An Efficient Real-Time Multiprocessor Scheduling Algorithm

Hitham Alhussian, Nordin Zakaria, Fawnizu Azmadi Hussin

IT Department, Universiti Teknologi PETRONAS
Bandar Seri-Iskandar, 31750 Tronoh, Malaysia
alhussain_g01789@utp.edu.my, halhussian@gmail.com
NordinZakaria@petronas.com.my

EE Department, Universiti Teknologi PETRONAS
Bandar Seri-Iskandar, 31750 Tronoh, Malaysia
Fawnizu@petronas.com.my

Abstract

Most currently existing optimal real-time multiprocessor scheduling algorithms follow the fairness rule, in which all tasks are forced to make progress in their executions proportional to their utilization, to ensure the optimality of the algorithm. However, obeying the fairness rule results in large number of task preemptions and migrations and these highly affect the practicability of the algorithm. In this paper, we present an efficient real-time multiprocessor scheduling algorithm in which the fairness rule is completely relaxed and a semi-greedy algorithm is introduced. In the simulation, the proposed algorithm showed promising results in terms of number of task preemptions and migrations that are very few compared to the current state of the art real-time multiprocessor scheduling algorithms. Although the algorithm can sometimes miss a very few deadlines, we assume that these deadline misses can be tolerated in view of the great reduction of task preemptions and migrations.

Keywords: Real-time systems, Multiprocessor, Scheduling

1. Introduction

Real-time systems are systems that maintain their correctness by producing output results within specific time constraints called deadlines [1]. Meeting the deadlines of a given real-time task set cannot be achieved without the use of an optimal scheduling algorithm unless some constraints are imposed. An optimal scheduling algorithm is defined in [2] as "one which may fail to meet a deadline only if no other one can", i.e. it will successfully schedules all tasks without missing any deadlines for any schedulable task set [3, 4].

Recently, many global optimal scheduling algorithms have been proposed and have the ability to achieve higher utilization even equal to m, such as P-fair, LLREF, LRE-TL, DP-Wrap [4-8]. However most of them follow the fairness rule, in which all tasks make progress in their executions proportional to their utilization. Although the fairness rule ensures the optimality, it generates high scheduling overhead in-terms of tasks preemptions as well as migrations [3, 9]. Due to the high scheduling overhead that accompany the enforcement of the fairness rule, such algorithms may not work well when actually implemented since the processors will be more busy executing the scheduler than executing the actual work [3].

The empirical study that has been conducted in [10] showed that preemptions and migrations delays could reach up to 1 ms on a multiprocessor system that contains 24 cores running at 2.13 GHz with three levels of cache memory. Therefore a real-time multiprocessor scheduling algorithm should also consider the reduction of scheduling overhead in order to be implemented practically.

The goal of this paper is to present an efficient global real-time multiprocessor scheduling algorithm that greatly reduces the scheduling overhead, in terms of task preemptions as well as migrations compared to those that conform to fairness rule. However, since the fairness rule is completely relaxed, sometimes the algorithm may miss a very few deadlines few enough to be tolerated in view of the great reduction in task preemptions and migrations.

The rest of this paper is organized as follows: In Section 2 we describe the task model and define the terms that will be used in this paper. Section 3 reviews related works. Section 4
2. Model and terms definition

This paper considers the scheduling of n periodic tasks with implicit deadlines on a platform of m Symmetric Shared-Memory multiprocessor (SMP). A periodic task, in real-time systems, is a task that is periodically released at a constant rate. Usually, two parameters are used to describe a periodic task \( T_i \); its worst-case execution time \( e_i \) as well as its period \( p_i \). An instance of a periodic task (i.e., release) is known as a job and is denoted as \( T_{ij} = (e_{ij}, p_{ij}) \) where \( j = 1, 2, 3, \ldots \), and \( e_{ij} \) denotes the worst-case execution requirement of job \( T_{ij} \) where \( p_{ij} \) denotes its period. The deadline of a job is the arrival time of its successor. For example the deadline of the job of \( T_{ij} \), would be the arrival time of job \( T_{ij+1} \), that is at \((j + 1)p_i\). The laxity of a job \( T_{ij} \) at time \( t \), denoted \( l_{ij,t} \), is the time that \( T_{ij} \) can remain idle before its execution should be started, i.e. \( l_{ij,t} = p_i - e_{ij,t} - t \), where \( e_{ij,t} \) denotes the remaining execution of job \( T_{ij} \) at time \( t \). One more important parameter that is used to describe a task \( T_i \) is its utilization and is denoted as \( u_i = \frac{e_i}{p_i} \).

