Proceedings of the 1996 IEEE
International Conference on Robotics and Automation
Minneapolis, Minnesota - April 1996

A Rate-Monotonic Scheduler for the Real-time Control of Autonomous Robots

Robert George

Information Sciences Directorate
Army Research Laboratory
Adelphi, MD 20783
george@cs.nps.navy.mil

Yutaka Kanayama

Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943
kanayama@cs.nps.navy.mil

Abstract

Most existing real-time control systems use ad hoc static priority scheduling methods, in spite of the fact that the rate monotonic scheduling algorithm was proved to be the optimal static priority scheduling algorithm over 20 years ago. In this paper, we discuss a task library we are using for the real-time control of autonomous robots. The task library comprises a preemptive rate-monotonic scheduler which provides guaranteed optimal scheduling when certain conditions of processor utilization are met. The task library has been implemented as a collection of lightwight threads, which operate entirely in user-space for maximum efficiency. We show the performance advantages resulting from the reduced overhead of this approach, compared with commercial operating systems. The task system is robust, extensible, and portable, and has been successfully used to control the autonomous mobile robot Yamabico-11 developed at the Naval Postgraduate School.

1 Introduction

Although the developers of most robotic systems have an intuitive sense of what they mean by a real-time operating system, definitions vary widely. The distinction between real-time computer systems and general purpose computer systems lies not in their performance specifications, but in the relative importance of each system's timing considerations. In real-time applications, the correctness of a computation depends not only on the results of computation, but also the time at which outputs are generated. The real-time control of an autonomous robot provides an excellent example of a collection of related tasks which *must* complete execution by a well-specified deadline.

The measures of merit in a real-time system include:

- Predictably fast response to urgent events.
- High degree of schedulability. We define *schedulability* as the degree of resource utilization at or below which the timing requirements of all tasks can be guaranteed.
- Stability under transient load. When the system is overloaded by events and meeting all deadlines is impossible, we must still guarantee the deadlines of selected critical tasks.

A real-time operating system may satisfy its applications' computation deadlines implicitly, by hardware brute force, or by blind luck. Historically, most real-time operating system resource management is intended to be "real fast" as opposed to real-time. The execution time of the operating system services and internal operations are designed to be as fast as possible, to minimize the average execution times, and to have a relatively predictable upper bound for the worst case execution time. But these assumptions often are not explicitly identified, or even known. Such systems may successfully operate in real-time, and provide a cost-effective solution for certain applications. By classic definition, however, they are not real-time systems because they do not employ time-constraint driven resource management [1].

Traditionally, real-time systems use cyclical executives to schedule concurrent threads of execution. Under this approach, a programmer lays out an execution timeline by hand to serialize the execution of critical sections and to meet task deadlines.

While such an approach is managable for simple systems, it quickly becomes unmanagable for large systems. It is a painful process to develop application code so that it fits the time slots of a cyclical executive while ensuring that the critical sections of different tasks do not interleave. Forcing programmers to schedule tasks by fitting code segments on a time-line is no better than the outdated approach of managing memory by manual memory overlay.

Designers of real-time systems would like to have the same benefits that modern software engineers enjoy: they would like to be able to show early in the project that their designs meet all the requirements and are thus correct. With stringent timing considerations, this can be difficult.

2 Rate-Monotonic Scheduling

2.1 Overview

Rate-monotonic analysis, a collection of quantitative methods, provides a basis for designing, understanding, and analyzing the timing behavior of real-time computing systems. In essence, this theory ensures that as long as the CPU utilization of a task set lies below a certain bound and appropriate scheduling algorithms are used, all tasks will meet their deadlines without the programmer knowing exactly when any given task will be running. Even if a transient overload occurs, a fixed subset of critical tasks will still meet their deadlines as long as their CPU utilizations lie within appropriate bounds [4].

In short, rate-monotonic scheduling theory puts real-time software engineering on a sound analytical footing. Applying this theory to the control of autonomous robots, for example, allows us to separate concerns for the logical correctness of the tasks which comprise the robot's control system from the concerns of timing correctness.

