Global Rate-Monotonic Scheduling with Priority Promotion *

Shinpei Kato[†‡], Akira Takeda[‡], and Nobuyuki Yamasaki[‡]
[†]Department of Electrical and Computer Engineering, Carnegie Mellon University
[‡]Department of Information and Computer Science, Keio University

Abstract

In this paper, we consider a multicore real-time scheduling algorithm incorporating benefits of both fixed-priority and dynamic-priority disciplines. Specifically, the algorithm first assigns globally-effective priorities to real-time tasks statically, based on the well-known Rate-Monotonic scheduling policy. It may however change the task priorities at runtime, only when the tasks reach the zero-laxity condition, where no slack remains until the deadline, to avoid timing violations as much as possible. Implementation simplicity and response time predictability are therefore inherited from the fixed-priority discipline, while minimal dynamic-priorities are exploited, if necessary, to maintain the system to be schedulable as much as possible. We also provide a schedulability analysis and derive a schedulability test for the algorithm. Our evaluation then demonstrates that the algorithm outperforms the existing global fixed-priority scheduling algorithms in terms of schedulability.

1 Introduction

The advent of multicore technology has accelerated a better use of computing systems. Due to further successful development of low-power chips, multicore platforms are becoming to be used commonly even in embedded real-time systems, where energy constraints are usually imposed. Recently, the real-time systems community has therefore been interested in extending resource management schemes into the multicore context. In particular, much attention is being paid to studies on multicore real-time scheduling, given the natural need for concurrency management on multiple CPU cores.

Traditionally, one often discusses and compares fixed-priority and dynamic-priority scheduling algorithms. Fixed-priority algorithms, such Rate-Monotonic (RM) [14] and Deadline Monotonic (DM) [13], usually lead to simpler system implementation and better response time predictability. On the other hand, dynamic-priority algorithms, such as Earliest Deadline First (EDF) [14] and Least Laxity First (LLF) [16], likely achieve better system utilization under deadline constraints. Other issues to be compared between the the fixed-priority and the dynamic-priority approaches are reported in [6].

As for multicore extensions, one often discusses and compares global and partitioned scheduling schemes. In a global scheduling scheme, all tasks are conceptually stored in a global queue, and at any moment the m highest-priority tasks, if any, are scheduled on m CPU cores. Task migrations may therefore be exploited. In a partitioned scheduling scheme, on the other hand, each CPU core has its own task queue. Each task is then assigned to a specific CPU core, and is scheduled on the local CPU core without migrations. Indeed, there are relative merits in the two scheduling schemes. Global scheduling algorithms, such as Pfair [3], LLREF [9], and EDZL [12], are potentially able to maintain the system schedulable even with high-utilization task sets, whereas partitioned scheduling algorithms, such as EDF-FF [11] and EDF-FFD [15], allow us to ignore runtime overhead and complexity regarding task migrations. The summary of the two scheduling schemes is reported in [8].

This paper presents multicore global real-time scheduling with efficient priority assignments, where benefits of both fixed-priority and dynamic-priority disciplines are incorporated. Specifically, we propose a scheduling algorithm, called Rate-Monotonic until Zero Laxity (RMZL), which applies a laxity-driven priority promotion strategy adopted in the EDZL algorithm to the Rate-Monotonic algorithm. The primary advantage of the RMZL algorithm is that it provides any task sets schedulable under the Rate-Monotonic algorithm with implementation simplicity and response time predictability as the Rate-Monotonic algorithm does, while it maintains the system to be schedulable as much as the EDZL algorithm does, far beyond the Rate-Monotonic algorithm can do.

The rest of this paper is organized as follows. Section 2 describes our system model, including assumptions and terminologies used in this paper. Section 3 presents the RMZL algorithm and its properties, and Section 4 provides its schedulability analysis. The schedulability performance of the RMZL algorithm is evaluated through both simulation studies and practice experiments in Section 5. We summarize our concluding remarks in Section 6.

*This technical report is an English-version of the article, with slight modifications and extensions, published in the IPSJ Transactions on Advanced Computing System (ACS), Vol. 2, No. 1, pp. 64–74, March 2008.

2 System Model

The system is composed of m CPU cores. A task set including n tasks, denoted by $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$, is given to the system. Each task τ_i is represented by a tupple (C_i, T_i), where C_i and T_i are the *worst-case execution time (WCET)* and the *period* of τ_i respectively. Tasks in τ are assumed to be sorted in non-decreasing order of periods, i.e., $T_1 \leq T_2 \leq ... \leq T_n$ holds. The *utilization* U_i of τ_i is defined to be $U_i = C_i/T_i$, and the total utilization $U(\tau)$ of the tasks in τ is defined to be $U(\tau) = \sum_{\tau_i \in \tau} U_i$. In particular, we use a word "*system utilization*" for $U(\tau)/m$, which indicates the total utilization of the tasks normalized by the number of CPU cores.

