Scheduling Hard Real-time Tasks on Multi-core using Intelligent Rate-monotonic

Amin Rezaeian, Mahmoud Naghibzadeh and Peyman Neamatollahi  
Dept of Computer Engineering  
Ferdowsi University of Mashhad  
Mashhad, Iran  
Email: amin.rezaeian@stu-mail.um.ac.ir, naghibzadeh@um.ac.ir, neamatollahi@ieee.org

Abstract—Recently, researchers have applied semi-partitioned approaches to improve performance of hard real-time scheduling algorithms in multiprocessor architectures. RMLS is one of these methods. However, advantages of using semi-partitioned methods are often limited by well-known scheduling algorithms such as RM and EDF, which the former is simple but inefficient and the latter is efficient but has high processing overhead. There is an intelligent algorithm working on uniprocessor named IRM which takes advantages of both RM and EDF algorithms using that we present a new method called intelligent rate-monotonic least splitting to improve RMLS. Experimental results show that the proposed algorithm outperforms many other algorithms in literature in terms of processor utilization.

Keywords—Real-time, Multi-core, Scheduling, IRM, Semi-partitioning

I. INTRODUCTION

Many applications such as aerospace control systems run types of tasks which must be done before a deadline. In these kind of applications, software system namely the scheduler has to schedule tasks in a timely manner. This means that besides keeping semantical correctness of each task, scheduler must be committed to finish execution of each task before its stated deadline. These kind of systems are called hard real-time systems, in which exceeding of even one task’s ending time over its deadline may cause catastrophe [1]. Rate-monotonic (RM) and earliest deadline first (EDF) are scheduling algorithms for hard real-time systems. It is proved that these two are optimal for single processor scheduling [2].

RM and EDF can be considered at two extreme ends of schedulers, which the former is simpler to implement and has less overhead though it does not use all of processor’s utilization and the latter schedules whole processor’s utilization but it has considerable processing overhead [3]. Scheduling algorithms can be designed to stand somewhere between these two extremes. These algorithms must balance efficiency and overhead in a way that the utilization moves toward EDF’s and low overhead as the overhead of RM. One of such algorithms is IRM [4]. More description about IRM is brought in section III.

From the scheduling algorithms perspective, scheduling algorithms for single processor like RM and EDF are not optimal for multi-cores [1]. This leads to creation of algorithms concerning about multi-processor systems. These algorithms are roughly divided into global and partitioned scheduling. Global scheduling uses a single global queue to schedule whole tasks as they requested over all of processors [5]. On the other hand, in partitioned scheduling, each processor has its own queue, which is completely independent and unaware of other processors [1]. According to Dhall’s effect [6], for some task sets global scheduling could achieve less utilization than partitioned scheme. In partitioned scheme, it’s necessary that before starting the system, an allocation algorithm assigns tasks permanently to each processor. Whereas optimum allocation of tasks to each processor is a solution to solve bin-packing problem and because of complexity of this problem, the optimum allocation for partitioning scheme is not practical [1] [7] [8].

Because of the problems of global and partitioned methods, e.g. global method is effected by Dhall’s effect and partitioned method is bonded to the solution of bin-packing problem, many researches [1] [9] [10] [7] [11] [12] tried to combine positive features of these methods, and make a new scheme for scheduling, currently known as semi-partitioned scheduling. Same as partitioning, in the first step, some new methods allocate some of tasks permanently to each processor, which usually do not fully utilize those processors. Then they allow the tasks which are not scheduled yet to run on several processors. However, in these algorithms, one question remains challenging, which is, how to force tasks to move between processors without losing their (if they have) positive real-time aspects such as long intervals and/or late deadlines. The rate-monotonic least-splitting (RMLS) scheduling algorithm [12] is a semi-partitioned algorithm which keeps these positive features of split-tasks and utilizes processors better than similar algorithms. RMLS is described in section III.