2.2 Scheduling Periodic Tasks

The problem of scheduling periodic tasks was first addressed by Liu and Layland in 1973 [2]. The Liu and Layland analysis was derived under several assumptions:

- Tasks are periodic, are ready at the start of each period, have deadlines at the end of the period, and do not suspend themselves during execution.
- Tasks can be preempted, and the overhead for context swapping and task scheduling can be ignored.
- Tasks are independent, i.e., there is no task synchronization and tasks have known, deterministic worst-case execution times.

Now consider a set of n periodic tasks $\tau_1, ..., \tau_n$. Each task τ_i is characterized by four components $(C_i, T_i, D_i, I_i), 1 \le i \le n$ where

- C_i = deterministic computation requirement of each task of τ_i,
- T_i = period of τ_i,
- D_i = deadline of τ_i,
- I_i = phasing of τ_i relative to some fixed time origin.

Liu and Layland proved a set of n independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if

$$\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \le U(n), \tag{1}$$

where $U(n)$ is the *scheduling bound*, the maximum fraction of processor utilization allowable for n tasks:

$$U(n) = n(2^{\frac{1}{n}} - 1) \tag{2}$$

Finally, Liu and Layland showed that the rate monotonic scheduling algorithm is optimal among all fixed priority scheduling algorithms for scheduling periodic task sets with $D_i = T_i$.

While the mathematics for the proof may appear complex, the actual theory itself is quite simple. "Rate monotonic" scheduling implies that priorities should be assigned to tasks as a monotonically increasing function with respect to the task request rate $(1/T_i)$. In other words, the more often a task is run, the higher its priority should be. The rate monotonic theory ensures that as long as the processor's utilization is below a certain bound, all tasks in the task set will complete by their deadlines without the individual tasks requiring timing information about the other tasks.

Equations (1) and (2) offer a sufficient (worst-case) condition that characterizes schedulability of a task set under the rate monotonic algorithm. This bound converges to 69.3% (ln 2) as the number of tasks approaches infinity. This means that if we follow the rules of RMT and use no more than 69.3% of available processor cycles, we can guarantee optimal scheduling. The values of the scheduling bounds for one to nine independent tasks are as follows:

The worst-case bound of 69.3% processor utilization given in equations (1) and (2) are respectibly

Number of Tasks	Scheduling Bound
1	1.0
2	0.828
3	0.779
4	0.756
5	0.743
6	0.734
7	0.728
8	0.724
9	0.720
∞	0.693

Table 1: Processor utilization bounds

large. They are, in fact, quite pessimistic. Randomnly generated task sets are often schedulable by the rate monotonic algorithm at much higher utilization levels, even with worst-case phasing. In general, it has been shown that when periods are generated from a uniform distribution with a sufficiently wide range of values, the breakdown utilization will be in the 88% to 92% range.

In fact, in [4] Lehoczky, Sha, Strosnider, and Tokuda show that the rate monotonic scheduling algorithm can schedule task sets up to 100% utilization when $D_i = T_i$ and the tasks periods are harmonic:

If a task set $\tau_1, ..., \tau_n$ is scheduled using the rate monotonic algorithm and T_j evenly divides T_i for $1 \leq j \leq i$, then τ_i meets all its deadlines if and only if:

$$\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \quad \leq \quad 1 \tag{3}$$

□

In the next section, we will apply this important extension of rate-monotonic scheduling theory to generate task sets with processor utilization rates approaching unity.

2.3 Scheduling Robotic Control Tasks

The Yamabico-11 autonomous robot is programmed with a motion control libary called the *Model-Based Mobile Robot Language* (MML) [8]. In MML, motion planning is described in terms of an abstract two-dimensional coordinate system. This library provides a well-defined, clean interface to Yamabico users, where details of the low-level motion system are hidden from the user and are not required to describe motion. Sensor data is also available to the user in either a raw or processed format to be used in motion planning. The control of Yamabico is processed through the *user* task, which communicates with the motion control task, as well as others, through global shared memory.