Each task τ_i produces an infinite sequence of jobs periodically. The kth job of τ_i is denoted by $\tau_{i,k}$. The *release time* and the *deadline* of $\tau_{i,k}$ are denoted by $r_{i,k}$ and $d_{i,k}$ respectively. Note that $d_{i,k} = r_{i,k+1} = r_{i,k} + T_k$ holds for any $\tau_{i,k}$. We also denote the *remaining execution time* of $\tau_{i,k}$ at time t by $C_{i,k}(t)$. The *laxity* $L_{i,k}(t)$ of $\tau_{i,k}$ at time t is then defined to be $L_{i,k}(t) = d_{i,k-(t+C_{i,k}(t)}$. The laxity of a job directly reflects the degree of urgency to meet its deadline. The less the laxity of a job is, the more the job is urgent. In particular, when the laxity of a job becomes zero, the job is said to be in the *zero-laxity* condition. The absolute value of the laxity is also called the *tardiness*, when the laxity becomes negative. The tardiness indicates the amount of time behind the deadline.

When all CPU cores are occupied by jobs with priorities higher than a ready job $\tau_{i,k}$, $\tau_{i,k}$ is said to be *blocked*. All tasks are preemptive and independent of each other. No more than one CPU core executes the same job at the same time. Once the system begins running, no tasks join and leave there dynamically.

A task set to be schedulable under a scheduling algorithm means that all jobs in the task set are guaranteed to scheduled by the scheduling algorithm, without any deadline misses. The schedulability test is a function that verifies whether or not the given task set is schedulable under the given scheduling algorithm. Any scheduling algorithms used in hard real-time systems must contain explicit schedulability test functions. In soft real-time systems, meanwhile, explicit schedulability test functions may not be necessary, though the tardiness of each task is desired to be bounded in advance so that quality of service is guaranteed at a certain level. It is also preferable for the scheduling algorithm to be *work-conserving* when response times are important in the system. Here, the scheduling algorithm is said to be work-conserving, if it has a property to ensure that the system does not become idle in the presence of ready jobs.

3 The RMZL Scheduling Algorithm

We now present the RMZL algorithm. The RMZL algorithm applies the laxity-driven priority promotion strategy adopted in the EDZL algorithm to the Rate-Monotonic algorithm. Specifically, it schedules jobs according to the Rate-Monotonic algorithm until some jobs reach the zero-laxity condition, and the priorities of these zero-laxity jobs are immediately promoted to the top to avoid timing violations. Figure 1 illustrates the pseudo-code of the RMZL algorithm. Every time some job is released, it is first assigned the Rate-Monotonic priority (line2-4). However, if the laxity of some job becomes zero, priority promotion is exploited (line 5-7). At any moment, jobs are scheduled according to their priorities (line 8-12). It should be noted that it depends on system implementation how to deal with the case where more than m jobs have the zero-laxity condition. In such a case, some job inevitably has negative laxity, which means that the job would miss its deadline if its execution time is equal to the WCET. If we consider hard real-time systems, this case should never happen. If we consider soft real-time systems, on the other hand, this case may happen, and our solution is to keep jobs with negative laxity being assigned the highest priority so that they can complete as early as possible.

```
1.  while TRUE do
2.    if some job τ_{i,j} is released then
3.      assign the Rate-Monotonic priority to τ_{i,j};
4.    end if
5.    if the laxity of some job τ_{i,k} becomes zero then
6.      assign the highest priority to τ_{i,j};
7.    end if
8.    if there are more than m jobs in the ready queue then
9.      schedule the m highest-priority jobs;
10.   else
11.     schedule all the jobs in the ready queue;
12.   end if
13. end while
```

Figure 1. The RMZL algorithm.

Figure 2 shows a simple example that demonstrates how the RMZL algorithm outperforms the Rate-Monotonic algorithm on two CPU cores, regarding task set $\tau = \{\tau_1 = (2, 3), \tau_2 = (2, 3), \tau_3 = (2, 3)\}$. For simplicity, let us assume that ties are broken in favor of lower-indexed tasks. Since τ_1 and τ_2 are scheduled first, τ_3 misses a deadline under the Rate-Monotonic algorithm. However, τ_3 can preempt τ_2 due to the zero-laxity condition under the RMZL algorithm, and as a result, all the tasks successfully meet deadlines.

Work-conserving property: The RMZL algorithm is work-conserving as the Rate-Monotonic algorithm is. Specifically, the system never becomes idle as long as there are ready jobs. The average response time of the system is therefore maximized. This property improves the average response time in particular when the system load is low, as compared to such algorithms that are not work-conserving.

Domination property: The RMZL algorithm strictly dominates the Rate-Monotonic algorithm under the worst-case assumption, given the fact that the RMZL algorithm changes task priorities only when some jobs are verified to

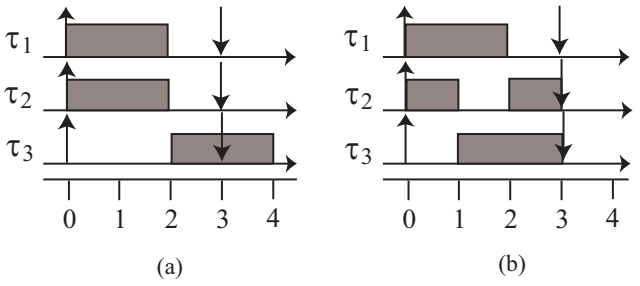

Figure 2. Example: (a) Rate-Monotonic scheduling and (b) RMZL scheduling.