In this paper, we introduce a new semi-partitioned scheduling method named IRMLS for sporadic task-sets with implicit deadlines on multi-core platform. We changed the packing algorithm of RMLS, which causes using less processors. This consists of proposing a two phased algorithm for packing, and for its first phase we introduced two different algorithms, that one of them is very fast and another is optimal. We also made changes in RMLS scheduler algorithm, to cooperate with IRM. IRMLS takes advantages of both IRM and RMLS. Experimental results shows that this algorithm is highly utilizing processors, and it uses less number of processors than similar algorithms.

The rest of this paper is organized as follows. System model and background are described in sections II and III respectively. In section IV we brought the proposed algorithm. After experimental results in section V, we conclude our paper in section VI.
II. SYSTEM MODEL

The system model considered in this paper is presented in the current section. The investigated real-time system consists of \( n \) independent tasks denoted as \( \Gamma = \{ t_1, t_2, ..., t_m \} \). These tasks have implicit deadlines and are executed on \( m \) identical processors. Processors are denoted as \( \rho = \{ P_1, P_2, ..., P_m \} \). We use \( \Gamma_{pk} \) to denote the task-set on processor \( P_k \). \( |\Gamma_{pk}| \) and \( \Gamma_{pk} \) denote number of tasks in processor \( P_k \). A task \( t_j \) produces a (potentially infinite) sequence of requests (or jobs). The arrival time of requests cannot be controlled and predicted by the scheduling algorithm before the request arrives. However, it is considered that the time between two successive requests from the same task \( t_j \) is known. Each task \( t_j \) is determined by a tuple \((c_j, t_j)\), where \( t_j \) is the minimum interarrival time between any two consecutive requests of \( t_j \), and \( c_j \) \((0 < c_j \leq t_j)\) is the worst-case execution time of \( t_j \). We used \( \Theta(n) \) as Liu and Layland bound for \( n \) tasks, as shown in (1).

\[
\Theta(n) = n(2^k - 1) \tag{1}
\]

The utilization of a task \( t_j \) is denoted by \( u_j \) for \( t_j \in \Gamma \) and \( u_{t'} \) for \( t' \in \Gamma \). The utilization of task-set in processor \( P_j \) is denoted as \( U(\Gamma_j) \). We also described tasks with high utilization i.e. tasks with \( u_j > \Theta(2) \) as large tasks and other tasks as “small tasks”.

III. REFERENCE ALGORITHMS

There are many algorithms in semi-partitioned [11] [13] [8] [14]. However, we apply two algorithms, RMLS and IRM, to present IRMLS. Therefore we explain these two algorithms in the following.

A. RMLS

“Rate-Monotonic Least Splitting” (RMLS), similar to most of other semi-partitioned algorithms, includes two phases: (1) Packing and (2) Scheduling. Each of large tasks (tasks with utilization bigger than 0.83) is assigned to a separate processor at the beginning of the first phase. Then, tasks are sorted according to their priority, namely in ascending order of their intervals. Then for each processor \( P_k \), the following operations are done:

- While the inequality (2) is correct, remove a task from the head of the sorted list and allocate it to the current processor (\( P_k \)).

\[
|P_k| \leq \sum_{i=1}^{u_i} \leq \Theta(|P_k|) \tag{2}
\]

- The first task that which adding it causes the inequality (2) to become incorrect is considered a “split task” between processors \( P_k \) and \( P_{k+1} \). However, the amount of time this split-task is allowed to spend on processor \( P_k \) equals to \( c_{a1} \), which achieves as in (3).

\[
c_{a1} = t_s(\Theta(|P_k|)) - \sum_{i=1}^{|P_k|-1} u_i \tag{3}
\]

\( c_{a2} \) is the remaining running time of split-task in processor \( P_{k+1} \) and it is calculated as follows.

\[
c_{a2} = c_a - c_{a1} \tag{4}
\]

Now the split-task is shared between processors \( P_k \) and \( P_{k+1} \). To avoid overruns, in processor \( P_{k+1} \), the utilization of the split-task is considered bigger than what it really is, which is calculated in (5).

\[
u_{a2} = \frac{c_{a2}}{t_a - c_{a1}} \tag{5}
\]

The remaining tasks in the ordered list are also packed into other processors in a similar course of action.

