Q-Thread: A New Execution Model for Dynamic QOS Control of Continuous-Media Processing*

Kiyokuni KAWACHIYA

IBM Research,
Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato,
Kanagawa 242, Japan
<kawatiya@trl.ibm.co.jp>

Hideyuki TOKUDA

Faculty of Environmental Information,
Keio University
5322 Endo, Fujisawa,
Kanagawa 252, Japan
<hxt@sfc.keio.ac.jp>

Abstract

One characteristic of continuous media is that they should be processed both continuously and periodically. Although "periodic execution models" are suitable for such processing, most existing periodic models do not take account of resource reservation and adaptation (QOS control) in themselves, and such resource management must be performed through some external mechanism. In this paper, we propose a new process-execution model, named "Q-Thread," that incorporates such continuous-media support. By using a Q-Thread, a user can specify the tolerable ranges of the period and computation time for periodic invocations, and can easily write a continuous-media session that controls the QOS dynamically. The Q-Thread execution model has been implemented on RT-Mach, and dynamic QOS-control experiments have been carried out with this prototype.

Keywords: Process-execution model, dynamic QOS control, continuous media, resource management

1 Introduction

In the Keio-MMP (MultiMedia Platform) project [1], we have been extending Real-Time Mach 3.0 (RT-Mach) [2] and constructing a software platform for multimedia processing on our extension [3, 4, 5]. One research topic related to such a multimedia platform is an execution model suitable for continuous-media processing.

Basically, "periodic execution models" seem to be effective for processing continuous media in a computer. However, most existing periodic models focus on real-time processing such as machine control, and do not take account of resource reservation and adaptation (QOS control) for continuous media. Such resource management has to be achieved through some other mechanism outside of the execution model.

In this paper, we propose a new execution model, named "Q-Thread," that incorporates such continuous-media support. Section 2 examines the basic structure and required features of continuous-media processing. Section 3 proposes the Q-Thread execution model, which satisfies the requirements, and Section 4 reports dynamic QOS-control experiments using a prototype implementation of the Q-Thread. After a discussion of related work on multimedia execution models in Section 5, Section 6 offers some conclusions and topics for future work.

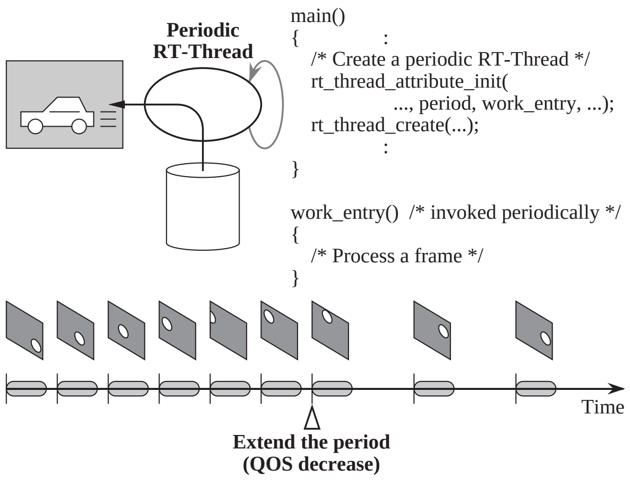
2 Execution Model for Continuous-Media Processing

As the name indicates, one characteristic of continuous media is that they are processed continuously and periodically. Therefore, the "periodic execution model," in which an entry point is called periodically, is suitable for continuous-media processing [6]. In RT-Mach, which is our base software, a periodic real-time thread (RT-Thread) is provided and can be used to perform continuous-media processing, as shown in Fig. 1 [7]. In this example, a media data unit (MDU), such as a frame of video data, is processed on every invocation of `work_entry()`.

However, the periodic RT-Thread was originally designed for real-time processing, and is not entirely suitable for continuous-media processing. Unlike in static real-time processing, for example, where some task set is predefined, it is normal in continuous-media processing for the system environment to be dynamically changed by user interaction and so on. To guarantee the real-time nature of the processing in such an environment, some *resource reservation* mechanism is necessary. The periodic RT-Thread model does have the deadline-miss detection mechanism, but does not guarantee the execution. Such guaranteeing must be managed outside of the execution model by, for example, a CPU reservation mechanism [8, 9].

