

1-1-2017

Minimizing scheduling overhead in LRE-TL real-time multiprocessor scheduling algorithm

HITHAM SEDDIG ALHASSAN ALHUSSIAN

MOHAMED NORDIN BIN ZAKARIA

FAWNIZU AZMADI BIN HUSSIN

Follow this and additional works at: <https://journals.tubitak.gov.tr/elektrik>



Part of the [Computer Engineering Commons](#), [Computer Sciences Commons](#), and the [Electrical and Computer Engineering Commons](#)

Recommended Citation

ALHUSSIAN, HITHAM SEDDIG ALHASSAN; ZAKARIA, MOHAMED NORDIN BIN; and HUSSIN, FAWNIZU AZMADI BIN (2017) "Minimizing scheduling overhead in LRE-TL real-time multiprocessor scheduling algorithm," *Turkish Journal of Electrical Engineering and Computer Sciences*: Vol. 25: No. 1, Article 22. <https://doi.org/10.3906/elk-1403-273>

Available at: <https://journals.tubitak.gov.tr/elektrik/vol25/iss1/22>

This Article is brought to you for free and open access by TÜBİTAK Academic Journals. It has been accepted for inclusion in Turkish Journal of Electrical Engineering and Computer Sciences by an authorized editor of TÜBİTAK Academic Journals. For more information, please contact academic.publications@tubitak.gov.tr.

Minimizing scheduling overhead in LRE-TL real-time multiprocessor scheduling algorithm

Hitham Seddig Alhassan ALHUSSIAN*, Mohamed Nordin Bin ZAKARIA,
Fawnizu Azmadi Bin HUSSIN

Universiti Teknologi Petronas, Bandar Seri-Iskandar, Tronoh, Malaysia

Received: 31.03.2014

Accepted/Published Online: 17.12.2015

Final Version: 24.01.2017

Abstract: In this paper, we present a modification of the local remaining execution-time and local time domain (LRE-TL) real-time multiprocessor scheduling algorithm, aimed at reducing the scheduling overhead in terms of task migrations. LRE-TL achieves optimality by employing the fairness rule at the end of each time slice in a fluid schedule model. LRE-TL makes scheduling decisions using two scheduling events. The bottom (B) event, which occurs when a task consumes its local utilization, has to be preempted in order to resume the execution of another task, if any, or to idle the processor if none exist. The critical (C) event occurs when a task consumes its local laxity, which means that the task cannot wait anymore and has to be scheduled for execution immediately or otherwise it will miss its deadline. Event C always results in a task migration. We have modified the initialization procedure of LRE-TL to make sure that tasks that have higher probability of firing a C event will always be considered for execution first. This will ensure that the number of C events will always be at a minimum, thereby reducing the number of task migrations.

Key words: Real-time, multiprocessor, scheduling, migration, preemption

1. Introduction

In real-time systems the correctness of the system does not depend on only the logical results produced, but also on the physical time at which these results are produced [1–5]. Meeting the deadlines of a real-time task set in a real-time multiprocessor system requires the use of an optimal scheduling algorithm. A scheduling algorithm is said to be optimal if it successfully schedules all tasks in the system without missing any deadline provided that a feasible schedule exists for the tasks [3,5–8]. The scheduling algorithm decides which processor the task will be executed on, as well as the order of the tasks' execution. Although a scheduling algorithm may be optimal, sometimes it cannot be applied practically [9]. This is because of the scheduling overheads, in terms of task preemptions and migrations that accompany its work. These overheads can potentially be very high, especially when considering the hardware architecture. The fact that jobs can migrate from one processor to another can result in additional communication loads and cache misses, leading to increased worst-case execution times [10–12].

In this paper, we consider the possibility of reducing the scheduling overhead incurred by task migrations in the LRE-TL algorithm, which uses deadline partitioning, or a time slices technique, in order to generate a successful schedule if a possible one exists. The idea behind our work to reduce task migrations in this algorithm is considering the tasks with largest local remaining execution first (LLREF) or least laxity first (LLF) when

*Correspondence: halhussian@gmail.com

initializing the TL-plane, i.e. at the beginning of each time slice. We have realized through extensive experiments that such tasks, when not considered for execution first, would likely fire a critical (C) event, which in turn would result in task migration.

