Overrun-freeness verification of Rate-Monotonic Least-Splitting Real-Time Scheduler on Multicores

Mahmoud Naghibzadeh and Amin Rezaeian
Department of Computer Engineering
Ferdowsi University of Mashhad, Mashhad, Iran
naghibzadeh@um.ac.ir, amin.rezaeian@stu-mail.um.ac.ir

Abstract—In real-time task scheduling, semi-partitioning allows some tasks to be split into portions and each portion to be assigned to a different core. This improves the performance of the system but by counting each portion as a separate task it increases effective number of tasks to be scheduled. This research suggests a semi-partitioning method and assigns each partition to a separate core to be scheduled by the well-known scheduler called Rate-Monotonic (RM). To assure non-concurrent execution of portions of a task, there is no need to define release time for any portion. It is theoretically proven that with the proposed semi-partitioning and RM scheduling, all cores always run their tasks overrun-free. Besides, experimental results show that overall system utilization is noticeably boosted and also number of broken tasks is not higher than the best RM-based methods.

keywords: rate-monotonic least splitting, semi-partitioning, hard real-time scheduling

I. INTRODUCTION

A multicore system is composed of several processing elements, called cores, in which all cores can do their processing in parallel. They all share the same main memory but each can have its own private cache memory. With this structure, a sequential computation can be shared among many cores if not more than one core is executing the computation simultaneously [1]. While manufacturers tend to use multicore processors in new artifacts, software facilities to use all available power of multicores are yet to develop [2]. Scheduling algorithms play a significant role in overrun-freeness verification of hard real-time systems, i.e., making sure that every request is executed before its deadline. However, being multiprocessor/multicore adds a new dimension to the analysis; how to assign tasks or their requests to different processors/cores.

In this paper, the problem of scheduling periodic hard real-time task sets with implicit deadlines, i.e., when the relative deadline of a request is equal to its minimum request interval, on multicores is investigated. One way of categorizing scheduling methods for multicores is global, partitioned, and semi-partitioned, categories. In global scheduling, there is only one queue (or pool) of requests and each core takes its next request for execution from this queue. In partitioned, the set of tasks are divided and each partition is assigned to a separate core. Finally, in semi-partitioned, some tasks are wholly assigned to specific cores and some tasks are shared among more than one cores, with the restriction that not more than one core can work on a request of the shared task, simultaneously.

It is usually the case that semi-partitioned scheduling leads to a higher overall utilization of the whole system than global scheduling, for both fixed-priority and dynamic priority. However, partitioning is a time consuming task which is computationally equivalent to bin-packing problem that is known to be an NP-hard problem [3]. The good side of it is that partitioning is done off-line. Therefore, for small number of tasks the time taken by partitioning is tolerable, but for large number of tasks efficient heuristics are thought. A semi-partitioned approach binds a disjoint set of whole tasks to each core and lets remaining tasks be executed on multiple cores while everyone’s share is defined. In one of the researches on semi-partitioned methods in which Rate-Monotonic (RM) scheduler is used in each processor, worst case utilization is reported to be 0.693 [4].

In this paper, a different semi-partitioned scheduling algorithm called Rate-Monotonic Least Splitting (RMLS) is proposed for multicores. The scheduler of each core is basically RM with very minor changes to avoid simultaneous execution of a shared task by more than one processor. Using this algorithm, the number of split tasks is at the most equal to number of used cores minus one. Besides, no task is split in more than two portions. Splitting fewer tasks has two benefits, (1) effective number of tasks in the Liu and Layland’s bound, i.e., \( \Theta(n)=2(2^{m+1}-1) \), is reduced which in turn (2) increases overall system utilization.

The following notations are used throughout the paper. \( n \): total number of tasks, \( n_i \): total number of task and subtasks, \( m \): total number of available cores (or processors), \( m_i \): total number of used cores, \( C_i \): \( i^{th} \) task, \( T_i \): minimum interarrival time between any two consecutive requests of task \( C_i \), \( C \): maximum computation time needed by every request of task \( C_i \) with \( C_i \leq T_i \), and finally \( u_i \): the utilization of task \( C_i \) which is equal to \( C_i/T_i \).