The MML system is comprised of three primary tasks which may be running at any given time. Other user-defined tasks are added as needed. The highest priority task is the *motion control* task, which performs all low-level path calculations and direct motor control. The next highest priority task is the *sonar processor* task, which processes all incoming sonar returns and generates line segments representing obstacles in the local environment. The lowest priority task is the *user* task. This task computes high-level reasoning functions, and sends commands to the motion control subsystem through a command queue in shared memory [9].

To schedule these task with rate-montonic scheduling, we consider the case of three periodic tasks:

- Motion control task τ_1 : $C_1 = 3$ msec; $T_1 = 10$ msec; $U_1 = 0.3$
- Sonar control task τ_2 : $C_2 = 2$ msec; $T_2 = 30$ msec; $U_2 = 0.067$
- User function task τ_3 : $C_3 = 100$ msec; $T_3 = 300$ msec; $U_3 = 0.334$

We then calculate the processor utilization of this task set, according to Equations (1) and (2). The total utilization of these three tasks is 0.701, which is below Equation (2)'s bound for three tasks: $3(2^{\frac{1}{3}} - 1) = 0.779$. Thus, rate-monotic scheduling theory guarantees these tasks are schedulable – they will meet their deadlines if task τ_1 is given the highest priority, τ_2 the next highest, and τ_3 the lowest. The results of the rate-monotonic scheduling calculations are shown in Table 2, where $U_i = C_i/T_i$.

Task	T_i	C_i	$\sum_{i=1}^{n} U_i$	$U(n)$
τ_1	10	3	0.300	1.00
τ_2	30	2	0.367	0.828
τ_3	300	100	0.701	0.779

Table 2: Yamabico task utilization

Notice that we have scheduled the *user* task as if it were a periodic task. Although the *user* task will appear to the Yamabico programmer to run continuously, we have chosen to schedule the task with a long period. This insures that the *user* task will have the lowest rate-monotonic scheduling priority in the system. To ensure the maximum utilization of Yamabico's processor, we have also chosen the period of the *user* task to be a harmonic of the *motion control* and *sonar control* tasks. This allows the task set to approach 100% processor utilization, as defined by Equation (3).

Figure 2 provides a visual description of how Yamabico's tasks will be scheduled.

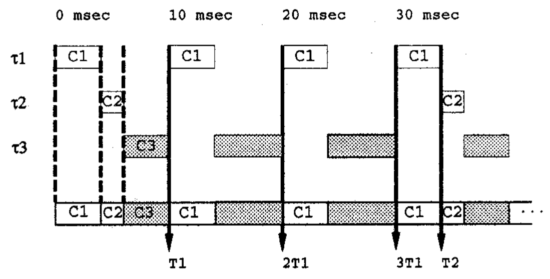

Figure 1: Yamabico's task Schedule.

2.4 Modifying the Schedule

Now let us consider the effects of changing the task set for Yamabico. We are developing a low-level control task called a *forerunner.* This task projects an image of Yamabico along the robot's current path of motion. An attempt in made to predict any collisions that may occur in the next moments of Yamabico motion plan. We have determined that the prototype of this task will require approximately 5 msec to compute on Yamabico's processor. We would like this task to have a priority less than the *motion control* task and less than, or equal to the *sonar control* task. We would also like the *forerunner* task to have a higher priority than the *user* task. Therefore, we choose a period (T_i) of 30 msecs, with a computation requirement (C_i) of 5 msec.

It should be noted that, with the addition of the fourth task, we have exceeded the rate-monotonic scheduling bound from Equation (2). However, since we have chosen the tasks to have periods which are harmonic with respect to each other, we utilize the scheduling bound in Equation (3). We summarize the re-computation of the rate-monotonic scheduling bounds in Table 3.

Figure 2 provides a visual description of how Yamabico's tasks will be scheduled after the *forerunner* task is added.