The rest of this paper is organized as follows: Section 2 describes the task model and defines the terms that will be used in this paper. Section 3 gives an overview of real-time multiprocessor scheduling as well as reviewing some related algorithms. In Section 4 we show how to reduce task migrations in LRE-TL. In Sections 5 and 6 we present and discuss the experimental results, and lastly we conclude with Section 7.

2. Model and term definitions

In real-time systems, a periodic task [13] is one that is released at a constant rate. A periodic task T_i is usually described by two parameters: its worst-case execution requirement e_i and its period p_i . The release of a periodic task is called a job. Each job of T_i is described as $T_{ik} = (e_i, p_i)$ where $k=1, 2, 3, \dots$. The deadline of the k th job of T_i , i.e. T_{ik} , is the arrival time of job $T_{i(k+1)}$, i.e. at $(k+1)p_i$. A task's utilization is one of the important parameters and is described as $u_i = e_i/p_i$. A task's utilization is defined as the portion of time that the task needs to execute after it is released and before it reaches its deadline. The total as well as the maximum utilization of a task set T are described as U_{sum} and U_{max} , respectively. A periodic task set is schedulable on m identical multiprocessors iff $U_{sum} \leq m$ and $U_{max} \leq 1$ [14].

3. Literature review

Scheduling on real-time multiprocessor systems can be classified into three categories: partitioning, global, and cluster scheduling. Due to the large deficiencies of partitioning as well as cluster approaches, there has been much interest in recent years in global schedulers. This is because in global scheduling, tasks are allowed to migrate between processors. Hence, they are able to achieve the highest processor utilizations. Unfortunately, uniprocessor scheduling algorithms cannot be used here since they produce low processor utilization. Hence, recently there has been much interest in designing new global algorithms that are not extended from their uniprocessor counter parts, particularly in global optimal scheduling algorithms. In [15] Baruah et al. introduced the *Pfair* algorithm, the first optimal multiprocessor scheduling algorithm for the periodic real-time task model with implicit deadlines. As a global scheduler able to migrate tasks between processors, *Pfair* can successfully schedule any task set whose execution requirement does not exceed processor capacity. However, recently, a number of proposed algorithms have exploited the concept of deadline partitioning (dividing the time into time slices wherein all tasks share the same deadline) to achieve optimality while greatly reducing the number of required preemptions and migrations, such as in LLREF [16] and LRE-TL [14].

3.1. Largest local remaining execution first (LLREF) algorithm

LLREF is a real-time multiprocessor scheduling algorithm based on the fluid scheduling model, in which all tasks are executed at a constant rate. LLREF divides the schedule into time and local execution time planes (TL-planes or time slices), which are determined by task deadlines. The algorithm schedules tasks by creating smaller "local" jobs within each TL-plane. The only parameters considered by the algorithm during a TL-plane are the parameters of the local jobs. When a TL-plane completes, the next TL-plane is started. The duration of each TL-plane is the amount of time between consecutive deadlines [16]. For example, consider the task set in Table 1.

Table 1. Sample task set 1.

T_i	e_i	p_i
T ₁	3	7
T ₂	5	11
T ₃	8	17

In this case, the intervals of the TL-planes will be as shown in Table 2 below.

Table 2. TL-plane intervals for the task set in Table 1.

TL-plane	Interval
TL-0	[0, 7)
TL-1	[7, 11)
TL-2	[11, 14)
TL-3	[14, 17)
TL-4	[17, 21)
TL-5	[21, 22)
.	.
.	.