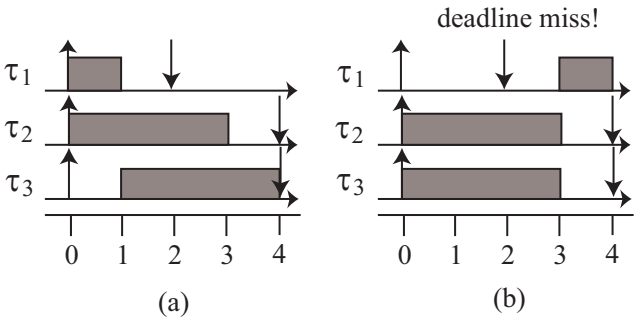

Figure 3. Example: (a) Rate-Monotonic scheduling and (b) RM-US scheduling.

reach the zero-laxity condition. In other words, any task sets that are schedulable under the Rate-Monotonic algorithm are also schedulable under the RMZL algorithm, if all the execution times are equal to the WCETs. Hence, the RMZL algorithm inherits all the properties of the Rate-Monotonic algorithm, such as implementation simplicity and response time predictability. Meanwhile, another global scheduling algorithm derived from the Rate-Monotonic algorithm, called RM-US [1], does not dominate the Rate-Monotonic algorithm. The RM-US algorithm statically assigns the highest priority to such tasks that have utilization greater than $m/(3m-2)$. Figure 3 shows a simple example where task set $\tau = \{\tau_1 = (1,2), \tau_2 = (3,4), \tau_3 = (3,4)\}$ is schedulable under the Rate-Monotonic algorithm, while it is not under the RM-US algorithm. As one can see, all the tasks are successfully scheduled by the Rate-Monotonic algorithm. However, τ_1 misses a deadline under the RM-US algorithm, since τ_2 and τ_3 are scheduled in priority due to their utilization.

Comparison with EDZL: We now compare the RMZL algorithm with the EDZL algorithm. Both the two algorithms use the same priority promotion strategy. However, the RMZL algorithm is more predictable than the EDZL algorithm in terms of response times, since it is based on the fixed-priority approach. The RMZL algorithm is also expected to accept more task sets than the EDZL algorithm by schedulability test, given that the Rate-Monotonic algorithm is able to accept more task sets than the EDF algorithm by schedulability test [4]. We show this effect by simulation studies in Section 5.

4 Schedulability Analysis

In this section, we present the RMZL schedulability analysis. Our approach is based on the response time analysis (RTA) for globally-scheduled systems [4] and the EDZL schedulability analysis [10]. Before providing our analysis, we define the following terms and terminologies.

Definition 1 (Interference)**.** *Interference $I_k(a,b)$ to task τ_k in interval $[a,b)$ is a cumulative length of intervals in $[a,b)$, for which τ_k is blocked by higher-priority tasks and cannot execute. The contribution of each individual task τ_i to $I_k(a,b)$ is then denoted by $I_k^i(a,b)$.*

Definition 2 (Work)**.** *Work $W_k(a,b)$ for task τ_k in interval $[a,b)$ is the total amount of time that must be consumed by τ_k in $[a,b)$ under the given timing constraints.*

Regarding the interference, we provide Lemma 1.

Lemma 1. *All global scheduling algorithms hold Condition* (1)*, where x is any positive value.*

$$I_k(a,b) \geq x \Leftrightarrow \sum_{i \neq k} min\left(I_k^i(a,b), x\right) \geq mx \tag{1}$$

Proof. The proof is subject to the EDF schedulability analysis provided by Lemma 4 in [5]. □

Our analysis proceeds as follows. Let J_k^* be a job of each task τ_k, which executes with the worst-case response time. Let also R_k^{ub} be the upper bound on the response time of τ_k. We first obtain the upper bound I_k^{ub} on the interference to J_k^* in interval $[r_k^*, r_k^* + R_k^{ub}]$. We then derive R_k^{ub} based on I_k^{ub} and C_k. Note that the lower bound L_k^{lb} on the laxity of a job of τ_k is computed by $R_k^{ub} - T_k$. According to the RMZL algorithm, if more than m jobs reach the zero-laxity condition at the same time, Hence, the necessary condition for the RMZL algorithm to cause deadlines to be missed is that $L_k^{lb} \leq 0$ is true for $m+1$ tasks and one of them strictly holds $L_k^{lb} < 0$. In fact, this is the same condition for the EDZL to cause deadlines to be missed, as presented in [10].

Now, we obtain I_k^{ub} as follows.

Lemma 2. *The contribution $I_k^i(a,b)$ of τ_i to the interference to τ_k in interval $[a,b)$ does not exceed work $W_i(a,b)$ of τ_i in $[a,b)$.*

Proof. A task interfere with another only when it is executed. It is therefore trivial from the definitions of the interference and the work. □

Lemma 3. *Work $W_i^{ub}(r_k^*, r_k^* + R_k^{ub})$ of task τ_i in interval $[r_k^*, r_k^* + R_k^{ub})$, which interferes with task τ_k is computed by Equation* (2)*, when task set $\tau = \{\tau_1, ..., \tau_n\}$ is scheduled by the* RMZL *algorithm on m CPU cores, where $n_i(R_k^{ub})$ is obtained by Equation* (3)*.*

$$W_i^{ub}(r_k^*, r_k^* + R_k^{ub}) = W_i^{ub}(R_k^{ub}) = \begin{cases} n_i(R_k^{ub})C_i \\ + \min(C_i, R_k^{ub} + T_i - C_i - n_i(R_k^{ub})T_i) & (i < k) \\ C_i & (i > k) \end{cases} \quad (2)$$

$$n_i(R_k^{ub}) = \left\lfloor \frac{R_k^{ub} + T_i - C_i}{T_i} \right\rfloor \quad (3)$$