In continuous-media processing, moreover, the processing amount can be adjusted without violating the timing constraints by changing the quality of service (QOS). For such QOS adjustment, it is desirable that

*This research is conducted under the Open Fundamental Software Technology Project of Information-technology Promotion Agency, Japan (IPA).



```
main()
{              :
   /* Create a periodic RT-Thread */
   rt_thread_attribute_init(
                 ..., period, work_entry, ...);
   rt_thread_create(...);
                 :
}

work_entry()  /* invoked periodically */
{
   /* Process a frame */
}
```

Figure 1: Continuous-media processing using the periodic RT-Thread

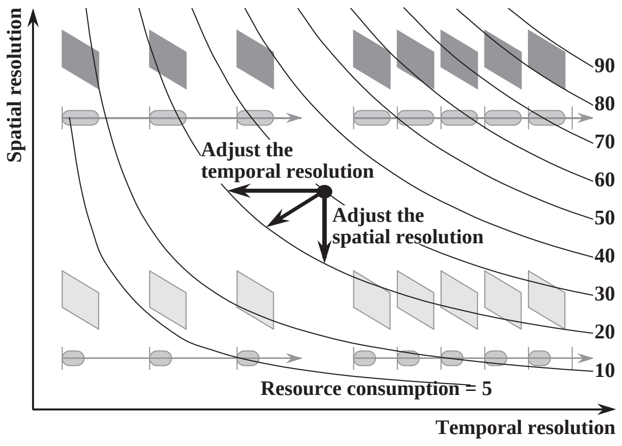


Figure 2: Various ways of changing the QOS

some *resource-use adaptation* mechanism based on actually available resources be provided. Such a mechanism is not included in the periodic RT-Thread model of RT-Mach. Therefore, a user program must explicitly implement the dynamic QOS control by itself, using some mechanism outside of the execution model [10]. As an example of this, the lower part of Fig. 1 shows how the QOS (frame rate) is degraded by explicitly extending the period of the RT-Thread.

To solve these resource-management problems for continuous-media processing, we have developed a new process-execution model, named "Q-Thread." The Q-Thread includes resource reservation and adaptation, and supports a user program for controlling the QOS.

3 A New Process-Execution Model, "Q-Thread"

There are various ways of changing the QOS of continuous media on the basis of the available resources. Two typical ways are adjusting the temporal resolution (e.g., the frame rate) and adjusting the spatial resolution (e.g., the frame quality) [11]. For example, the graph in Fig. 2 illustrates the relations between these resolutions for specific levels of resource consumption, and arrows in the figure show various ways of reducing the QOS when the available resources are decreased from 30 to 20.

In the periodic execution model, these ways of adjusting the QOS correspond to changing the period (invocation interval) of the processing and the amount of computation for each processing unit, respectively. The Q-Thread is an execution model that incorporates this adjustment of period and computation time.

3.1 Overview of the Q-Thread

The Q-Thread can be considered as an extension of the periodic RT-Thread for continuous-media processing with dynamic QOS control. In the periodic RT-Thread, a fixed invocation period is specified as a thread's attribute,[1] but in the Q-Thread, the tolerable *ranges* of the period and computation time (CPU time) for invocations can be specified. The user-specified entry point is called periodically, but the period (invocation interval) is changed automatically and dynamically within the specified range in accordance with the available CPU resource. In addition to this adjustment of the period, information on the available computation time is passed to a user program at each invocation as a hint for workload adaptation. Table 1 gives a comparative summary of the periodic RT-Thread and the Q-Thread.

So long as the user program observes the computation-time indication in the hint, its execution is guaranteed by the CPU reservation mechanism of RT-Mach [8, 9]. This mechanism enables a thread (or a group of threads) for reserving a specific part of the CPU resource in the form of "requires C seconds every T seconds (i.e. $\frac{C}{T}$ of the CPU)."

The CPU-resource allocation among multiple Q-Threads is dynamically managed by a "QOS-Control Server" in accordance with the "QOS-Ticket model" proposed in our previous work [12, 13, 14]. This server calculates the CPU-resource assignment for each continuous-media session in accordance with "QOS factors" registered by the sessions, and "issues" a CPU reserve to each session.

3.2 Implementation of the Q-Thread

Currently, the Q-Thread is implemented on RT-Mach as a user-level library by means of the periodic RT-Thread. Figure 3 shows the structure of the current implementation, consisting of a QOS-Control Server and a Q-Thread library [15]. Both programs are written in C; the QOS-Control Server contains about 1500 lines and the Q-Thread library about 1000 lines.

When a user program indicates the creation of a Q-Thread, the library first translates the ranges of

[1]It is possible to change the period of RT-Thread by re-specifying the thread's attribute. The Q-Thread is implemented by using this function.