The utilization of a task is the portion of time that it needs to execute after it has been released and before it reached its deadline. \( U_{\text{am}} \) denotes the total utilization of a given task set \( T \) whereas \( U_{m,as} \) describes its maximum utilization. A periodic task set \( T \) is said to be schedulable on \( m \) identical multiprocessor iff \( U_{\text{am}}(T) \leq m \) and \( U_{m,as}(T) \leq 1 \).

3. Related work

The Proportionate Fair (P-fair) [7] was the first optimal real-time multiprocessor scheduling algorithm. P-fair is defined for periodic tasks with implicit deadlines. P-fair executes tasks proportionate to their utilization, based on the idea of fluid scheduling, by dividing the timeline into equal length quanta. The algorithm allocates tasks to processors at every time quanta \( t \), such that the accumulated processor time allocated to each task \( t_i \) will be either \( \lceil u_i \rceil \) or \( \lfloor u_i \rfloor \), P-fair can achieve optimal utilization bound \( U \leq m \) [7]. However, P-fair is difficult to be implemented in practice since it makes scheduling decisions at each time quanta, which in turn, results in a huge amount of tasks preemptions and migrations. Many versions of P-fair algorithm were designed over the time (e.g., PD [11], PD2 [12], ER-PD [13]). However, all of them suffer from the huge amount of preemptions and migrations since all of them also ensure the fairness property at each time quanta.

Instead of ensuring the fairness rule at each time quanta \( t \), a new technique known as Deadline Partitioning (DP) is introduced in which time is partitioned into slices [5, 14]. The end of each time slice corresponds to the deadline of one of the tasks in the system. Within each time slice, all tasks in the system are scheduled to execute part of their work proportional to their utilization before the end of the time slice. This means that all tasks in each time slice share the same deadline. By ensuring the fairness rule at the end of each time slice, the overhead of preemptions and migrations is reduced significantly compared to P-fair algorithms. Based on DP, a family of algorithms known as the Deadline Partitioning Fair (DP-Fair) [5] or the Boundary Fair [14] algorithms have been proposed, such as BF [14], LLREF [6], LRE-TL [4], and DPWrap [5].

EKG [15] algorithm uses a parameter \( k \), \( 1 \leq k \leq m \) to split the tasks into \( k \) group of processors. When \( k=1 \) EKG uses partitioned Earliest Deadline First (EDF) to schedule tasks with limited utilization bound. When \( k=m \) EKG schedule tasks in a way very similar to DP-Wrap with utilization equal to \( m \). The authors in [16] presented an improvement to EKG to reduce the number of preemptions and migrations that it generates by decreasing the number of time slices needed to ensure that deadlines are met on one side, and by using a swapping algorithm to exchange execution time between tasks and time slices on the other side.

In [17] a new optimal algorithm hold the name RUN, Reduction to Uniprocessor, has been presented. RUN introduces a new approach to multiprocessor scheduling in which the scheduling problem is reduced to a series of uniprocessor ones using a dualization technique. Whenever a proper partitioning is found, RUN reduces to Partitioned EDF. It has been shown
that in [17] RUN achieves better performance than existing optimal algorithms in terms of
task’s preemptions with no more than 3 preemptions per job; however it cannot handle
scheduling of sporadic tasks.

In [3], a conclusion has been drawn from the study of the above algorithms: that whenever
the fairness rule is relaxed, the number of preemptions as well as migrations decreases. This
conclusion can be clearly observed in P-fair algorithms which produce much more scheduling
overheads than BF or DP-Fair which in turn produces more scheduling overheads than EKG.
According to this observation, authors of [3] have proposed U-EDF, Un-fair-Earliest Deadline
First, algorithm in which they have released the fairness property and incorporated EDF rules in
the scheduler. As a result, their algorithm shows a great performance in terms of number of
preemptions as well as migrations. However, their pre-allocation algorithm has a run-time
complexity of O(n×m) which seems to be quite high [3].

