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 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 portioning 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., $\Theta(n)=2(2^{1/n}-1)$, is reduced which in turn (2) increases overall system utilization.

The following notations are used throughout the paper. n: total number of tasks, n_1: total number of task and subtasks, m: total number of available cores (or processors), m_1: total number of used cores, τ_i: i^{th} task, T_i: minimum interarrival time between any two consecutive requests of task τ_i, C_i: maximum computation time needed by every request of task τ_i with $C_i \leq T_i$, and finally u_i: the utilization of task τ_i which is equal to Ci/Ti.

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 multicores 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*-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, τ_i, and τ_j, with total utilizations satisfying $\Theta(3) \leq U_i + U_j \leq 1$ are found and each pair is assigned to a separate processor. Meanwhile, heavy tasks, i.e. a task τ_l with $U_l \geq \Theta(2)$, 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
Result: Processor assignments
1  Find tasks with largest and smallest utilizations;
2  While smallest and the largest tasks are different
3    If it's worth assigning them to a separate processor
4      do so and find the next largest and smallest;
5    Else if it's worth to assign largest task to a separate
6          processor do so and find the next largest;
7        Else if sum of both utilization is too large
8              discard current largest and find next largest;
9            Else discard current smallest and find next smallest;
11 Take an unassigned processor as current-processor
12 While there is an unscheduled task
13   Find the unscheduled task with highest priority as
       current-task and assign it to current-processor;
14   If current-processor is not overrun-free
15     Remove the task with least loss from current-processor as
         split-task, split it and assign its first-portion to
         current-processor;
16     Take a new processor and make it current-processor and
         assign the second-portion of the task to it;
```

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 τ_i, will make the core overloaded. Task τ_i 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 τ_i. 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 p_k and task τ_i is the task which is split into two portions τ_{i1} and τ_{i2} with execution times C_{i1} and C_{i2}, respectively. The utilization of τ_{i1} is $u_{i1} = \frac{C_{i1}}{T_i}$ for core p_k. A new core, p_{k+1}, is taken and the second portion of task τ_i, τ_{i2}, is assigned to this core. Although the *actual utilization* of this portion is $\frac{C_{i2}}{T_i}$, its *effective utilization* on core p_{k+1} is taken to be

$$u_{i2} = \frac{C_{i2}}{T_i - C_{i1}} \quad (1)$$

This is because, in the worst case, a request from subtask τ_{i2} will have only T_i-C_{i1} 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-Freeness Verification of RMLS

In this section, we assume that two processors p_k and p_{k+1} share a task $\tau_i = (T_i, C_i)$ and for each request of the common task C_{i1} 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, τ_i, between two processors, p_k and p_{k+1}, never overruns.*

Proof: The preference of executing a request from a shared task τ_i 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, τ_i, 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 tasks, τ_i, is not higher than Liu and Layland's bound and there is no conflict-idle period with respect to the share task, both processors always run their corresponding tasks safely.*

Proof: Since processor p_k has a higher precedence to run the shared task τ_i 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 τ_{i2}, i.e. $\frac{C_{i2}}{T_i}$, 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 $\frac{C_{i2}}{T_i - C_{i1}}$.

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

$$E_{\tau_i,t} = \frac{Remaining\ execution\ time\ of\ \tau_i}{Remaining\ time\ to\ deadline\ for\ \tau_i}$$

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 τ_i. Effective utilization of a request from τ_i 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 τ_i 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 τ_{i2} on p_{k+1} is equal to $\frac{C_{i2}}{T_i - C_{i1}}$. We show that this is in fact maximal effective utilization of τ_{i2}, which means subtask τ_{i2}'s effective utilization never becomes greater than this. Recall that requests of task τ_i 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 τ_i 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, $t_0 \leq t_1 \leq t_0+C_i$, 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.

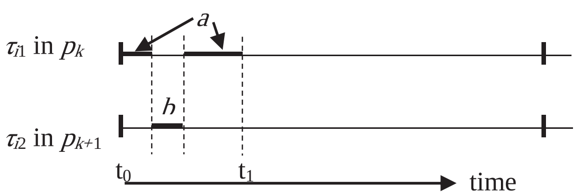

Fig. 1. A Sapmle execution of parts of a split task.

At time t_1 effective utilization of τ_{i2} is $\frac{C_{i2}-b}{T_i-(a+b)}$.

Since $a \leq C_{i1}$,

$$\frac{C_{i2}-b}{T_i-(a+b)} \leq \frac{C_{i2}-b}{T_i-(C_{i1}+b)} = \frac{C_{i2}-b}{T_i-C_{i1}-b}$$

To show that maximal effective utilization of τ_{i2} is $\frac{C_{i2}}{T_i-C_{i1}}$ it has to be shown that $\frac{C_{i2}-b}{T_i-C_{i1}-b} \leq \frac{C_{i2}}{T_i-C_{i1}}$.