Table 1: Comparison of the periodic RT-Thread and the Q-Thread

	Periodic RT-Thread	Q-Thread
Function	User-specified entry point is called periodically.	
Thread's attribute	(Fixed) invocation period, etc.	*Ranges* of the period and computation time, QOS-control policy, etc.
Invocation period	Fixed (can be re-specified)	Dynamically changed within the range based on the QOS-control policy
Arguments passed to the entry point	User-specified argument	Same, plus a QOS-control hint (available computation time, etc.)
Guarantee of execution	None (possible with CPU reservation)	Guaranteed, within the available computation time
Purpose	(Soft/hard) real-time processing	Continuous-media processing (with dynamic QOS control)

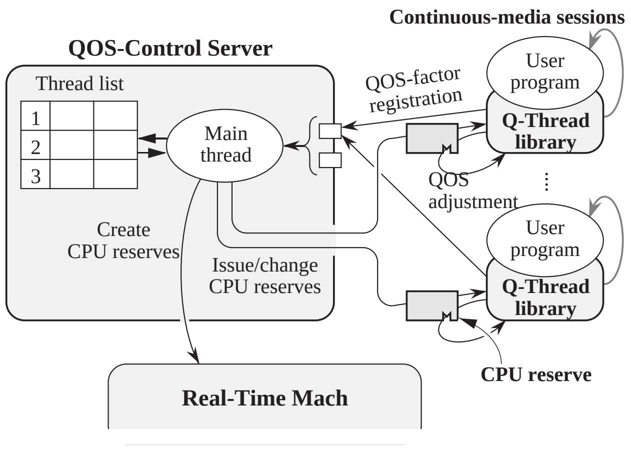


Figure 3: The QOS-Control Server and the Q-Thread library

the period and computation time in the Q-Thread attribute into the range of the required CPU-resource allocation (QOS factor), and registers it with the QOS-Control Server. On the basis of this information, the server allocates and reserves a part of the CPU resource and issues a CPU reserve to the Q-Thread. The available CPU-resource amount indicated in the reserve is reverse-translated into some possible combination of period and computation time, and a user-specified entry point is called with this period. The result is also passed to the user program as a hint on QOS control.

Note that this translation process is not a static one when the Q-Thread is created, but is performed dynamically in accordance with the system environment. For example, when a Q-Thread is newly created or finished, or some Q-Thread's attribute is changed, the CPU-resource assignment to the existing Q-Threads is recalculated by the QOS-Control Server, and the period and/or computation time of these threads are changed.

3.3 QOS-Control Policies of the Q-Thread

Several policies can be considered for reverse-translating the CPU reserve into a possible period and computation time. In the current implementation, the following four "QOS-control policies" are provided:

DEGRADE_PERIOD_FIRST: This policy tries to preserve the maximum computation time, and degrades (extends) the period first in a CPU-resource shortage. This corresponds to preserving the maximum possible spatial resolution.

DEGRADE_COMPUT_FIRST: This policy tries to preserve the minimum invocation period, and degrades (reduces) the computation time first in a CPU-resource shortage. This corresponds to preserving the maximum possible temporal resolution.

DEGRADE_BOTH: This policy degrades the period and computation time equally.

HINT_ONLY: In this policy, the QOS adjustment is totally left up to the user program. The library does nothing other than provide a QOS-control hint.

One of these policies is specified in the Q-Thread attribute.

Figure 4 shows the relation of the range specification and the QOS-control policies. In this example, the range from 30 to 100 msec is specified as the tolerable invocation period, and the range from 10 to 20 msec is specified as the tolerable computation time for invocations. Consequently, the inside of the rectangle in the figure contains all the possible combinations of the period and computation time. How a specific combination should be chosen from the available CPU resource is based on the QOS-control policy, as shown by the three paths in the figure.

The QOS-control policy can be re-specified while the thread is running. In this case, the period and computation time are recalculated by the Q-Thread library.