B. IRM

Many researchers believe that rate-monotonic is simpler than earliest deadline first and has low processing overhead [4] [7]. IRM [4] uses the simplicity of RM algorithm, however it assumes that we know deadline for each task, and takes advantage of this assumption. IRM is similar to RM, with only one additional condition which must be checked for every preemption. Both IRM and RM sort tasks in ascending order of their intervals, and use the identification number assigned to each task in ordered list as its priority. If a higher priority task requests, the lower priority task is always preempted in RM, although IRM checks their deadlines. If the deadline of the current running task is earlier than the requesting one, IRM keeps running the lower priority task. It is proved that IRM never overruns a task in a task-set that its overall utilization is less than or equal to Liu and Layland bound.

From the scheduler’s overhead perspective, IRM scheduler is based on RM, in which only a task with higher priority (according to RM) and earlier deadline can preempt a lower priority task. It means that whenever a higher priority task arrives, a rate-monotonic scheduler preempt tasks for the current running task, while an IRM scheduler verifies the deadlines and allows the task with earlier deadline to run on the processor. This method does not need “sorting data structures” e.g. a min-heap as EDF needs, thus no maintenance is required and overhead is just a condition in the scheduler algorithm. Thus the time complexity of scheduler’s overhead remains \( O(1) \), instead of EDF’s \( O(\log n) \) which maintains min-heap up-to-date.
IV. PROPOSED ALGORITHM

This section describes the Intelligent Rate-Monotonic Least-Splitting (IRMLS) for scheduling sporadic task-sets with implicit dead-lines on multi-core platform. This is a semi-partitioned algorithm, consists of two phases: (1) Packing and (2) Scheduling. The first phase is offline, which assigns tasks to processors. IRMLS uses the RMLS packing algorithm, yet some changes is made in this phase to use positive features of IRM. In the second phase, since the essence of RMLS causes naive IRM to overrun some task-sets, a cooperative method is suggested.

A. Packing

As RMLS uses an RM-based scheduler, its packing phase packs every large task (with \( u_i \geq \Theta(2) \)) in a single processor. Allowing a large task to be shared among two processors will cause increasing in cardinality of task-sets in both processors, and subsequently will decrease the feasibility bound of those processors. On the other hand, a pair of tasks which their sum of utilization is less than or equals to one are schedulable using IRM. If such tasks are large enough and they are shared among processors, they cause a fall in feasibility bound of their processors, as it happened to RMLS. Hence to avoid allowing such pairs to be shared, the packing phase of the proposed method (IRMLS) packs every large pair of tasks in a single processor. To be precise, these pairs must have the property given in (6). It means every pair of tasks which sum of their utilization is less than or equals to one, and is greater than or equals to Liu and Layland bound for three tasks can be packed in a single processor.

\[ \Theta(3) \leq u_i + u_j \leq 1 \]  

(6)

Using this positive feature of IRM, packing phase must find the set of task pairs with maximum collective load factor. These task pairs must have the property given in (6). We used two different methods to do so: (1) Optimal and (2) Fast.

The optimal method in finding pair tasks consists of finding all possible pairs of tasks with the time complexity of \( O(n^2) \) for a task-set of size \( n \), and selecting pairs from this list which have the maximum collective load, albeit the time complexity of selection will be \( O(2n^2) \), because if one decides to select greedy, and always selects a best pair, this selection may destroy other possible pairs. As an example, for a set of four tasks with utilizations equal to \{0.9, 0.7, 0.1, 0.05\}, a greedy approach produces the pair of (0.9, 0.1), while the optimal produces both pairs of (0.9, 0.05) and (0.7, 0.1).