Task	T_i	C_i	$\sum_{i=1}^{n} U_i$	$U(n)$
τ_1	10	3	0.300	1.00
τ_2	30	2	0.367	0.828
τ_3	30	5	0.534	0.779
τ_4	300	100	0.867	0.756

Table 3: Yamabico task utilization

Thus, with a few calculations, we have successfully modified the schedule for Yamabico's task set. If we had been using a cyclical executive or an interrupt-driven system, we would have had difficulties meeting the responsiveness, schedulability, and stability requirements of our system. It would have also been necessary to modify timing dependent code in the pre-existing tasks to allow for the changes in the system's timing requirements due to the addition of the *forerunner* task.

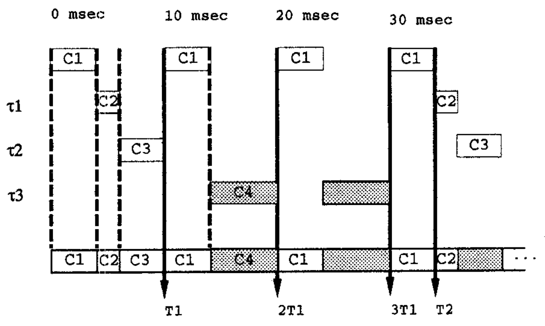

Figure 2: Yamabico's revised task Schedule.

3 Implementation

3.1 Yamabico-11 Hardware Description

Yamabico-11 is a wheeled, untethered autonomous mobile robot developed over the last seven years at the Naval Postgraduate School [6, 8]. The primary processing and control system is implemented on a 6U VMEbus system. In its current state, the system consists of the following components:

- An Ironics IV-SPRC SPARC microprocessor with 16 Mbyte of DRAM,
- A dual-axis motion controller and shaft deccoder,
- A custom sonar processing board,

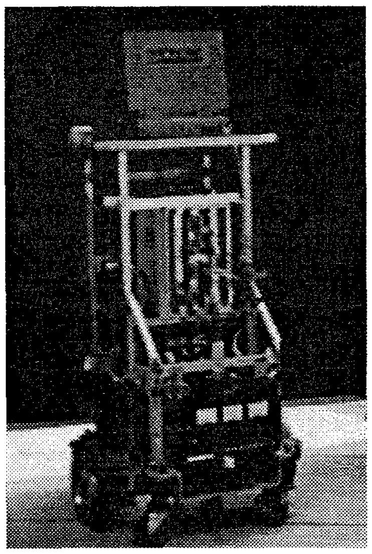

Figure 3: The Yamabico autonomous robot.

- An Imaging Technology, Inc. IMS color frame grabber,
- A serial communications board.

A lap-top computer is used as a real-time input/output device. Yamabico's size is $60(W) \times 60(L) \times 70(H)$ centimeters, and weighs approximately 60 kilograms (Fig. 3).

The kinematic architecture is a differential drive system used to control two drive wheels. Two 35 watt shaft-encoded DC motors drive 1/24 gear boxes. The sensor system consists of twelve 40KHz sonars mounted around the perimeter of the torso, and a CCD camera mounted atop the VME rack. The power system consists of two 12 volt gel-cell batteries.

Yamabico is programmed by first downloading object-code from a UNIX file-system. Next, The Sparc board is boot-strapped from a file server via the *bootp* protocol. The vehicle then operates as an untethered (self-contained) autonomous robot.

3.2 Multi-Threaded Implementaion of Rate-Monotonic Scheduling

We have implemented a rate-monotonic scheduler for Yamabico using lightweight processes (*threads*) executing in a single, partitioned address space. Conceptually, each of the threads of control can run independently and concurrently. Since they share a single address space, they can also share data.

Although the fashionable programming model has become multiple threads running in a common address space, this tends to make debugging difficult, and code reliability becomes an important issue. If the thread system can provide fast context switching, existing operating system services such as explicitly allocated shared memory between a team of cooperating processes can create a threaded environment, without opening the Pandora's box of problems that a fully shared memory space entails.

We have implemented *user-space* threads, which are managed entirely within the user address space. User threads are, in general, faster because they don't cross protection boundaries. When a user-space thread context-switches, the system must save and restore register values, including the stack pointer. Although this approach places the burden of many operating system services on the threads library, the performance enhancement from the reduction of overhead compared to a commerical operating system is considerable. Another convenient benefit of user-space threads is that all task creation and scheduling decisions are done in the user process, which provides a seamless means of integrating a rate-monotonic scheduler.