In Section 2 related work is briefly reviewed: Section 3 describes the proposed RMLS semi-partitioned scheduling, Section 4 is the theoretical foundations and overrun-freeness proof of the algorithm, in Section 5 the algorithm is simulated and results are documented, and finally a summary and future work is presented in Section 5.

II. RELATED WORK

Many researchers have studied the semi-partitioning problem with Earliest Deadline First (EDF) scheduling [5-7].
The best known worst-case utilization bound using semi-partitioned EDF scheduling on multicore is 65% for Earliest Deadline Deferrable Portion (EDDP) algorithm [8]. Later, they proposed EDF with Window-constraint Migration (EDF-WM) which has less context switch overhead [9]. The NPS-F is a configurable method that has a tradeoff parameter between utilisation and preemptions [10]. On the other hand, relatively fewer algorithms are proposed for fixed-priority algorithms [11]. Rate Monotonic Deferrable Portion (RMDP) and Deadline Monotonic with Priority Migration (DM-PM) fixed-priority algorithms are proposed by Kato et al [12, 13]. The worst-case utilization bound of those algorithms is 50%.

The concept of portion and how a shared request migrates between two cores is explained in the same references. PDMS, HPTS, DS is proposed by Lakshmanan et al. [2] which reaches 65% utilization. This bound can be extended to 69.3% for light tasks, i.e., tasks with utilizations less than 0.41. Guan et al. proposed two algorithms called SPA1 and SPA2 [4, 11]. SPA2 has a pre-assignment phase in which special heavy tasks are assigned to processors, first. The number of split tasks is m-1 and SPA2 reaches the worst-case utilization bound of 0.693. This is equal to the Liu and Layland bound [14] for single processor systems. However, the worst-case bound in SPA2 is calculated using n which is the cardinality of the whole task-set, and every processor’s utilization must be less than or equal to that. For further reading on real-time scheduling algorithms and related issues refer to [15].

III. SEMI-PARTITIONED RMLS

Basic idea of the semi-partitioned method which is being presented here is presented in workshop [16]. There, the fundamental theorem which guarantees the overrun-freeness of system was not proven. In addition, none of the other theoretical results provided by this paper have appeared in that paper. A brief introduction of the method is repeated here and new findings and performance evaluations follow. The method is called Rate-Monotonic Least splitting (RMLS) because it is a semi-partitioned method in which only m-1 tasks are split.

Our experiments show that achieved processor utilization is higher than the best known results for general real-time systems, i.e., no restrictions on utilization of individual tasks, running with fixed-priority schedulers up to now. The proposed assignment algorithm is composed of two steps, see Algorithm 1.

In Step 1 (Lines 1 to 9), all pairs of tasks, τi and τj, with total utilizations satisfying Θ(3) ≤ Ui + Uj ≤ 1 are found and each pair is assigned to a separate processor. Meanwhile, heavy tasks, i.e., a task τi with Ui ≥ Θ(2), are recognized and each such task is assigned to a separate processor. The scheduler of each core with two tasks is taken to be Delayed Rate Monotonic (DRM) which is a modified version of RM. Details of how DRM works are explained in [17]. Any system composed of two tasks with utilization less than or equal to one can run overrun-free with DRM. The scheduler of all other sets will be the conventional RM.

Step 1 serves two purposes: (1) it increases the number of cores with high, and (2) it increases the number of processors with no split task, i.e., decreases the total number of split-tasks.