On the other hand, the fast method checks only for some possible pairs, and discards other pairs. In this method, firstly tasks are sorted on descending order of their utilization. Then selecting tasks from the front and the rear of the ordered list, we check if selected tasks meet the condition in (6). If so, packing will be done and both tasks will be out of the list. If the condition is not met, one of those candidate tasks will be discarded. The decision is made based on the condition in equation (6) that is not met. If \( u_i + u_j < \Theta(3) \), we discard the smaller candidate, and else the next task from the head of the list will be selected. However before discarding the large task, we check if this task meets the inequality \( \Theta(2) \leq \tau \).

```
Data: Task-set \( \Gamma \); // Includes \( t_i \), \( c_i \) and \( u_i \) for each task \( i \)
Result: Packing \( \rho \)
1 \( \rho \leftarrow \{ \} \);  
2 \( m \leftarrow 0 \);  
3 \( \Gamma' \leftarrow \Gamma \);  
4 Sort \( \Gamma' \) in descending order of tasks’ utilizations;  
5 \( i \leftarrow 1, j \leftarrow \text{length}(\Gamma) \);  
6 while \( i < j \) do  
   7 if \( \Theta(3) \leq u_{i'} + u_{j'} \leq 1 \) then  
      8 \( m \leftarrow m + 1 \);  
      9 Add \( P_m \) to \( \rho \);  
     10 Move \( \tau_{i'} \) and \( \tau_{j'} \) from \( \Gamma \) to \( P_m \);  
     11 \( i \leftarrow i + 1, j \leftarrow j - 1 \);  
   12 else if \( \Theta(2) \leq u_{i'} \) then  
      13 \( m \leftarrow m + 1 \);  
      14 Add \( P_m \) to \( \rho \);  
     15 Move \( \tau_{i'} \) from \( \Gamma \) to \( P_m \);  
   16 else if \( \Theta(3) \leq u_{i'} + u_{j'} \) then  
      17 \( i \leftarrow i + 1 \);  
     18 else  
      19 \( j \leftarrow j - 1 \);  
   20 end  
21 end  
22 end  
23 \( m \leftarrow m + 1 \);  
24 Add \( P_m \) to \( \rho \);  
25 while \( \Gamma \neq \emptyset \) do  
26 \( i \leftarrow 1, j \leftarrow \text{length}(\Gamma) \);  
27 if \( U(\Gamma) = \Theta(|\Gamma|) \) then  
      28 \( m \leftarrow m + 1 \);  
   29 Add \( P_m \) to \( \rho \);  
26 end  
29 \( i \leftarrow 1, j \leftarrow \text{length}(\Gamma) \);  
30 if \( U(\Gamma) + u_i \leq \Theta(|\Gamma| + 1) \) then  
   31 Move \( \tau_i \) from \( \Gamma \) to \( P_m \);  
32 \( i \leftarrow i + 1 \);  
33 if \( U(\Gamma) \leq \Theta(|\Gamma| + 1) \) then  
   34 Split \( \tau_i \) into \( \tau_{i1} \) and \( \tau_{i2} \) such that \( U(\Gamma) + u_i = \Theta(|\Gamma| + 1) \);  
   35 Insert \( \tau_{i1} \) in \( \Gamma \);  
   36 Replace \( \tau_i \) in \( \Gamma \) by \( \tau_{i2} \) with \( u_{i2} = \frac{c_{i2}}{c_{i1}} \);  
   37 \( m \leftarrow m + 1 \);  
   38 Add \( P_m \) to \( \rho \);  
39 end  
40 return \( \rho \);

Algorithm 1: Packing algorithm for IRMLS
```

If so, \( \tau_i \) will be executed in a single processor. This process continues until the head passes the rear. The whole process of finding pairs in fast method traces the ordered list one time, so after the sort with time complexity of \( O(n \log n) \), we need an \( O(n) \) search to find those pairs. Algorithm 1 demonstrates the proposed packing algorithm. Line numbers 3 to 25 are the fast method to find task pairs.
B. Scheduling

Scheduling algorithm of RLMS was described in subsection III-A. In a worse-case scenario, a split-task may be consequently switched between processors \(P_p \) and \(P_{p+1} \). We noticed that this situation is usually happen. In such a situation, an IRM scheduler may not let \(\tau_s\) to continue on \(P_{p+1}\), even though it causes an overrun. We solve this issue by separating the processors containing two tasks or less, and processors that execute more than two tasks, i.e. the latter use IRM, and the former use RMLS.