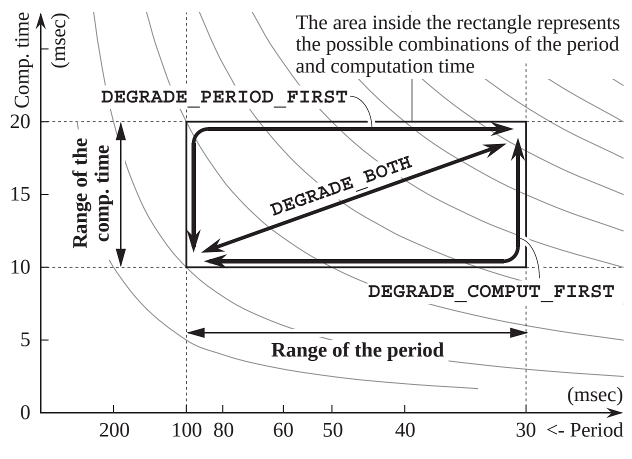


Figure 4: The Q-Thread attribute and the QOS-control policies

```
/*
 * qthread_hint - Hint on QOS adaptation
 */
typedef struct qthread_hint_data {
  int iteration_count;/* Iteration count             */
  int time_from_start;/* Time from start      (msec) */
  int period;         /* Current period       (msec) */
  int comput;         /* Available comp. time (msec) */
  int period_prev;    /* Previous period      (msec) */
  int comput_prev;    /* Previous actual comp. time  */
  int changed;        /* Is the hint changed?        */
  int dl_miss_count;  /* Deadline miss count         */
} qthread_hint_data_t, *qthread_hint_t;
```

Figure 5: Structure of the QOS-control hint

3.4 QOS-Control Hint

The period of a Q-Thread is automatically changed by the library (unless the policy is `HINT_ONLY`). However, the amount of computation performed at each invocation must be adjusted inside the user program. To facilitate this adjustment, a "QOS-control hint" is passed to the user program when the user-specified entry point is called. Figure 5 shows the structure of the QOS-control hint in the current implementation. So long as the user program observes the computation-time indication in the hint (the member `comput` in Fig. 5), its execution is guaranteed by the CPU reserve.[2]

The QOS-control hint also contains information useful for workload adaptation, including the amount of CPU time that was actually consumed at the last invocation (the member `comput_prev` in Fig. 5). This information is calculated by the Q-Thread library from the accumulated CPU-time information in the CPU reserve, and can be used by the user program to adjust its workload to match the available computation time.

In addition, the user program can re-specify (modify) the Q-Thread attribute in accordance with the QOS-control hint. In this case, the Q-Thread library re-registers the QOS factor with the QOS-Control Server. The CPU allocation is then recalculated, and the period and computation time of the thread are changed.

[2]If the user program calls some server program, the CPU reserve is temporarily passed to the server via RT-Mach IPC, and its processing is also guaranteed.

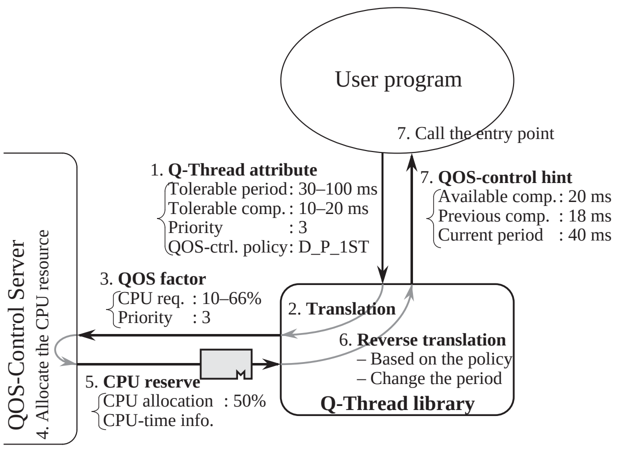


Figure 6: QOS translations by the Q-Thread library

3.5 QOS Translations by the Q-Thread Library

As already mentioned in Section 3.2, one major role of the Q-Thread library is the following "QOS-translation" process:

- Translation of the Q-Thread attribute into the QOS factor, and negotiation with the QOS-Control Server
- Reverse translation of the CPU reserve into a QOS-control hint, and provision of a hint to the user program.

Figure 6 shows an example of this QOS-translation process:

1. A user program specifies tolerable ranges for the period and computation time in its Q-Thread attribute. In this example, 30 to 100 msec is specified for the period and 10 to 20 msec for the computation time.
2. The Q-Thread library translates the Q-Thread attribute into a QOS factor for the session, and registers it with the QOS-Control Server.
3. The translated QOS factor indicates that the session requires 10 (= $\frac{10}{100}$) to 66 (= $\frac{20}{30}$) percent of the CPU resource.
4. The QOS-Control Server allocates the CPU resource on the basis of the QOS factors registered by sessions, and issues a CPU reserve to the session.

5. The CPU reserve indicates that 50% of the CPU resource is assigned to the session. The reserve also contains information on the CPU time used through it.