To the best of our knowledge both RUN and U-EDF outperform all other existing algorithms
[3, 17]. However, we will show that our new algorithm achieves better performance in terms of
number of preemptions as well as migrations.

4. The proposed algorithm

Our proposed algorithm is inspired from LRE-TL which is based on the concept of fluid
scheduling [4]. However; the difference in our approach compared to LRE-TL is that we have
completely relaxed the fairness rule. However, relaxing the fairness rule leads to greedy
schedule which fails to schedule some task sets as explained in [5]. To reduce the side effect of
completely relaxing the fairness rule as well as alleviate the problem of greedy schedulers, we
have proposed a semi greedy algorithm named USG for Un-fair Semi Greedy. A greedy
scheduler is one in which the execution is given to a job according to specific priority, for
example according to job’s deadline or laxity. EDF and Least Laxity First (LLF) are well known
elements of greedy scheduler. On the other hand, a semi greedy scheduler is one in which the
priority rule can sometimes be violated or loosely forced. In the example that appears later in
this section we show the difference between our algorithm and greedy schedulers.

In both LLREF and LRE-TL the fairness rule is always ensured at the deadline of tasks; i.e.
they divide the time into time slices which are bounded by two successive deadlines, and the
end of each time slice corresponds to the deadline of one of the tasks in the system. For example
in the task set shown in Table 1 below [4, 6]; although tasks T₂ and T₅ have worst-case
execution requirements and periods (1, 16) and (2, 26) respectively, they are forced to make
progress in their execution in each TL-plane even though they can wait for 15 and 24 units of
time respectively before they became critical. This means that task T₁ will be preempted 6 times
before it reach its deadline and similarly task T₃ which is preempted 11 times before it reach its
deadline.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>e</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>T₂</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>T₃</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>T₄</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>T₅</td>
<td>2</td>
<td>26</td>
</tr>
<tr>
<td>T₆</td>
<td>15</td>
<td>26</td>
</tr>
<tr>
<td>T₇</td>
<td>20</td>
<td>29</td>
</tr>
<tr>
<td>T₈</td>
<td>14</td>
<td>17</td>
</tr>
</tbody>
</table>

In our new approach, we propose three states for each task, as depicted in Figure 1
below, Running, Waiting and Not Active. The tasks traverse between the states with the
advance of time.
A task’s state changes from one state to another as a result of a scheduling event. We propose to use three scheduling events: The End (E) event, the Zero Laxity (Z) event, and the Active (A) event.

4.1 The End (E) Event

The End (E) event is fired when a task finishes its execution i.e. when it completes all its execution units (its $e_i$), and as a result the task state is changed from “Running” to “Not Active” and it is removed from the running list and added to the “Not Active” list. In this case the task with the least laxity is selected from the waiting list to replace the task just finished and accordingly, its state is changed from “Waiting” to “Running” before it is added to the running list.

4.2 The Zero Laxity (Z) Event

The Zero Laxity (Z) event is fired when a task reaches zero laxity meaning that it can’t wait any more and it has to be instantly selected for execution; otherwise it will miss its deadline. In this case, the task with the minimum remaining execution in the running list will be preempted and added to the waiting list to let the critical task resume its execution.

4.3 The Active (A) Event

The Active (A) event is fired whenever a task is released i.e. it became an active. If there is an idle processor, the task is immediately selected for execution and assigned to that processor. If no processor is idle, then a choice will be made according to the task's laxity. If the task's laxity is equal to 0, then it will be directly scheduled for execution by preempting the task with the minimum remaining execution time from the running list which in its turn will be added to the waiting list until it becomes critical again. If the task laxity is greater than 0; then it will be added to the waiting list until it becomes critical.

We propose to use the heap data structure as in [4] to implement our algorithm. A heap is a data structure which maintains the minimum item (key) in its root node. The algorithm contains three heaps. The first heap is $H_R$ which is used to implement the “Running” tasks list. The second heap is $H_W$ and it is used to implement the “Waiting” tasks list. The third one is $H_{NA}$ which is used to implement the “Non Active” tasks list.