Proof. We first consider task τ_i that has a longer period than task τ_k, i.e., $i > k$. According to the RMZL algorithm, in order for τ_i to interfere with τ_k, it is necessary for τ_i to have zero-laxity, since τ_i is assigned a lower priority when its laxity is positive. The work of τ_i is therefore maximized when each job of τ_i executes for C_i time units at the very end of the period. As seen in Figure 4, the work of τ_i, indicated by the shaded area in Figure 4, is no greater than C_i. Hence, Work $W_i^{ub}(r_k^*, r_k^* + R_k^{ub})$ must satisfy Condition (4).

$$W_i(r_k^*, r_k^* + R_k^{ub}) \leq C_i \quad (4)$$

We next consider task τ_i that has a shorter period than τ_k, i.e., $i < k$. It is clear from the prior discussion that the work of a job of τ_i which has a release time before and a deadline within the interval under consideration is maximized when the job executes for C_i time units at the very end of the period. The work of a job of τ_i which has a release time within and a deadline after the interval under consideration is, on the other hand, maximized when the job executes for C_i time units at the very beginning of the period. Now, we claim that the work of τ_i is maximized when the job of τ_i which has a release time before and a deadline within the interval under consideration is released at time r_k^*; namely when J_k^* is released at the same time, as shown in Figure 5. Figure 5 implies that the number $n_i(R_k^{ub})$ of jobs which can execute for C_i time units completely in $[r_k^*, r_k^* + R_k^{ub})$ is given by Equation (3).

The work of a job executing at the end of the interval under consideration, indicated by the shaded area in Figure 5 is bounded by $\min(C_i, R_k^{ub} + T_i - C_i - n_i(R_k^{ub})T_i)$. Hence, Condition (5) must hold.

$$W_i(r_k^*, r_k^* + R_k^{ub}) \leq n_i(R_k^{ub})C_i + \min(C_i, R_k^{ub} + T_i - C_i - n_i(R_k^{ub})T_i) \quad (5)$$

The lemma is thus true. □

The upper bound on the contribution of each task τ_i to the interference to task τ_k in interval $[r_k^*, r_k^* + R_k^{ub})$ is now known from Lemma 3. According to the definition of the interference, however, the interference to τ_k is no greater than $R_k^{ub} - C_k$. Otherwise, the response time of τ_k would exceed R_k^{ub}. This observation leads to Lemma 4.

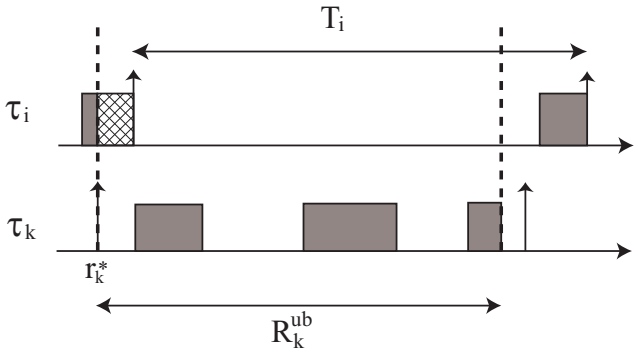

Figure 4. Case $i > k$

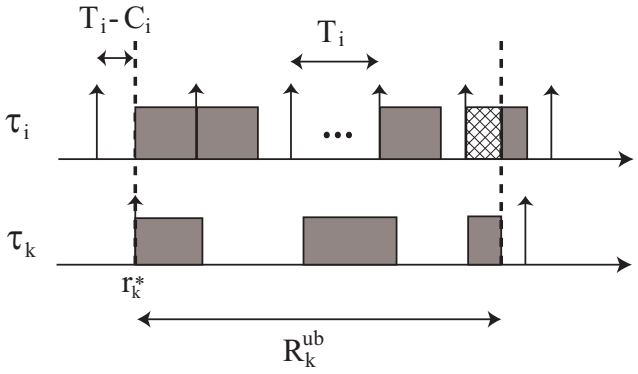

Figure 5. Case $i < k$

Lemma 4. *The response time of task does not exceed R_k^{ub}, if Condition* (6) *is satisfied.*

$$\sum_{i \neq k} min(I_k^i(R_k^{ub}), R_k^{ub} - C_k + 1) < m(R_k^{ub} - C_k + 1) \quad (6)$$

Proof. By Lemma 1, the following condition holds if Condition (6) holds.

$$I_k(R_k^{ub}) < (R_k^{ub} - C_k + 1)$$

Hence, the interference to J_k^* is no greater than $R_k^{ub} - C_k$, and J_k^* can complete by R_k^{ub} by the definition of the interference. □

Lemma 4 implies that the contribution of each task τ_i to the interference to task τ_k is bounded by $min(W_i^{ub}, R_k^{ub} - C_k + 1)$. The above discussion leads to the upper bound on the response time of each task under the RMZL algorithm as follows.