### Data: Task-Set

<table>
<thead>
<tr>
<th>Result</th>
<th>Processor assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Find tasks with largest and smallest utilizations;</td>
</tr>
<tr>
<td>2</td>
<td>While smallest and the largest tasks are different</td>
</tr>
<tr>
<td>3</td>
<td>If it’s worth assigning them to a separate processor</td>
</tr>
<tr>
<td>4</td>
<td>do so and find the next largest and smallest;</td>
</tr>
<tr>
<td>5</td>
<td>Else if sum of both utilization is too large</td>
</tr>
<tr>
<td>6</td>
<td>discard current largest and find next largest;</td>
</tr>
<tr>
<td>7</td>
<td>Else discard current smallest and find next smallest;</td>
</tr>
<tr>
<td>8</td>
<td>Take an unassigned processor as current-processor</td>
</tr>
<tr>
<td>9</td>
<td>While there is an unassigned task</td>
</tr>
<tr>
<td>10</td>
<td>Find the unscheduled task with highest priority as</td>
</tr>
<tr>
<td>11</td>
<td>current-task and assign it to current-processor;</td>
</tr>
<tr>
<td>12</td>
<td>If current-processor is not overrun-free</td>
</tr>
<tr>
<td>13</td>
<td>Remove the task with least loss from current-processor</td>
</tr>
<tr>
<td>14</td>
<td>as split task, split it and assign its first-portion to</td>
</tr>
<tr>
<td>15</td>
<td>current-processor;</td>
</tr>
<tr>
<td>16</td>
<td>Take a new processor and make it current-processor and</td>
</tr>
<tr>
<td></td>
<td>assign the second-portion of the task to it;</td>
</tr>
</tbody>
</table>

Algorithm 1. Packing algorithm

In step 2 (Lines 11 to 16.), all unassigned task are sorted in decreasing order of RM priorities, i.e., non-descending order of their request interval lengths. An empty core is picked and starting from the first unassigned task, tasks are assigned to the core one at a time until the current task, say task τi, will make the core overloaded. Task τi is also assigned to the core but one of the assigned tasks, except the one which is shared with the previous core, is selected to be split and shared with the next core. The split task may happen to be τi. In the following example a scenario is explained and it is clarified what criteria is used to select a task to be split. The selected task is split into two subtasks such that the first subtask is assigned to the current core and makes it full with respect to Liu and Layland’s bound for the respective number of tasks and subtasks in this processor.

A new core is taken and the second portion of the current split task is assigned to it. The process of assigning tasks to cores continues until all tasks are assigned. If there are enough cores the assignment successfully complete.

**Example 1:** Suppose the current core is p1 and task τi is the task which is split into two portions τi1 and τi2 with execution times Ci1 and Ci2, respectively. The utilization of τi is uτi = Ci1 / Ti as for core p1. A new core, p1+i, is taken and the second portion of task τi, τi2, is assigned to this core. Although the actual utilization of this portion is Ci2 / Ti, its effective utilization on core p1+i is taken to be

\[ u_{i2} = \frac{C_{i2}}{T_{i}} \]

This is because, in the worst case, a request from subtask τi2 will have only Ti - Ci1 time to be executed. Effective utilization of the subtask is always greater than or equal to its actual utilization. Therefore, UtilizationLoss = \( \frac{C_{i2}}{T_{i} - C_{i1}} - \frac{C_{i2}}{T_{i}} \geq 0 \). Since higher utilization loss causes lower total utilization of system, when we are forced to split a task, a whole task with the least utilization loss is selected.
IV. OVERRUN-FRIENESS VERIFICATION OF RMLS

In this section, we assume that two processors \( p_k \) and \( p_{k+1} \) share a task \( \tau = (T, C_i) \) and for each request of the common task \( C_i \) is executed by \( p_k \) and \( C_{i2} \) is executed by \( p_{k+1} \) such that \( C_i = C_{i1} + C_{i2} \).

**Lemma 1:** If Liu & Layland’s bound is satisfied by all processors, the second part of a request from a shared task, \( \tau \), between two processors, \( p_k \) and \( p_{k+1} \), never overruns.