4 Experimental Results

The fundamental operations of the threads system are task creation and task switching. In order to make a meaningful evaluation of our scheduling system's performance, equivalent programs using task and UNIX Operating System processes were written. Each of the first pair of programs repeatedly creates new trivial tasks (threads) and waits for them to terminate. Each of the second pair of programs creates a group of eight children, and repeatedly passed control from one task (thread) to another. The programs were run on a SUN Sparc 4-490 under SunOS 4.1.3, and on a Silicon Graphics Iris running IRIX 5.2. The results were that task creation was 37 times faster with the threads library than with SunOS, and task switching was 10 times faster. The results are summarized in the following table:

It is important to note that the thread system and the UNIX Operating System are not equivalent, and that the results of these performance measurements do not imply that the threads system is 37 times better than UNIX. We merely intend to illustrate the performance gains available to the designers of robotic systems when the overhead associated with

Machine	UNIX Task Create	Thread Task Create
SPARC 4-490	1428.3	38.6
MIPS R3000	187.4	4.6

Table 4: Microseconds to task create

Machine	UNIX Task Switch	Thread Task Switch
SPARC 4-490	182.6	16.7
MIPS R3000	53.9	4.6

Table 5: Microseconds to task switch

commercial operating systems is not required.

5 Conclusion

In this paper, rate-monotonic theory was applied to the problem of scheduling real-time tasks comprising the control system of an autonomous robot. This approach allows the real-time software engineer to seperate the analysis of the logical correctness of the tasks comprising the control system from the timing correctness of the task set. The rate-monotonic scheduling approach greatly simplifies the modification of the tasks comprising the robot's control system, allowing the addition, modification, and deletion of tasks without great disturbance to the timing correctness of the real-time dealines. The authors have successfully implemented the multi-threaded rate-monotonic scheduling system on the autonomous mobile robot Yamabico-11 at the Naval Postgraduate School.

6 References

[1] D. L. Ripps *An Implementation Guide to Real-Time Programming*, Englewood Cliffs, NJ: Prentice Hall, 1989.

[2] C. L. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," Journal of the ACM, vol. 20. no. 1, 1973. pp 46-61.

[3] L. Sha and J. B. Goodenough, "Real-Time Scheduling Theory and Ada," IEEE Computer. vol. 23. no. 4, April 1990, pp. 53-62.

[4] John P. Lehoczky, Lui Sha, J.K. Strosnider and Hide Tokuda, "Fixed Priority Scheduling Theory for Hard Real-Time Systems," in Foundations of Real-time Computing, A. M. van Tilborg, Ed. Kluwer Academic Publishers, 1991, pp 6-8.

[5] Kanayama, Y., MacPherson, D.L., and Krahn, G.W., "Two Dimensional Transformations and Its Application to Vehicle Motion Control and Analysis," Proceedings of International Conference on Robotics and Automation, in Atlanta, Georgia, May 2-7, pp. 13-18, 1993.

[6] Y. Kanayama, Y. Kimura, F. Miyazaki and T. Noguchi, "A Stable Tracking Control Method for an Autonomous Mobile Robot," *Proc. IEEE Intntnl Conference on Robotics and Automation*, pp. 1315-1317, Cincinnati, Ohio, May 13-18, 1988.

[7] L. Sha, J. P. Lehoczky, and R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," IEEE Real-time Systems Symposium, IEEE-CS Press, 1986, pp. 181-191.

[8] Y. Kanayama and M. Onishi, "Locomotion Functions in the Mobile Robot Language, MML," IEEE Int. Conf on Robotics and Automation, (1991), pp. 1110-1115.

[9] Kanayama, Y., Kovalchik, J., Chuang, C., and Kelbe, F., "Motion Planning for Autonomous Mobile Robots," Proc. Autonomous Vehicles in Mine Countermesures Symposium, in Monterey, California, pp. 8-74 - 8-80, April 1995.