Abstract—Semi-partitioned scheduling algorithms attempt to utilize the spare capacity in the partitioned approaches by splitting a number of tasks between processors. The main challenge in these approaches is how to split tasks and assign each partition to a different processor to achieve the highest system utilization while the lowest number of processors is employed. Besides, these schemes must guarantee the schedulability of the real-time tasks. To schedule sporadic real-time tasks on the multi-core systems, we presented a new semi-partitioned algorithm. The algorithm is based on the Rate-Monotonic Scheduling (RMS) policy and can successfully schedule any task sets with the system utilization up to the Monotonic Scheduling (RMS) policy and can successfully schedule any task sets with the system utilization up to 80%. Our extensive experiment results demonstrate that the proposed algorithm can significantly improve the scheduling performance compared with previous work in terms of system utilization and the number of required processors.

IndexTerms—Hard real-time systems, rate-monotonic, semi-partitioning, fixed-priority, task scheduling.

I. INTRODUCTION

Systems referred as real-time when their correct behavior depends not only on the correctness of operations they perform, but also by their accomplished time. For example in order to safely control an aircraft, in avionics and flight control, the software must respond to every request within a fixed time interval in order to safely control the aircraft [1]. Real-time systems are classified in two categories: soft and hard. A system is considered hard if it can guarantee that a certain events will always be carried out in less than a specific time and failure to timely respond to any of the events may cause a catastrophe, hence intolerable. The most important part of such a system is the scheduling algorithm [2], [3] that decides on the timing, preempting, and resuming of requests.

Two approaches traditionally considered for scheduling algorithms are Rate Monotonic (RM) and Earliest Deadline First (EDF), which are proved to be optimal for uniprocessor scheduling [4]. In presence of multi-core systems, these uniprocessor scheduling algorithms are no longer optimal [5]. There are extensive researches published on real-time scheduling for multi-core systems [6], [7]. For multi-cores, real-time scheduling algorithms can be categorized into two main classes: the partitioned and the global. A hybrid approach is called semi-partitioned scheduling. In the latter approach according to [6], [7], [8], [9], [10], most of the tasks are each assigned to one processor and a few tasks are each split into several subtasks to be executed on different processors. During runtime, the requests of each split task will be migrated among corresponding processors. Semi-partitioned approaches are shown to be sound and practical in real implementations and, theoretically, have the greatest performance compared to the partitioned and the global approaches [11].

In this paper, we present a partitioned Rate-Monotonic Least-Splitting (RMLS) strategy for the sporadic tasks with implicit deadline on multi-core platforms. The RMLS algorithm consists of two phases: assignment and scheduling. In the assignment phase, some tasks are each assigned to a separate processor while the load factor of other tasks are each split in two which are assigned to two different processors. The two splits of a split task, say \( \tau_i \), are assigned to two consecutive indexed processors; say \( P_{j} \) and \( P_{j+1} \). A distinguished property of our approach is that the processor with the lower index is allowed to run more than its share of the split task. It is clear that it is not permitted to execute the two splits of a task by two processors simultaneously. The number of split tasks in the approach is at the most \( M-1 \), where \( M \) is the number of processors. In contrast with the split tasks, other tasks that are referred to as non-split tasks and each one is executed by only one processor. In the scheduling phase, the assigned tasks to each processor are scheduled based on RM. Experimental results demonstrate that the proposed approach can outperform previous works in terms of the number of required processors. Besides, it is shown that RMLS has a utilization bound of 80% in the average for the sporadic task sets with the implicit deadlines.

The rest of paper is organized as follows: Section 2 considers the system model for the algorithm, Section 3 describes the new approach and the pseudo-code of the algorithm is then presented. Section 4 presents the simulation results, and conclusion is at the end.

II. SYSTEM MODEL

This section presents the system model considered in this paper. The real-time system being investigated consists of \( N \)
sporadic tasks, denoted as $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$, which are executed on $M$ identical processors, i.e., $\rho = \{P_1, P_2, \ldots, P_M\}$. For the sake of simplicity, we use $\Gamma_{\tau_i}$ to denote the task set on processor $P_i$. A task $\tau_i$ produces a (potentially infinite) sequence of requests. The arrival time of requests cannot be controlled and predicted by the scheduling algorithm before the request arrives. However, it is considered that the minimum interarrival time between two successive requests from the same task $\tau_i$ is known. Each task $\tau_i \in \Gamma$, is determined by a tuple $(C_i, T_i)$, where $T_i$ is the minimum interarrival time between any two consecutive requests of $\tau_i$, which is called "period of $\tau_i$" in this paper, and $C_i(0 < C_i \leq T_i)$ is the worst-case execution time of $\tau_i$.