**Proof:** The preference of executing a request from a shared task \( \tau \) between processors \( p_k \) and \( p_{k+1} \) is always given to \( p_k \). Whenever \( p_k \) is not executing such a request \( p_{k+1} \) will be executing it unless the execution of the second part of the request is completed. This is because this request has the highest priority in \( p_{k+1} \). Therefore, in the worst case, the execution of the second part of the task will be complete after a time length of \( C_i \) is passed since the request is received, where \( C_i \leq T_i \).

**Definition 1:** A conflict-idle period is a time interval in which both processors, \( p_k \) and \( p_{k+1} \), that share the shared task, \( \tau \), want to run a request from the task but because \( p_k \) is given a higher precedence it will proceed with the execution; and at the same time, there is no other pending request for processor \( p_{k+1} \) within this period and it will be idle. Note that, not all conflict periods of processors \( p_k \) and \( p_{k+1} \) are necessarily conflict-idle because if there are other requests for \( p_{k+1} \) it will proceed with their execution and hence it will not be idle.

**Lemma 2:** If the utilization of each of the two processors, \( p_k \) and \( p_{k+1} \), which share a task, \( \tau \), is not higher than Liu and Layland’s bound and there is no conflict-idle period with respect to the shared task, both processors always run their corresponding tasks safely.

**Proof:** Since processor \( p_k \) has a higher precedence to run the shared task \( \tau \) than \( p_{k+1} \), this processor will always run safe. On the other hand, the only effect that \( p_k \) can have on tasks of processor \( p_{k+1} \) is that it may cause the execution of the second part of a request from the shared task to be postponed. This may harm the safety of the shared task in \( p_{k+1} \) but it may be beneficial to other tasks of this processor. However, in Lemma 1 it is proven that the second part of a request from a shared task never overruns.

Lemmas 1 and 2 will hold even if actual utilization of subtask \( T_{i2} \), i.e., \( \frac{c_{i2}}{T_{i2}} \), is used in the computation of utilization of \( p_{k+1} \). It is for compensation of possible conflict-idle periods that, in general, effective utilization of the shared task on processor \( p_{k+1} \) is computed as \( \frac{c_{i2}}{T_{i2} - c_{i1}} \).

**Definition 2:** Effective utilization of a request (not a task or subtask) at a given time \( t \) is defined as below:

\[
E_{\tau_{i1}} = \frac{\text{Remaining execution time of } \tau_i}{\text{Remaining time to deadline for } \tau_i}
\]

For example, suppose task \( \tau = (10, 4) \) has generated a request at time 20 and current time is 26 and up to now this request has received 1.5 unit of CPU time then the effective utilization of the request at time 26 is \( (4-1.5)/(30-26) = 0.625 \).

**Lemma 3:** Suppose two processors \( p_k \) and \( p_{k+1} \) share a task \( \tau \). Effective utilization of a request from \( \tau \) for processor \( p_{k+1} \) is maximal at the exact time when the execution of processor \( p_k \)’s share of this request is completed and \( p_k \) starts this request immediately after it is generated and continues until completion.

**Proof:** Suppose as soon as a request from \( \tau \) is generated at a time \( t_0 \) processor \( p_k \) starts executing it until its share is finished at time \( t_0 + C_{i1} \). At this time effective utilization of the subtask \( T_{i2} \) on processor \( p_{k+1} \) is equal to \( \frac{c_{i2}}{T_{i2} - c_{i1}} \). We show that this is in fact maximal effective utilization of \( T_{i2} \), which means subtask \( T_{i2} \)’s effective utilization never becomes greater than this. Recall that requests of task \( \tau \) have the highest priority in processor \( p_{k+1} \). This implies that any request from this task will be immediately pick up for execution by \( p_{k+1} \) if \( p_k \) is not executing it. On the other hand, if the execution of the second part of a request from task \( \tau \) is completed by processor \( p_{k+1} \), then its effective utilization becomes zero and remains zero until a new request is generated from the same task. With these points in mind, consider a situation where at any time \( t_1, t_0 \leq t_1 \leq t_0 + C_{i1} \), processor \( p_k \) has executed this request for duration of length \( a \), \( a \leq C_{i1} \), and processor \( p_{k+1} \) has executed the same request for duration \( b, b < C_{i2} \) and \( a + b = t_1 - t_0 \). See Figure 1.