6. The Q-Thread library reverse-translates the information into a possible period and computation time, in accordance with the QOS-control policy.

7. In this example, the `DEGRADE_PERIOD_FIRST` policy is specified, and a 40 msec period and a 20 msec computation time are selected. On the basis of the calculation, the user-specified entry point is called periodically, with a QOS-control hint as its argument.

3.6 A User Program with the Q-Thread

By using the Q-Thread, a user can easily write a continuous-media session that controls the QOS dynamically. All exported functions of the Q-Thread library are listed in Table 2.

Figure 7 shows a sample user program using the Q-Thread. In this example, the user-specified entry point `qt_entry()` is called periodically with periods from 30 to 100 msec. The available computation time (CPU time) for each invocation is indicated inside the second argument, `qt_hint->comput`, and the user program must change the amount of its work on the basis of this information. Although not shown in the example, the user program can also modify the range of tolerable computation time by using the `qthread_set_attribute()` function, in accordance with the previous actual computation time given by `qt_hint->comput_prev`.

4 Experiments on Dynamic QOS Control Using the Q-Thread

As mentioned in Section 3.2, the Q-Thread execution model has already been implemented on RT-Mach, and experiments in dynamic QOS control have been carried out to show the effectiveness of the model.

4.1 Experiment with Dummy Q-Threads

The first experiment was carried out to confirm the basic efficiency of the Q-Thread. In this experiment, three dummy threads written by using the Q-Thread were used. The structure of the dummy thread is almost the same as that of the sample program shown in Fig. 7. When the entry point is called, these dummy threads use up the CPU resource for the amount of the available computation time indicated in the QOS-control hint.[3] The Q-Thread attributes of the dummy threads are shown in Table 3. All threads specify the same ranges of the period (30 to 100 msec) and of the computation time (10 to 20 msec), but the priority (a high value means high priority) and the QOS-control policy are different.

Dummy thread 1 runs from 0 to 70 sec. `DEGRADE_-PERIOD_FIRST` is initially specified as the QOS-control policy, but this is changed to `DEGRADE_COMPUT_FIRST`

[3]In fact, this workload adjustment should be based on information about the previous computation time and so on, but, this mechanism is omitted in the experiment.

Table 2: Functions provided by the Q-Thread library

Name	Description
`qoslib_init()`	Initializes the Q-Thread library
`qthread_attribute_init()`	Initializes a Q-Thread attribute
`qthread_create()`	Creates a Q-Thread
`qthread_get_comput()`	Gets the consumed CPU time
`qthread_get_attribute()`	Retrieves a Q-Thread attribute
`qthread_set_attribute()`	Re-specifies the attribute
`qthread_terminate()`	Terminates a Q-Thread

```
/*
 * qthread_sample.c - Q-Thread Sample Program
 */

#include "qoslib.h"

/*
 * Periodic entry point
 */
void qt_entry(
  int            *arg,   /* User-specified argument     */
  qthread_hint_t qt_hint/* QOS-control hint             */
)
{
  DO_SOME_WORK(qt_hint);
        /* qt_hint->comput indicates the available
           computation time (msec)                      */
        /* qt_hint->comput_prev gives the actual
           computation time at the last invocation     */
}

/*
 * Main routine
 */
int main(int ac, char **av)
{
  qthread_t            qthread;
  qthread_attr_data_t  qt_attr;
  kern_return_t        kr;

  kr = qoslib_init();  /* Initialize the Q-Thread lib. */
  kr = qthread_attribute_init(/* Set up a Q-Thread attr*/
         qt_entry,NULL,/* Periodic entry point and arg.*/
         30,100,       /* Range of the period (msec)   */
         10,20,        /* Range of the comp.time (msec)*/
         3,            /* Priority                     */
         DEGRADE_PERIOD_FIRST,/* QOS-control policy    */
         NULL,NULL,    /* Deadline handler and its arg.*/
        &qt_attr);
  kr = qthread_create(mach_task_self(),&qthread,&qt_attr);
                       /* Create a Q-Thread            */
  thread_terminate(mach_thread_self());
}
```

