[7] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Os support for multimedia applications. In *IEEE International Conference on Multimedia Computing Systems*, May 1994.

[8] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On adaptive resource allocation for complex real-time applications. In *IEEE Realtime System Symposium*, pages 320–329, December 1997.

[9] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *the Symposium on Operating Systems Design and Implementation*, Nov. 1994.