Theorem 1 (RTA for RMZL)**.** *The upper bound on the response time of task τ_k under the* RMZL *algorithm is obtained by solving fixed-point iteration for Expression* (7) *beginning with $R_k^{ub} = C_k$, where $\hat{I}_k^i(R_k^{ub})$ is computed by Equation* (8) *and $W_i^{ub}(R_k^{ub})$ is given by Equation* (2)*.*

$$R_k^{ub} \leftarrow C_k + \left\lfloor \frac{1}{m} \sum_{i \neq k} \hat{I}_k^i(R_k^{ub}) \right\rfloor \tag{7}$$

$$\hat{I}_k^i(R_k^{ub}) = min(W_i^{ub}(R_k^{ub}), R_k^{ub} - C_k + 1) \tag{8}$$

Proof. We provide proof by contradiction. Specifically, we assume that the convergent value of R_k^{ub} is greater than the response time of τ_k. The convergent value of R_k^{ub} is computed by the following expression.

$$R_k^{ub} = C_k + \left\lfloor \frac{1}{m} \sum_{i \neq k} min(W_i^{ub}(R_k^{ub}), R_k^{ub} - C_k + 1) \right\rfloor$$

By Lemma 2, $W_i^{ub}(R_k^{ub}) \geq I_k^i(r_k^*, r_k^* + R_k^{ub})$ holds. Therefore, the following condition must hold.

$$R_k^{ub} \geq C_k + \left\lfloor \frac{1}{m} \sum_{i \neq k} min(I_k^i(R_k^{ub}), R_k^{ub} - C_k + 1) \right\rfloor$$

By our assumption and Lemma 4, the following condition must also hold.

$$R_k^{ub} \geq C_k + \left\lfloor \frac{1}{m} m(R_k^{ub} - C_k + 1) \right\rfloor = R_k^{ub} + 1$$

This is a contradiction. The theorem is thus true. □

As discussed before, the necessary condition for the RMZL algorithm to cause deadlines to be missed is that more than m tasks satisfy $L_k^{lb} \leq 0$ and one of them strictly holds $L_k^{lb} < 0$. Finally, the schedulability test for the RMZL algorithm is derived as follows.

Theorem 2 (RMZL test)**.** *Task set $\tau = \{\tau_1, ..., \tau_n\}$ is schedulable under the RMZL algorithm on m CPU cores, unless at least $m + 1$ tasks satisfy Condition* (9) *and one of them strictly holds $<$ in Condition* (9)*, where R_k^{ub} is given by Theorem 1.*

$$L_k^{lb} = T_k - R_k^{ub} \leq 0 \tag{9}$$

The schedulability test derived in Theorem 2 can be improved. If the response time of τ_i which interfere with task τ_k is known, the contribution of τ_i to the interference to τ_k can be underestimated by taking into account the lower bound on the laxity of τ_i. Figure 6 depicts the case where L_i^{lb} is taken into account. Now, Equation (2) and (3) can be rewritten as follows.

$$W_i^{ub}(r_k^*, r_k^* + R_k^{ub}) = W_i^{ub}(R_k^{ub})$$
$$= \begin{cases} n_i(R_k^{ub})C_i + \min(C_i, \\ R_k^{ub} + T_i - C_i - L_i^{lb} - n_i(R_k^{ub})T_i) & (i < k) \\ C_i & (i > k) \end{cases} \tag{10}$$

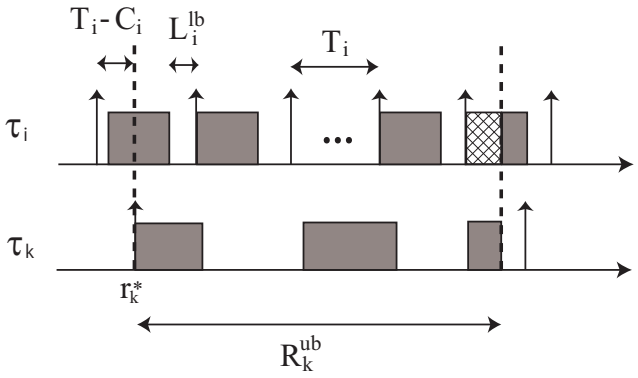

Figure 6. Case $i < k$ in the improved analysis.

$$n_i(R_k^{ub}) = \left\lfloor \frac{R_k^{ub} + T_i - C_i - L_i^{lb}}{T_i} \right\rfloor \tag{11}$$

The refined upper bound on the response time of τ_k is given by Theorem 3.

Theorem 3 (Refined RTA for RMZL)**.** *The upper bound on the response time of task τ_k under the* RMZL *algorithm is obtained by solving fixed-point iteration for Expression* (7) *beginning with $R_k^{ub} = C_k$, where $\hat{I}_k^i(R_k^{ub})$ is computed by Equation* (8) *and $W_i^{ub}(R_k^{ub})$ is given by Equation* (10)*.*

Theorem 4 (Refined RMZL test)**.** . *Task set $\tau = \{\tau_1, ..., \tau_n\}$ is schedulable under the RMZL algorithm on m CPU cores, unless at least $m + 1$ tasks satisfy Condition* (9) *and one of them strictly holds $<$ in Condition* (9)*, where R_k^{ub} is given by Theorem 3.*

Tardiness Bound: In soft real-time systems, or in any system where soft real-time tasks exist, the tardiness from the deadline is desired to be bounded for each soft real-time task. Under the RMZL algorithm, the tardiness bound is easily derived by using the upper bound on the response time.