Figure 7: Sample user program using the Q-Thread

Table 3: Q-Thread attributes of the dummy threads

No.	Exec. term (sec)	Period (msec)	Comp. (msec)	Priority	QOS-ctl. policy
1	00 → 30	30 – 100	10 – 20	2	`D_P_1ST`
	30 → 70				`D_C_1ST`
2	10 → 80	30 – 100	10 – 20	3	`D_P_1ST`
3	20 → 40	30 – 100	10 – 20	3	`D_BOTH`
	40 → 80			6	
D	50 → 60	(Disturbing thread that exhausts the CPU by looping infinitely)			

after 30 sec. Dummy thread 2 runs from 10 to 80 sec, and has a higher priority than thread 1. The QOS-control policy of thread 2 is `DEGRADE_PERIOD_FIRST` from start to end. Dummy thread 3 runs from 20 to 80 sec. It has the same priority as thread 2 initially, but a different QOS-control policy, `DEGRADE_BOTH`, is specified, and at 40 sec, the priority of thread 3 is upgraded. In addition to these dummy threads, a "disturbing" thread runs from 50 to 60 sec. This thread tries to exhaust the CPU by looping infinitely.

We observed the behavior of dynamic QOS control by executing these threads with the QOS-Control Server. The experiment was carried out on an IBM ThinkPad 755C (9545-L, IntelDX4-75MHz). The version of RT-Mach was MK94[4] and the resolution of the system clock was changed to 1 msec. Figure 8 shows the results. The upper part of this figure shows the actual period (i.e., the temporal QOS) of each dummy thread, and the lower part shows the CPU utilization and deadline misses during the experiment. The available computation time (i.e., the spatial QOS) indicated by the Q-Thread library is also shown in the upper graph by expressions of the form "c=XX (msec)."

In the first 10 seconds, only thread 1 was running, and the highest QOS (30 msec period and 20 msec computation time) was achieved. But after higher-priority threads 2 and 3 were started at 10 and 20 sec, the period of thread 1 was extended to about 90 msec. Threads 2 and 3 had the same priority, but their QOS-control policies were different. Therefore, thread 3 was able to run with a shorter period than thread 2, but its available computation time was less than that of thread 2.

After 30 sec, the QOS-control policy of thread 1 was changed to `DEGRADE_COMPUT_FIRST` (maximize the temporal resolution). Consequently, the period of thread 1 was reduced and the computation time was decreased. In this case, the QOS factor was not re-registered, so the change had no influence on other threads. After 40 sec, the priority of thread 3 was changed and its QOS factor was re-registered. Because of this re-registration, the QOS-Control Server recalculated the CPU-resource assignment and set it to the CPU reserves. As a result of the recalculation, the period and computation time of the three dummy threads were modified.

[4]RT-Mach MK94 is a version released in 1994 by our Keio-MMP project [1], based on CMU's MK83i with some extensions.

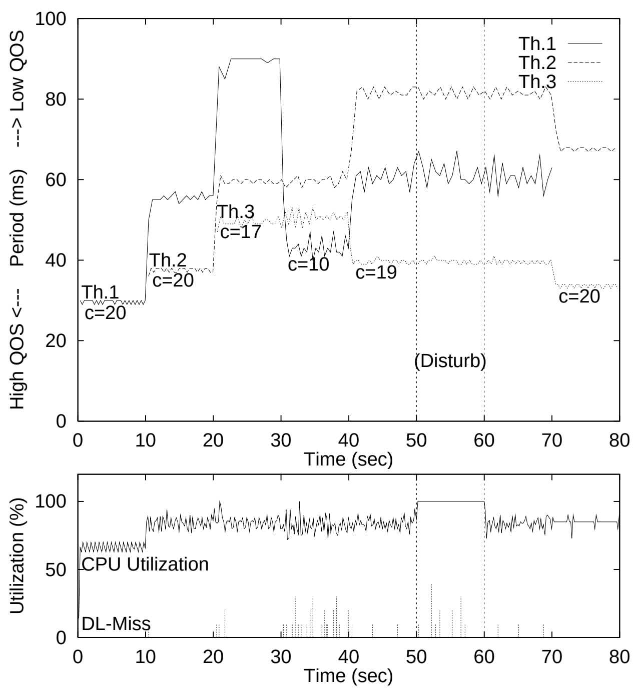


Figure 8: Result of dynamic QOS control with Q-Threads

Even during the 50 to 60 sec that the disturbing thread was running, the three dummy threads were able to run without being affected, in spite of the 100% CPU utilization shown in the lower graph. The behavior of the dummy threads was very stable, and very few deadline misses occurred, as the lower graph shows.[5]

In this experiment, most of the QOS adjustments and the necessary resource management were handled by the Q-Thread library (and the QOS-Control Server), and users were easily able to write QOS-controllable continuous-media sessions. We believe that the experiment verifies the basic efficiency of the Q-Thread execution model.