The algorithm contains four procedures. The main procedure, the initialize procedure, the handleEorZEvents procedure and the handleAEvent procedure.
4.4 The main Procedure

The main procedure, as shown in Figure 2 (a) below, starts with initializing the system by calling the initialize procedure (line 1). Unlike LRE-TL, which calls the initialize procedure at the end of each TL-plane, in our algorithm this procedure is called once when the system is started to initialize the heaps. After the initialize procedure ends, the system is started by checking for all types of events as the time advanced (lines 3-6). Firstly the system checks for both $E$ or $Z$ events (line 3) and whenever one of them occurs, the system respond by calling the handleEorZEvent procedure (line 4). After that, the system checks for the $A$ event (line 5), and if a task became an active i.e. it is released, the system respond by calling the handleAEvent procedure (line 6). After that, all processors are instructed to perform their assigned jobs (line 7), and the system waits until the next time an event will be fired (line 8).

```
1. call initialize()
2. while true
3.  if a E or Z event occurs
4.    call handleEorZEvent()
5.  if an A event occurs
6.    call handleAEvent()
7.    instructs each processor to execute its job
8.    sleep until next time an event is fired
```

Figure 2. (a) The main procedure

4.5 The initialize Procedure

In the initialize procedure, which is shown in Figure 2 (b) below, all tasks are added to a priority queue ($pq$) before they are assigned to the processors (lines 2-4). The priority queue is assumed to be sorted according to task’s laxity. A task with least laxity will have a higher priority. Then we extract the first $m$ tasks from the priority queue and add them to heap $H_R$ after updating their keys to the next time when they fire an $E$ event i.e. they finish (lines 6-12). All other remaining tasks will be added to heap $H_W$ with their keys set to the next time they become critical (lines 14-15). This will ensures that the tasks selected for execution firstly are the most critical ones. Such tasks will continue to execute until either they finish or another task(s) from heap $H_R$ became critical (reaches zero laxity), as we will see later in the handleEorZEvent Procedure. In lines (17-18) remaining processors, if any, are set to null.

```
1.  Tcur = 0;
2.  for each task t in T
3.    t.key = t.p - t.e;
4.    pq.offer(t);
5.  int z = 0;
6.  while (! pq.empty())
7.    t = pq.pull();
8.    if (z==P.length)
9.      t.key = t.p - t.key + Tcur;
10.     t.procId = z;
11.     P[z].taskId = t.id;
12.     heapR.insert(t);
13.     z = z + 1;
14.  else
15.    heapW.insert(t);
16.  //Null all remaining processors
17.  for i=z to P.length
18.    P[i].taskId = 0;
```

Figure 2. (b) The initialize procedure
The handleEorZEvents Procedure

The handleEorZEvents procedure, which is shown in Figure 2 (c) below, is called whenever an E or Z event occurs. The procedure starts firstly by handling any E event(s) line (2-18). In (line 3), the completed task is extracted from heap $H_R$, and if there is available tasks on heap $H_W$ then the task with minimum laxity will be extracted, its key is updated the next time when it will fire an E event, assigned to the same processor of the completed task, and inserted to heap $H_R$ (lines 5-10). If no tasks are available in heap $H_W$ (line 13), then the processor occupied by the completed task will be set to idle (line 14). The completed task will then be added to the not active heap $H_{NA}$ after updating its key to the next time when it will be released again (lines 14).