Theorem 5 (Tardiness bound for RMZL)**.** *The tardiness bound for task τ_k executing in the system scheduled by the RMZL algorithm on m CPU cores is obtained by Equation* (12)*, where R_k^{ub} is given by Theorem 3.*

$$tardiness(\tau_k) = R_k^{ub} - T_k \tag{12}$$

5 Evaluation

We now provide a quantitative evaluation of the RMZL algorithm. Our performance metric is the *success ratio* defined by the following formula, which indicates the ability of a scheduling algorithm to successfully schedule given task sets.

$$\text{Success Ratio} = \frac{\text{\# of successfully scheduled task sets}}{\text{\# of scheduled task sets}}$$

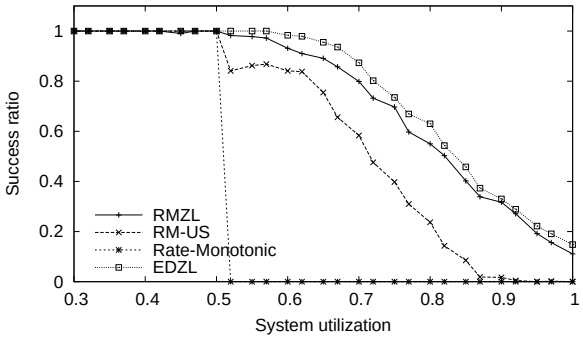

Figure 7. Simulation results for schedulability test on 2 CPU cores.

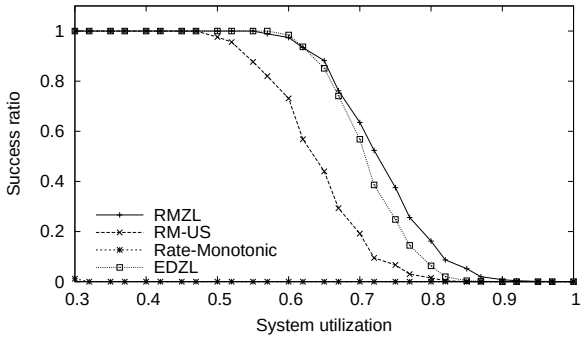

Figure 9. Simulation results for schedulability test on 8 CPU cores.

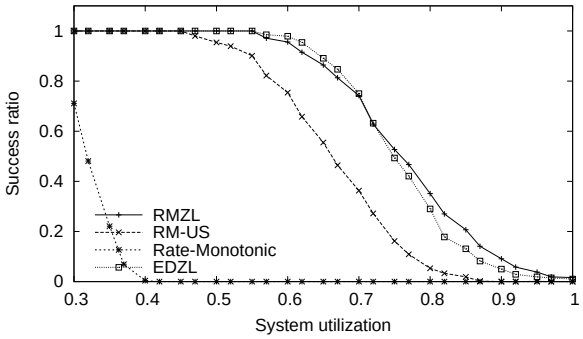

Figure 8. Simulation results for schedulability test on 4 CPU cores.

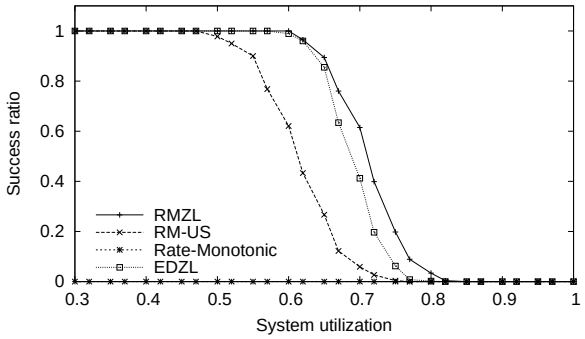

Figure 10. Simulation results for schedulability test on 16 CPU cores.

In our evaluation, we compare the RMZL algorithm with the two well-known fixed-priority scheduling algorithms: Rate-Monotonic and RM-US. We also in part compare it with the EDZL algorithm that adopts the same priority promotion strategy as the RMZL algorithm under a dynamic-priority discipline.

5.1 Simulation Studies

We first study the performance of the RMZL algorithm through simulations. Each simulation generates a random task set including 1000 tasks. The system load to be produced by the task set is determined by two parameters: the number m of CPU cores and the system utilization U_{sys}. Due to space constraints, we show only the results with limited characteristics of task sets as follows. For each task set τ, the utilizations of the tasks are uniformly set within range $[0.01, 1.0]$ so that $U_{sys} = \sum_{\tau_i \in \tau} U_i/m$ is satisfied. The task periods are also uniformly set within range $[100, 3000]$, and the WCET for each task τ_i is computed by $C_i = U_i T_i$.