4.2 Experiment with a Real Application

As a "real" application, we are now developing a QuickTime player that uses the Q-Thread. This program processes and displays a corresponding frame of uncompressed QuickTime video data in the memory to X-Window, on every invocation. In the current version, only the frame rate (period) can be controlled. A fixed computation time (calibrated by using the previous actual computation time in the QOS-control hint) and the range of period calculated from the range of frame rate are specified as the Q-Thread attributes.

A second experiment is being carried out with this real application. Different ranges for the frame rate

[5]In this experiment, 52 deadline misses occurred in about 4000 invocations of the three dummy threads. These were caused by the consumption of more CPU resources than the available computation time allowed, because of some mis-evaluation of the workload.

Table 4: Attributes of the QuickTime players

No.	Start time (sec)	Frame rate (fps)	Priority
1	00 –	10 – 50	1 (low)
2	05 –	10 – 30	2 (medium)
3	10 –	10 – 100	3 (high)

and different priorities were specified for three QuickTime players, as shown in Table 4. Figure 9 shows the preliminary results obtained by using these players. The upper graph shows the real frame rate of each QuickTime player, and the lower graph shows the CPU utilization and deadline misses during the experiment.

5 Related Work

There have been several studies of execution models and processor scheduling for continuous-media processing. The RBE (Rate-Based Execution) model, proposed by researchers at the University of North Carolina [16], also targets deterministic execution and adaptive resource management for multimedia computing, and uses (x, y, d) parameters to represent the *execution rate* of a process. A group of researchers at Stanford University and Sun Microsystems have proposed a scheduling mechanism that tries to integrate continuous-media computations with conventional activities [17]. They also specify a minimum acceptable rate of execution for multimedia applications, and translate it into a series of deadlines at run time. Another rate-based scheduling mechanism, named RAP (Rate-based Adjustable Priority Scheduling), has been proposed by a group at the University of Kentucky [18].

We think that the period specification in the Q-Thread corresponds to the rate specification in these studies. But in addition, the tolerable ranges of the period and computation time can be specified in our Q-Thread model. These parameters enable the system to guarantee the CPU-resource allocation and to change the QOS dynamically. In addition, as an underlying resource-management mechanism for the Q-Thread, we have proposed the QOS-Ticket model [15]. Our primary contribution, we believe, has been to incorporate the QOS-control mechanism into the execution model in such a way as to combine resource reservation and adaptation.

6 Conclusions and Future Work

Because of the timing constraint in continuous-media processing, the QOS should be changed dynamically on the basis of available resources. In this paper, we have proposed a new process-execution model suitable for such dynamic QOS control, named "Q-Thread." The Q-Thread is an extension of the existing periodic RT-Thread incorporating a resource reservation and adaptation mechanism. A Q-Thread allows a user to specify tolerable ranges of the period and computation time for invocations, and makes it easy to write a continuous-media session that can control

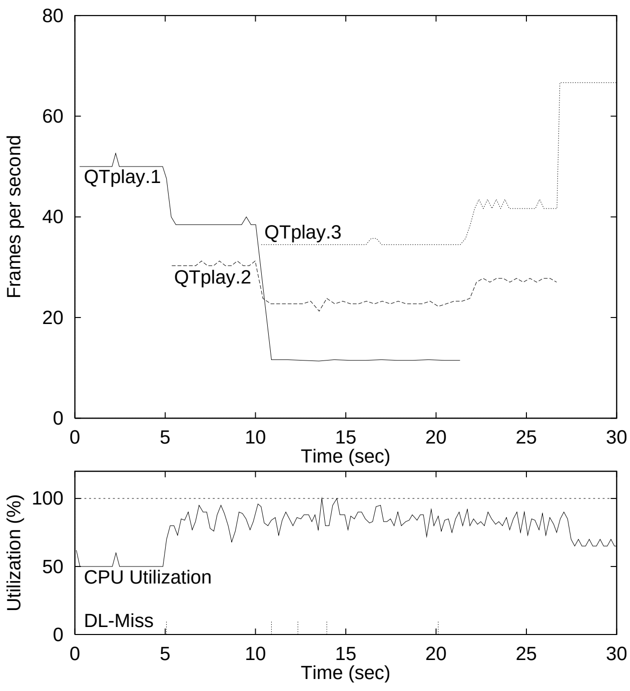


Figure 9: Result with QuickTime players using the Q-Thread

the QOS dynamically. We have also described some dynamic QOS-control experiments that show the effectiveness of the Q-Thread execution model.