Generally, the total cost of preemption, migration, and the runtime operation of the scheduler is assumed to be either negligible or they are subsumed into the worst-case execution time of each task (such as [12]). A processor can execute at most one request at a time and, on the other hand, a request cannot execute on two or more processors simultaneously. For the rest of the paper, we make two assumptions: first, the deadline of each task is equal to its period (i.e., implicit deadline); second, $\Gamma$ is sorted in decreasing priority order according to RMS, i.e., task $\tau$ has a higher priority than task $\tau_j$ if $i < j$. In which case $T_i \leq T_j$.

The utilization of a task $\tau_i$ is denoted as $u_i$, where

$$u_i = \frac{C_i}{T_i}.$$  \hspace{1cm} (1)

The total utilization of a task set $\Gamma$ is denoted as $U(\Gamma)$ where

$$U(\Gamma) = \sum_{i=1}^{n} u_i.$$ \hspace{1cm} (2)

The system utilization of task set $\Gamma$ on a multi-core system with $M$ processors is denoted as $U_u(\Gamma)$, where

$$U_u(\Gamma) = \frac{U(\Gamma)}{M}.$$ \hspace{1cm} (3)

Liu and Layland [4] demonstrated that a task set $\Gamma$ can be feasibly scheduled by RMS on a single processor as long as

$$U(\Gamma) \leq \Theta(N) = N(2^{1/N} - 1).$$ \hspace{1cm} (4)

$\Theta(N)$ is commonly referred to as the Liu & Layland bound.

III. THE RMLS ALGORITHM

In this section, we describe the RMLS algorithm, a semi-partitioned scheduling algorithm that consists of two phases: partitioning and scheduling. In the partitioning phase, most tasks are each assigned to only one processor, i.e., to be completely executed on that particular processor. Such a task is called a non-split task. A few tasks, called split tasks, are each split into two subtasks and each subtask is assigned to a different processor. Note that, once the partitioning phase terminates, for the scheduling phase, the assignment of tasks and subtasks to each processor is permanent, thus, each one can only run on that particular processor. However, an important property of our splitting approach is that if a task is split between two consecutive processor the processor with the lower index is allowed to run more than its share of the split task. In fact, it may run a whole request of such a task by itself. In the scheduling phase, the strategy decides when to run each task and for how long. In our approach, all tasks that are assigned to the same processor are scheduled strictly according to RMS policy.

Fig.1 presents RMLS algorithm for assigning tasks to the processors. In this algorithm, the index of the current processor to which task are assigned is denoted by $m$ and tasks are considered one at a time. Index $i$ denotes the task, which is currently being considered for assignment. At first, RMLS sorts tasks in decreasing order of RM priorities. The algorithm assigns tasks to the processors such that the utilization of each processor (except perhaps the last one) is equal to $\Theta(n)$. Note that $n$ is the number of assigned tasks to this processor. It is obvious that a task whose utilization is equal to or higher than $\Theta(2) = 83\%$ cannot share a processor with any other task. Therefore, all such tasks are removed from $\Gamma$ and each one is assigned to a different processor (Lines 1-8). No other task will be assigned to these processors because by adding other task the utilization of the corresponding processor will exceed safe utilization bound expressed by Equation (4).

The main idea of this algorithm for partitioning tasks is to check the possibility of assigning the highest priority unassigned task, $\tau_i$, to the current processor $m$. If it is possible, i.e., the sum of the current utilization of the processor $m$ and that of the task $\tau_i$ (i.e. $u_i$) is less than or equal Liu & Layland bound (i.e., the condition stated in the Line 12 is true) the task is assigned to processor $m$ and the algorithm moves to the next unassigned task. This is performed at Lines 12 and 13. Otherwise, if the processor $m$ is not filled yet, the utilization (or load factor) of the task $\tau_i$ is distributed between two processors $m$ and $m+1$ (Lines 14 to 17). That is, task $\tau_i$ is split into subtasks $\tau_{i1}$ and $\tau_{i2}$ such that the sum of $U(\Gamma_{\tau_i})$ and $u_i$ is equal to $\Theta(|\Gamma_{\tau_i}| + 1)$. The split task $\tau_{i1}$ is then assigned to the current processor $m$. Then, task $\tau_{i2}$ in $\Gamma$ is replaced by the subtask $\tau_{i2}$. This subtask is assigned to the processor $m+1$ in the next round of the loop. At the end, we make sure that the number of required processors is not higher than the total number of available processors in a system (Lines 19 to 22).

RMLS splits at the most $M-1$ in the whole system. We devised a method for preventing the two portions of a split task to executing simultaneously without any modifications in the ready time or deadline of subtasks. Under RMLS, the execution of that portion of the split task which is assigned to the lower index processor is not affected by the execution of that portion of the split task which is assigned to the higher index processor. However, the execution of that portion of the split task which is assigned to the higher index processor may be delayed because the lower index processor is running the same split task. To say it in other words, the lower index processor can run its own portion of the split task whenever desires to, but the higher order processor can only run its own
portion if the lower index processor is not running its buddy. Fortunately, this can easily be implemented in multicore systems.