That is, $(C_{i2} - b)(T_i - C_{i1}) \leq C_{i2}(T_i - C_{i1} - b)$

Or,

$$-bT_i + bC_{i1} \leq -bC_{i2}$$

Or,

$$b(C_{i1} + C_{i2}) \leq bT_i$$

which is always true because b is positive and $C_{i1} + C_{i2} \leq T_i$.

Theorem 1: *If effective utilization of each of two processors p_k and p_{k+1} which share a task τ_i, 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 $\frac{C_{i2}}{T_i - C_{i1}}$ which, based on Lemma 3, is the maximum utilization which τ_{i2} 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.

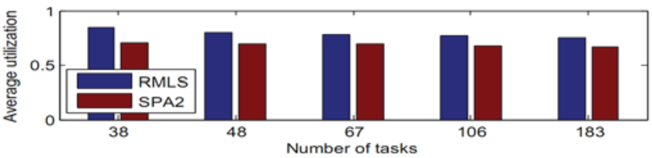

Fig. 2. Average of performance, by each method, for U=16

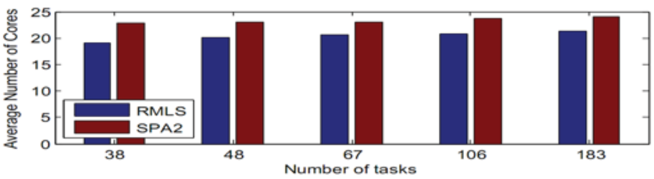

Fig. 3. Number of cores used for each method, for U=16

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.

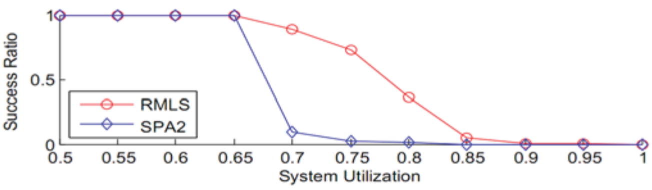

Fig. 4. Rate of schedulable task-sets

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

[1] C.L. Liu, "Scheduling algorithms for multiprocessors in a hard real-time environment". *JPL Space Programs Summary*, vol. 37-60, pp. 28–31, 1969.

[2] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on*. IEEE, 2009, pp. 239–248.

[3] M. R. Gary and D. S. Johnson: "Computers and Intractability; A Guide to the Theory of NP-Completeness" (W. H. Freeman & Co.), 1979

[4] N. Guan, M. Stigge, W. Yi, and G. Yu, "Fixed-priority multiprocessor scheduling with liu and layland's utilization bound," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*. IEEE, 2010, pp. 165–174.

[5] J. Anderson, V. Bud, and U. Devi, "An edf-based scheduling algorithm for multiprocessor soft real-time systems," in *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*, 2005, pp. 199–208.

[6] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in Embedded and Real-Time Computing Systems and Applications, *2006. Proceedings. 12th IEEE International Conference on*. IEEE, 2006, pp. 322–334.

[7] A. Burns, R. I. Davis, P. Wang, and F. Zhang, "Partitioned edf scheduling for multiprocessors using a c=d task splitting scheme," *Real-Time Systems*, vol. 48, no. 1, pp. 3–33, 2012.

[8] S. Kato and N. Yamasaki, "Portioned edf-based scheduling on multiprocessors," in *Proceedings of the 8th ACM international conference on Embedded software*. ACM, 2008, pp. 139–148.

[9] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on*. IEEE, 2009, pp. 249–258.

[10] Bletsas, K. & Andersson, B. "Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound" *Real-Time Systems*, vol. 47, no. 4, pp. 319-355, 2011

[11] N. Guan and W. Yi, "Fixed-priority multiprocessor scheduling: Critical instant, response time and utilization bound," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 2470–2473.

[12] S. Kato and N. Yamasaki, "Portioned static-priority scheduling on multiprocessors," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–12.

[13] S. Kato and N. Yamasaki, "Semi-partitioned fixed-priority scheduling on multiprocessors," in *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*. IEEE, 2009, pp. 23–32.

[14] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[15] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011.

[16] M. Naghibzadeh, P. Neamatollahi, R. Ramezani, A. Rezaeian, and T. Dehghani, "Efficient semi-partitioning and rate-monotonic scheduling hard real-time tasks on multi-core systems," in *Industrial Embedded Systems (SIES), 2018 8th IEEE International Symposium on*. IEEE, 2013, pp. 85–88.

[17] M. Naghibzadeh, and K.H. Kim "The yielding-first rate-monotonic scheduling approach and its efficiency assessment", *International Journal of Computer System Science & Engineering*, 2003, pp. 173-180

[18] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.