Using the Q-Thread, we are currently developing real applications that can control its QOS dynamically. The QuickTime player mentioned in this paper is a prototype of such an application. In addition to extending the execution model itself, we think that the integration with other components (for example, the enhanced X-window system [19] and storage server [20]) are necessary for such real applications. A user-level real-time thread package is being developed as a part of our project [21]. We are also considering implementing the Q-Thread model by using this package.

Acknowledgements

The authors would like to thank members of the Keio-MMP project [1] and the Keio-IBM partnership program [22] for their help and advice.

References

[1] Keio-MMP Project: WWW Home Page, URL: <http://www.mmp.sfc.keio.ac.jp/>.

[2] H. Tokuda et al.: "Real-Time Mach: Towards a Predictable Real-Time System," *Proc. USENIX Mach Workshop*, pp. 73–82 (1990).

[3] K. Kawachiya et al.: "Extending Real-Time Mach for Continuous Media Applications," *Collected Abstracts 4th. Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 55–58 (1993).

[4] N. Nishio et al.: "Conductor-Performer: A Middle Ware Architecture for Continuous Media Applications," *Proc. 1st Intl. Workshop on Real-Time Computing Systems and Applications*, pp. 122–131 (1994).

[5] N. Nishio and H. Tokuda: "A Middle-Ware for Continuous Media Processing in the Keio-MMP Project," *Proc. Multimedia Japan '96*, pp. 278–284 (1996).

[6] J. F. Koegel Buford et al.: *Multimedia systems*, Chapter 8, ACM Press SIGGRAPH Series, Addison-Wesley (1994).

[7] H. Tokuda et al.: "A Real-Time Thread Model for Continuous Media Applications," *ART Group Tech. Report*, Carnegie Mellon University, May 1993 (1993).

[8] C. W. Mercer et al.: "Processor Capacity Reserves: An Abstraction for Managing Processor Usage," *Proc. 4th Workshop on Workstation Operating Systems*, pp. 129–134 (1993).

[9] C. W. Mercer et al.: "Processor Capacity Reserves: Operating System Support for Multimedia Applications," *Proc. 1st Intl. Conf. on Multimedia Computing and Systems*, pp. 90–99 (1994).

[10] H. Tokuda and T. Kitayama: "Dynamic QOS Control based on Real-Time Threads," *Proc. 4th Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 113–122 (1993).

[11] H. Tokuda et al.: "Continuous Media Communication with Dynamic QOS Control Using ARTS with an FDDI Network," *Proc. ACM SIGCOMM '92*, pp. 88–98 (1992).

[12] K. Kawachiya et al.: "QOS Control of Continuous Media: Architecture and System Support," *IBM Research Report, RT0108*, IBM (1995).

[13] K. Kawachiya et al.: "Evaluation of QOS-Control Servers on Real-Time Mach," *Proc. 5th Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 123–126 (1995).

[14] K. Kawachiya and H. Tokuda: "QOS-Ticket: A New Resource-Management Mechanism for Dynamic QOS Control of Multimedia," *Proc. Multimedia Japan '96*, pp. 14–21 (1996).

[15] K. Kawachiya and H. Tokuda: "Dynamic QOS Control Based on the QOS-Ticket Model," *3rd Intl. Conf. on Multimedia Computing and Systems*, to appear (1996).

[16] K. Jeffay and D. Bennett: "A Rate-Based Execution Abstraction for Multimedia Computing," *Proc. 5th Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 67–78 (1995).

[17] J. Nieh and M. S. Lam: "Integrated Processor Scheduling for Multimedia," *Proc. 5th Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 215–218 (1995).

[18] R. Yavatkar and K. Lakshman: "A CPU Scheduling Algorithm for Continuous Media Applications," *Proc. 5th Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 223–226 (1995).

[19] S. Tada: "Continuous Media Extension of X-Window System for Interactive Multimedia Applications," *Proc. Multimedia Japan '96*, pp. 285–292 (1996).

[20] H. Tezuka and T. Nakajima: "Design and Implementation of a Continuous Media Storage System on Real-Time Mach," *JAIST Research Report, IS-RR-94-15S*, Japan Advanced Institute of Science and Technology (1994).

[21] S. Oikawa and H. Tokuda: "User-Level Real-Time Threads," *Proc. 11th IEEE Workshop on Real-Time Operating Systems and Software*, pp. 7–11 (1994).

[22] Keio-IBM Partnership Program: WWW Home Page, URL: `<http://ibmpp.sfc.wide.ad.jp/>`.