Consider the situation in which the task $\tau_i$ is split to subtasks $\tau_{i1}$ and $\tau_{i2}$, and they are assigned to processors $m$ and $m+1$ respectively. Note that, task $\tau_{i1}$ has the lowest priority amongst all tasks assigned to processor $m$ while task $\tau_{i2}$ has the highest priority amongst all tasks assigned to processor $m+1$. This makes the chance of encountering the situation in which both processor wanting to run the task $\tau_i$ to be very low, although not zero. Suppose, at time $t$ both processor want to run their corresponding portion of task $\tau_i$, simultaneously. Concerning the split task, the worst case for the processor $m+1$ happens when at all times that the processor $m$ executes the split task the processor $m+1$ also wants to execute it but it is forbidden and there are no other tasks for this processor to execute. Due to paper size restriction, the proof that this is the worst case is not given here. For such a rare situation, the effective load of processor $m+1$ is increased and it is possible that some tasks do not meet their deadlines. To compensate, the effective utilization of task $\tau_i$ on processor $m+1$, $u_{i2}$ is taken to be:

$$u_{i2} = \frac{C_i}{T} - C_{i}$$ (5)

Although, by using Equation (5) in computing the total utilization of processor $m+1$ no task will overrun, it reduces the sum of the task utilization on the processor $m+1$ to less than Liu&Layland bound. However, in our simulations, the effect is shown to be not high.

IV. SIMULATION RESULTS

In this section, we investigate the performance of the proposed algorithm with the following experiments. We compare the RMLS algorithm with one of the most recent semi-partitioned algorithm, i.e. the SPA algorithm [9].

SPA sorts tasks in ascending order RM priority. In each step, SPA assigns the current task to the processor with the highest remaining utilization while the sum of its utilization is less than or equal to Liu&Layland utilization bound. If there is not enough remaining utilization to assign the complete task to the mentioned processor, the task is split such that one portion of the task fills the processor. The other portion of the task remains in the pool of unassigned task and it may split again in the future.

![Fig. 1. Pseudo code of RMLS Algorithm.](image)

![Fig. 2. Experimental results based on different task numbers for investigating (a) the number of processors and (b) utilization.](image)
Therefore, in this algorithm each task may split to several portions. If the average utilization of some task sets with respect to the number of available processors is larger than Liu&Layland bound the SPA simply aborts them. In our experiments, we varied the number of available processors such that both algorithm can schedule the set of tasks and compare the system utilization. Besides, we investigate the efficacy of changing the number of tasks on the utilization and the number of required processors. The number of tasks is varied from 20 to 200 and the number of processors is taken to be $M = 4, 8, \ldots, 64$. In addition, the utilization of each task is selected randomly from 0.01 to 1.00.

First, we compare the number of required processors for the two different approaches, i.e. RMLS and SPA, for different number of tasks. Diagram of Fig. 2(a) illustrates the required number of processors/cores in the system for the two mentioned algorithms when the number of tasks varies. From Fig. 2(b), we can observe that RMLS can achieve an overall system utilization significantly better than SPA approach. Note that, for instance when the task number is equal to 100, RMLS can achieve a system utilization equals to 80% which is an improvement of 1.14 times compared with that of SPA (70%). Also, when the task number reaches 180, RMLS can achieve utilization equals to 82% which is an improvement of 1.18 times over SPA (69%). Also, in this figure we can see that the system utilization of RMLS changes monotonically as the number of tasks increases. The utilization bound of RMLS on each processor is computed based on the number of tasks on that processor by Equation (4) ($n =$ the number of tasks in processor $)$. However, in SPA, Equation (4) is used to compute a general utilization gained based on all of tasks ($n =$ the number of tasks of all processors $). Therefore, the number of processors in RMLS is lower than that of SPA and due to the lower number of processors of RMLS its system utilization (gained from Equation (3)) is higher than that of SPA. Fig. 3 shows which algorithm has higher overall system utilization, for different number of processors. RMLS wins on SPA by considering this metric. Therefore, RMLS is more scalable than SPA in large systems or clusters.

V. CONCLUSION

In this paper, we proposed an algorithm (RMLS) for scheduling sporadic hard real-time tasks in multi-core systems based on RM, which assign tasks to the processors according to a semi-partitioned strategy. In RMLS, most tasks do not need to migrate and can be executed only on one processor, however, the migratory (split) tasks are assigned to only two processors. We illustrated that the proposed approach obtains utilization bound approximately 80% in average, which is better than existing other algorithms in the literature. Besides, the approach needs less number of processors than SPA for sporadic task sets with implicit deadlines. We left open the important question on how to extend this algorithm for sporadic tasks where the deadline of the tasks is not equal to their minimum inter-arrival time.

VI. REFERENCES