Figure 7, 8, 9, and 10 show the simulation results for the schedulability test ability of each algorithm on 2, 4, 8, and 16 CPU cores respectively. Here, task sets are said to be successfully scheduled if they are verified to be schedulable by the schedulability test. For simplicity, the schedulability tests presented in [2] are used for the Rate-Monotonic and the RM-US algorithms. Those for the RMZL and the EDZL algorithms are respectively provided by Theorem 4 in this paper and Theorem 3 in [10]. These results demonstrate that the RMZL algorithm outperforms the traditional fixed-priority scheduling algorithms, i.e., Rate-Monotonic and RM-US in terms of hard real-time schedulability test guarantees. Since the current state of the art in global scheduling analysis is still pessimistic in particular when a large number of CPU cores is given, the absolute success ratios decrease as the number of CPU cores increase. Especially, the Rate-Monotonic algorithm suffer from a larger number of CPU cores due to the well-known Dhall's effect [11]. One can also observe that it is competitive with and even better for a large number of CPU cores than the EDZL algorithm. Such superiority comes from the fact that the global scheduling analysis is less pessimistic for fixed-priority algorithms than dynamic-priority algorithms [4].

Figure 11, 12, 13, and 14 show the simulation results for the runtime scheduling ability of each algorithm on 2, 4, 8, and 16 CPU cores respectively. Here, task sets are

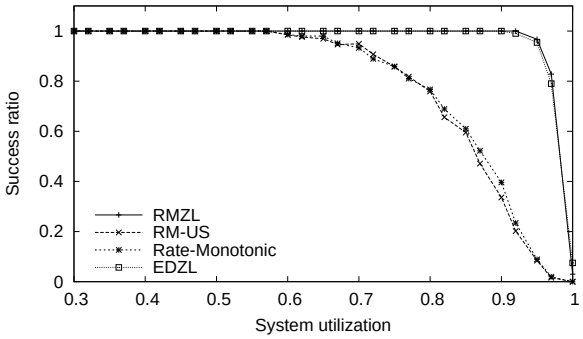

Figure 11. Simulation results for runtime scheduling on 2 CPU cores.

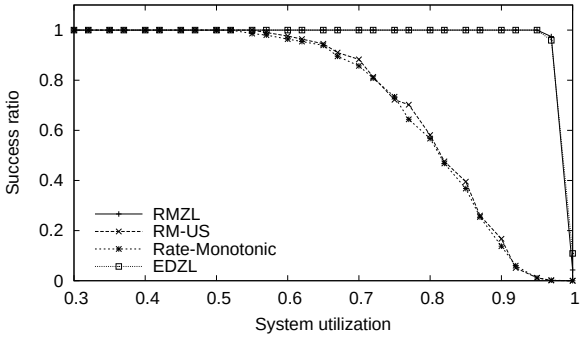

Figure 12. Simulation results for runtime scheduling on 4 CPU cores.

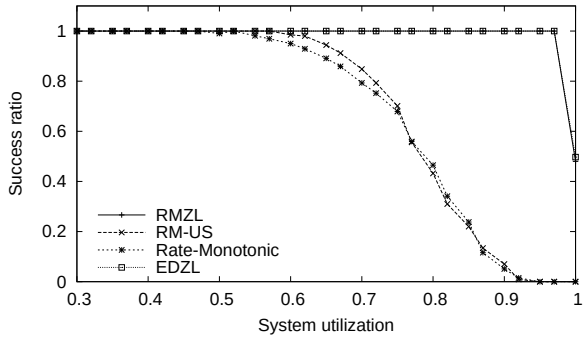

Figure 13. Simulation results for runtime scheduling on 8 CPU cores.

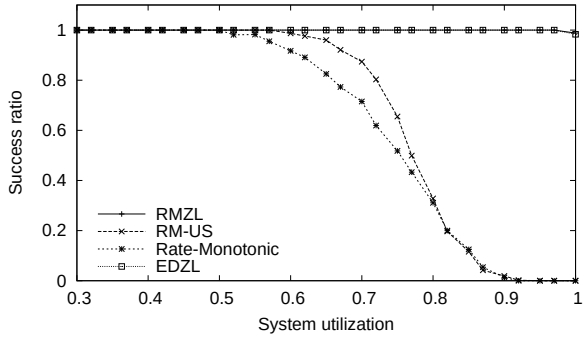

Figure 14. Simulation results for runtime scheduling on 16 CPU cores.

said to be successfully scheduled if they are actually scheduled without any deadline misses in 1000000 simulation time units. As compared to the schedulability test results, all the algorithms achieve much higher success ratios. This means that there are still huge gaps between analysis and practice, which should be mitigated in the future. One can remark that the RMZL algorithm is competitive with the EDZL algorithm even for the runtime scheduling performance. Given that the RMZL algorithm still inherits the nice properties of fixed-priority algorithms, such as implementation simplicity and response time predictability, we believe that it is a good choice as a scheduling algorithm for future multicore systems.

5.2 Practice Experiments

We next evaluate the practice performance of the RMZL algorithm, using the LITMUS^{RT} operating system [7] and the Intel Core 2 Duo and Quad processors operating at 1.83GHz and 2.0GHz respectively. All the scheduling algorithms tested are implemented as the LITMUS^{RT} scheduler plugins. To detect the zero-laxity condition for the RMZL algorithm, the laxity check procedure is added to the scheduler tick function. In our experiments, busy-loop tasks consuming the assigned WCETs are used to measure the runtime scheduling performance. The utilizations, the periods, and the WCETs of the tasks are determined by the same method as the one presented in Section 5.1. The number of tasks in each task set is however reduced to 100 and the test duration of each experiment is set at 30 seconds in consideration of the total time consumption.