![Fig. 1. A sample execution of parts of a split task.](image)

At time \( t_1 \) effective utilization of \( T_{i2} \) is \( \frac{c_{i2} - b}{T_{i2} - (a + b)} \).

Since \( a \leq C_{i1} \),

\[
\frac{c_{i2} - b}{T_{i2} - (a + b)} \leq \frac{c_{i2} - b}{T_{i2} - c_{i1}} = \frac{c_{i2} - b}{T_{i1} - c_{i1}}
\]

To show that maximal effective utilization of \( T_{i2} \) is \( \frac{c_{i2}}{T_{i2} - c_{i1}} \), it has to be shown that

\[
\frac{c_{i2} - b}{T_{i2} - c_{i1}} \leq \frac{c_{i2}}{T_{i2} - c_{i1}}
\]

That is, \( (c_{i2} - b)(T_{i2} - c_{i1}) \leq c_{i2}(T_{i2} - c_{i1}) - b \)

Or,

\[-bT_{i1} + bC_{i1} \leq -bC_{i2} \]

Or,

\[b(C_{i1} + C_{i2}) \leq bT_{i1} \]

which is always true because \( b \) is positive and \( C_{i1} + C_{i2} \leq T_{i1} \).

**Theorem 1:** If effective utilization of each of two processors \( p_k \) and \( p_{k+1} \) which share a task \( \tau \) is not greater than Liu and Layland’s bound, both processors will always safely run their corresponding tasks.

**Proof:** This theorem is similar to Lemma 2 in which it is assumed that there will be no conflict-idle period. However, here, this restriction is removed. In Lemma 2, it is mentioned that processor \( p_{k+1} \) does not have any influence on the execution of tasks and subtasks assigned to processor \( p_k \). Since
Liu and Layland’s bound is satisfied for \( p_k \) it will always safely run its assigned tasks. In the packing algorithm, the utilization of the shared task on processor \( p_{k+1} \) is computed as \( C_{k+1} \) which, based on Lemma 3, is the maximum utilization which \( T_k \) can ever impose on the processor. On the other hand, the utilization is taken to be less than or equal Liu and Layland’s bound. Therefore, this processor will always safely run its assigned tasks, too.

V. Simulations

In this section, the proposed method is compared with SPA2. We used UUnifast algorithm [18] to produce random unbiased task-sets in which each task’s utilization must not exceed one. For each category of task sets, e.g., task sets with total utilization equal to 4, the total of 3000 task-sets, with different number of tasks are generated. For RLMS we do not have to know the number of cores in advanced but we must know it for SPA2. Therefore, for a fair comparison, for SPA2 and for each tasks set, we had to find the overrun-free case with the least number of processors. As the minimum number of processors needed for each method are found, the average utilization of all processor is calculated by dividing overall utilization of the task-set by the number of processors used.

To be brief, only two experiments are shown here. In the first experiment, for task-sets with total utilization equal to 16 and task sets of sizes 38, 48, 67, 106, and 183, the calculated average utilizations are depicted in Figure 2. RLMS leads to an average utilization which is always higher than that of SPA2. Figure 3 shows number of cores used by each method.

![Fig. 2. Average of performance, by each method, for U=16](image1)

![Fig. 3. Number of cores used for each method, for U=16](image2)

In the second experiment, rates of schedulable tasks are compared. Figure 4 shows the result of one such experiment where average utilization of task sets grows from 0.5 to 1.0.

![Fig. 4. Rate of schedulable task-sets](image3)

More experiments should be performed on RMLS and also should be compared with other methods. Finding a utilization bound for RMLS is in progress.

### References