As the processors’ task lists and scheduling algorithms are different, no prior condition of proofs presented in [4] are changed, namely IRM doesn’t miss any deadline for task-sets running on processors containing two tasks or less. For processors which run three tasks or more, we use RMLS, which overestimates utilization of split task using equation (5), it claims not to miss any deadline. These conditions are met in our packing algorithm, thus the new method do not miss any deadline.

V. EXPERIMENTAL RESULTS

In this section the proposed method is compared with RMLS [12]. To make random task-sets with no bias, we used UUnifast algorithm [15]. The original UUnifast generates tasks with \(u_i > 1\) and we changed the algorithm to keep only convenient task-sets. On the other hand, to compare the results of packing, we used the method offered by Burns et al [7]. Since using optimal packing method was too time consuming and was not possible for some of task-sets, we have used the fast method.

According to [7], we generated 1000 task-sets, with cardinalities of 16, 20, 28, 44, and 76, and total utilization of 8. We allowed both algorithms to assign as much processors as they need, and then calculated the overall utilization of each task-set for every algorithm according to equation (7). The median for each task-set is shown in figure 1. Using medium or average solely does not show the whole difference between two sets of results. Thus according to [7], we have used error bars to compare results for best and worst 25 percent of the test task-sets. We have more accurate inspection about two methods.

\[
Utilization = \frac{U_C}{m} \tag{7}
\]

Two another tests are applied too, one for total utilization of 16, and task-set cardinalities of 38, 48, 67, 106, and 183, and the other with task cardinalities of 6, 8, 12, 20, and 32 and total utilization of 4. Results are shown in figures 2 and 3 respectively. Error bars indicate the 25 and 75 percentiles.

Experiments show a good improvement for most of the experimented task-sets, for example for \(U = 16\) and the task-set of cardinality 38, we have an improvement about 14 percent in performance for more than 75 percent of task-sets, and for cardinality of 48, more than 50 percent of task-sets scheduled by IRMLS have an improvement about 10 percent over all of the same task-sets scheduled by RMLS.
It seems that as the number of tasks increases, both utilizations of IRMLS and RMLS decreases, whereas the downward trend in utilization of IRMLS is more and it moves toward RMLS. The reason for this problem is the way UUnifast randomly generates the utilization of each task. As the number of task grows against U, UUnifast tends to generate tasks with smaller utilization, and because of that the chance of having two tasks which meet the equation (6) decreases. This causes the performance of IRMLS tends towards RMLS.

VI. CONCLUSION

In this paper, we proposed the IRMLS, a semi-partitioned algorithm to pack and schedule sporadic task-sets with implicit deadlines on multi-core platform. Because IRM is a high performance scheduler and on the other hand, RMLS allocates the task-sets to processors with high utilization, we took advantage of IRM and RMLS to present IRMLS. We have improved the packing phase of RMLS to cooperate with IRM scheduler. We have used IRM to schedule processors running one or two tasks and RMLS to schedule processors running more than two tasks. The results show when task-set contains tasks with high utilizations, IRMLS utilizes the processors much better than RMLS and if all of tasks are small (low utilization), it works similar to RMLS. We achieve the performance of IRMLS for more than 10,000 task-sets and presented them through charts. The average utilization of RMLS is about 0.71 on experimented task-sets, and IRMLS reaches the utilization of 0.75 on the same tasks-sets. However, calculating the precise and worst-case bound is left for the future work.

REFERENCES