Figure 15 and 16 show the experimental results for the runtime scheduling ability of each scheduling algorithm on the Intel Core 2 Duo and Quad processors. Like simulation studies, the RMZL algorithm outperforms the Rate-Monotonic and the RM-US algorithms. The performance difference between the RMZL algorithm and the other algorithms is however decreased as compared to the simulation cases. The performance decrease for the RMZL algorithm is mainly caused by the runtime overhead. While the Rate-Monotonic and the RM-US algorithms simply assign static priorities to the tasks, the RMZL algorithm needs to check the laxity of the current task at every scheduler tick and change its priority if necessary. However, we still claim that non-trivial performance improvements are achieved by the RMZL algorithm in practice.

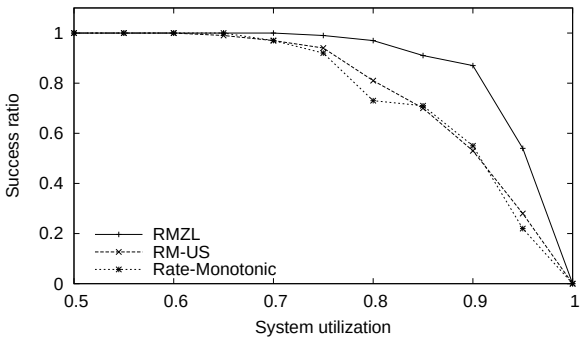

Figure 15. Experimental results for runtime scheduling on the Core 2 Duo processor.

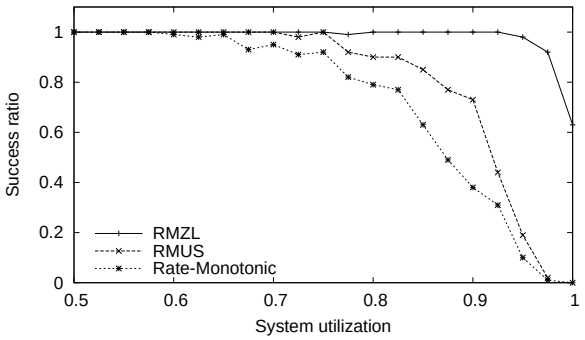

Figure 16. Experimental results for runtime scheduling on the Core 2 Quad processor.

6 Conclusions

In this paper, we have presented the Rate-Monotonic until Zero Laxity (RMZL) algorithm, which applies the laxity-driven priority promotion strategy to the global Rate-Monotonic algorithm. We also have derived the schedulability test and the tardiness bound for the RMZL algorithm, based on the prior schedulability analytical results. According to our evaluation, the RMZL algorithm is able to accept more task sets than the Rate-Monotonic and the RM-US algorithms by schedulability test. In fact, the RMZL algorithm is competitive with and is even better for a larger number of CPU cores than the EDZL algorithm. We also have shown that the RMZL algorithm in practice outperforms the Rate-Monotonic and the RM-US algorithms on the real-world machines.

In future work, we will consider more strict schedulability analysis for the RMZL algorithm, since our evaluation shows that the presented schedulability test is pessimistic, while it outperforms the existing fixed-priority algorithms. We are also interested in deriving the upper bound on the system utilization where any task sets are schedulable under the RMZL algorithm, given that scheduling algorithms are often compared in terms of this schedulable system utilization. In addition, implementation issues, such as how to detect the zero-laxity condition, are still left open for future work.

References

[1] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on mutiprocessors. In *Proceedings of IEEE Real-Time Systems Symposium*, pp. 193–202, 2001.

[2] T. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 120–129, 2003.

[3] S. Baruah, J. Gehrke, and C. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the International Parallel Processing Symposium*, pp. 280–288, 1995.

[4] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proceedings of the IEEE International Real-Time System Symposium*, pp. 149–158, 2007.

[5] M. Bertogna, M. Cirinei, and G. Lipari. Improved Schedulability Analysis of EDF on Multiprocessor Platforms. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pp. 209–218, 2005.

[6] G. Buttazzo. Rate monotonic vs. EDF: Judgement day. *Real-Time Systems*, 29(1):5–26, 2005.

[7] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 111–123, 2006.

[8] J. Carpenter, S. Funk, P. Holman, J. Anderson, and S. Baruah. *A categorization of real-time multiprocessor scheduling problems and algorithms*, chapter 30. CHAPMAN & HALL/CRC, 2004.

[9] H. Cho, B. Ravindran, and E. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 101–110, 2006.

[10] M. Cirinei and T. P. Baker. EDZL Scheduling Analysis. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pp. 9–18, 2007.

[11] S. Dhall and C. Liu. On a real-time scheduling proglem. *Operations Research*, 26:127–140, 1978.

[12] S. Lee. On-line multiprocessor scheduling algorithms for real-time tasks. In *Proceedings of the IEEE Region 10's Annual International Conference*, pp. 607–611, 1994.

[13] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation, Elservier Science*, 22:237–250, 1982.

[14] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[15] J. Lopez, M. Garcia, J. Diaz, and D. Garcia. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28:39–68, 2004.

[16] A. Mok and M. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proceedings of the Texas Conference on Computer System*, 1978.