After handling all E events, the procedure then continues to handle all Z events (lines 17-27). In (line 18) the procedure checks heap $H_W$ for any tasks whose laxity became zero. If such task exists, then it will preempt the task with the minimum remaining execution time from heap $H_R$ after updating its key to the next time when it will fire an E event (lines 20, 22-26). The preempted task from heap $H_R$ will be added to heap $H_W$ after setting its key to the next time when it will fire a Z event (lines 19, 21, 27).

```
1. //Handle E Event
2. while (heapR.getMinimum().key == Tcur)
3. Task Tr = heapR.extractMinimum();
4. int z = Tr.procId;
5. if (heapW.getSize()>0)
6. Task Tw = heapW.extractMinimum();
7. Tw.key = Tw.p - Tw.key%Tw.p + Tcur;
8. Tw.procId = z;
9. P[z].taskId = Tw.id;
10. heapR.insert(Tw);
11. Else
12. P[z].taskId = 0;
13. //Add Tr to Not Active heap
14. Tr.key = (Tr.key%Tr.p)? Tcur: (Tr.p - Tr.key%Tr.p) + Tcur;
15. heapNA.insert(Tr);
16. //Handle Z Event
17. if (heapW.getMinimum().key==Tcur)
18. while (heapW.getMinimum().key==Tcur)
19. Task Tr = heapR.extractMinimum();
20. Task Tw = heapW.extractMinimum();
21. Tr.key = Tr.p - Tr.key%Tr.p + Tcur;//change to laxity
22. Tw.key = Tw.p - Tw.key%Tw.p + Tcur;//change to key
23. int z = Tr.procId;
24. Tw.procId = z;
25. P[z].taskId = Tw.id;
26. heapR.insert(Tw);
27. heapW.insert(Tr);
```

Figure 2. (c) The handleEorZEvent procedure

4.7 The handleAEvent Procedure

The handleAEvent procedure, Figure 2 (d), is fired whenever a task is released i.e. it became an active. A task becomes an active when its key is equal to the current time. The procedure starts by checking for any active tasks. If a task became an active then it is extracted from the not active heap $H_{NA}$ (line 3). The procedure firstly checks to see if any idle processor is available, and if so, then the task will be added to heap $H_R$ and assigned to the idle processor after updating its key to the next time when it will fire an E event (lines 7-17). If no idle processor is available (line 18), the procedure checks to see if the task is critical, i.e. has a zero
laxity, or not (line 20). If it is not critical, then it will be added to heap $H_R$ after updating its key to the next time when it will fire a Z event (lines 21-23). If it is critical, then it will preempt the task with the minimum remaining execution from heap $H_R$ (lines 24-27, 29-32). The preempted task will be added to heap $H_R$ after updating its key to the next time when it will fire a Z event (lines 28, 33).

```
1. //Handle A event
2. while (heapNA.getMinimum().key == Tcur)
3. Task Tna = heapNA. extractMinimum (), Tr;
4. int z;
5. //if heapR size is less than m then
6. // add Tna to heapR
7. if (heapR.getSize()<m)
8. Tna.key = Tna.e + Tcur;
9. //get any idle processor
10. int idleProcessor=0;
11. for (int i=0; i<P.length; i++)
12. if (P[i].taskId==0)
13. idleProcessor = i;
14. break;
15. Tna.procId = idleProcessor;
16. P[idleProcessor].taskId = Tna.id;
17. heapR.insert(Tna);
18. else  //HeapR is full
19. //check to see if Tna should go for heapW
20. //or preempt one for it
21. if (Tna.e<Tna.p)
22. //add Tna to heapW
23. Tna.key = Tna.p-Tna.e + Tcur;
24. heapW.insert(Tna);
25. else//preempt one for Tna
26. Tr = heapR.extractMinimum();
27. Tna.key = Tna.e + Tcur;
28. //change Tr.key to laxity
29. Tr.key = Tr.p - (Tr.key%Tr.p)+ Tcur;
30. z = Tr.procId;
31. Tna.procId = z;
32. P[z].taskId = Tna.id;
33. heapR.insert(Tna);
34. heapW.insert(Tb);
```

Figure 2. (d) The `handleAEvent` procedure

4.8 Example

In this example, we show how USG schedules the feasible task set \{ $T_1=(9, 10)$, $T_2=(9, 10)$, $T_3=(7, 40)$ \} in which greedy schedulers, such as LLF, fail to schedule on two processors. LLF fails to schedule the above mentioned example because task $T_3$ can only take the processor in the intervals [9-10], [19-20], [29-30] before it can be prioritized over $T_1$ and $T_2$ at time $t=36$ in which it will have zero laxity. Therefore $T_3$ will be executed to competition before its deadline at $t=40$. On the other hand, both $T_1$ and $T_2$ still have 3 units remaining per each i.e. total of 6 units however only 3 units of time are remaining on the other processor before their deadline at $t=40$. Thus one of $T_1$ or $T_2$ will miss its deadline. The schedule generated by LLF is depicted in Figure 3 (a) below. In the following scenario we explain how USG can successfully schedule the above task set. As it is explained in the algorithm above, when the scheduler starts it will firstly call the `initialize` procedure. The `initialize` procedure will then populate heap $H_R$ with tasks $T_1$ as well as $T_2$ since they have the least laxity. Task $T_3$ will be added to heap $H_W$. When the `initialize`
procedure finishes execution, the system is started by checking for any type of the three events $E$, $Z$, or $A$. As we can see from the schedule depicted in Figure 3 (b) below, $T_1$ and $T_2$ continue to execute until time $t=9$ then both of them will fire $E$ event. As a result $T_3$ will be selected for execution on one processor and the other one will be idle. $T_3$ will continue to execute until time $t=10$ at which both of $T_1$ and $T_2$ will arrive. So $T_1$ will be assigned to the idle processor, whoever since $T_2$ is not critical, i.e. its laxity is not zero, then it will be added to heap $H_W$ even though it has least laxity than $T_3$. Thus $T_3$ would have a chance to make progress in its execution one more unit of time before $T_2$ reaches zero laxity and preempt it. Hence we have called our algorithm Semi Greedy. The scenario continues as depicted in Fig. 3 (b) and a feasible schedule is generated.

![Figure 3. (a) Schedule generated using LLF for the task set given in the example, at time $t=38$ the three tasks reaches zero laxity and if $T_1$ and $T_3$ are allowed to execute then $T_2$ will miss its deadline.](image)

![Figure 3. (b) Schedule generated using USG for the task set given in the example all the three tasks $T_1$, $T_2$, and $T_3$ meet their deadlines.](image)

5. Results and Discussion

To test our proposed algorithm, we have conducted an extensive experimental work. We have generated random task sets of 4, 8, 16, 32 and 64 tasks using a uniform integer distribution. The generated tasks sets are executed on 2, 4, 8, 16 and 32 processors respectively, i.e., the number of tasks is double the number of processors $(n=2m)$. For each task set, we have generated 1000 samples with full utilization $\sum u_i = m$. The results that we obtained were encouraging. For example, the average number of migrations per job that we got for 2 processors is only 0.24 which is better than the result generated by the best known algorithms RUN and U-EDF. This can be clearly seen from Figure 4 as well as Figure 5, in which we compare the results of migrations as well as preemptions generated by our algorithm against the results generated by the best known algorithms in this field. In Figure 6 we show the average of deadline misses per task. It can be clearly seen from Figure 6 that the number of deadlines misses are very few that can be tolerated considering the great reduction of task preemptions and migrations. Figure 7 and Figure 8 also show the percentages of scheduled and unscheduled task sets out of the 1000 task sets for USG compared to the non-optimal multiprocessor algorithms LLF and Global-EDF (G-EDF) [18]. According to these results, we strongly believe that our algorithm is suitable to be implemented in practical real-time systems that can tolerate few deadline misses.
Figure 4. Comparison of migrations per job between USG algorithm and the best known algorithms

Figure 5. Comparison of preemptions per job between USG algorithm and the best known algorithms
Figure 6. Average deadline misses per task for USG and G-EDF

Figure 7. Percentage of scheduled task sets for USG, LLF and G-EDF

Figure 8. Percentage of unscheduled task sets for USG, LLF and G-EDF
6. Conclusion

In this paper, we presented an efficient real-time multiprocessor scheduling algorithm. The algorithm has been designed by completely relaxing the fairness rule to avoid the huge number of preemptions and migrations that it generates. Although the algorithm can sometimes miss a very few deadlines, we assume that these deadline misses can be tolerated since the algorithm produces very few number of tasks migrations as well as preemptions. The algorithm uses a global job queue ordered by an increasing laxity. Tasks with zero laxity have higher priority and they are always scheduled for execution immediately. The experimental work that has been conducted showed a few number of task preemptions and migrations compared to the best known algorithms in this field. In the future work, we will continue to improve our algorithm in terms of optimality as well as performance.

7. References