Within each TL-plane, the local execution is calculated for all tasks. For example, if t_{f_0} and t_{f_1} are the starting and ending times of a TL-plane, then T_i 's local execution is calculated using Eq. (1).

$$l_{i,0} = u_i(t_{f_1} - t_{f_0}) \quad (1)$$

Recall that $u_i = \frac{e_i}{p_i}$ as mentioned previously in Section 2. This means that the local execution of each task is proportional to its utilization.

For example, given the task set in Table 1, then the local executions of the first three tasks on the first TL-plane [0, 7) are calculated as follows.

$$\text{local execution for task } T_1 : l_{1,0} = u_i(t_{f_1} - t_{f_0}) = \frac{3}{7} \times (7 - 0) = 3.0$$

$$\text{local execution for task } T_2 : l_{2,0} = u_i(t_{f_1} - t_{f_0}) = \frac{5}{11} \times (7 - 0) = 3.2$$

$$\text{local execution for task } T_3 : l_{3,0} = u_i(t_{f_1} - t_{f_0}) = \frac{8}{17} \times (7 - 0) = 3.3$$

If task T_i starts its execution at time t_x then its local remaining execution $l_{i,x}$ starts to decrease. Whenever a scheduling event occurs, LLREF selects the m highest remaining execution tasks for execution. The selected tasks will continue to execute until one of the following events occur [16].

- *Event B*: the bottom (B) event occurs when a task completes its local remaining execution (i.e. when $l_{i,x} = 0$) [16].
- *Event C*: the critical (C) event occurs when a task consumes its local laxity and cannot wait anymore; therefore, it must be selected directly for execution or else it will miss its deadline (i.e. $l_{i,x} = u_i \times (t_{f_1} - t_{f_x})$) [16].

Figure 1 shows both B and C events.

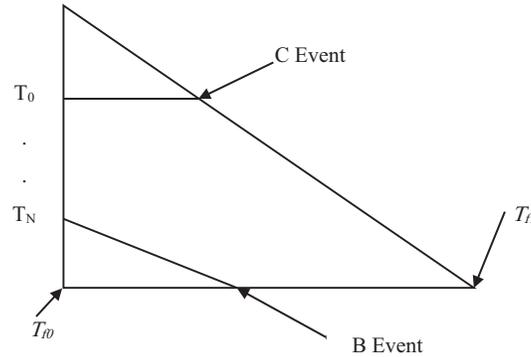


Figure 1. The bottom (B) and the critical (C) events.

LLREF continues to execute until all tasks within the TL-plane complete their local remaining execution [16], and then the next TL-plane is initialized and the process is repeated.

3.2. Local remaining execution-time and local time domain (LRE-TL) algorithm

LLREF introduces high overhead in terms of running time as well as preemptions and migrations [14]. LRE-TL is a modification of LLREF. The key idea of LRE-TL is that there is no need to select tasks for execution based on the largest local remaining execution time when a scheduling event occurs. In fact, any task with remaining local execution time will do. This idea greatly reduces the number of migrations within each TL-plane compared to LLREF. Moreover, LRE-TL is extended to support scheduling of sporadic tasks with implicit deadlines while achieving a utilization bound of m [14].

The LRE-TL algorithm contains four procedures. The main procedure starts by calling the TL-plane initializer procedure at each TL-plane boundary. Then it checks for each type of scheduling event and calls the respective handler when an event occurs. After that, the main procedure instructs the processors to execute their designated tasks [14].

The TL-plane initializer, which is called at each TL-plane boundary, sets all parameters for the new TL-plane. The A event handler determines the local remaining execution of a newly arrived sporadic task. The B and C event handler handles the bottom (B) events as well as the critical (C) events [14].

LRE-TL maintains three heaps, the deadline heap H_D , the B event heap H_B , and the C event heap H_C . The algorithm starts by first initializing the TL-plane, in which the deadline heap will be populated with tasks that arrived at time T_{cur} , and then the algorithm starts adding tasks to be scheduled for execution on heap H_B until all processors are occupied. After that, all remaining tasks will be added to heap H_C . For tasks added to heap H_B and H_C , their keys are set to the time at which the task will trigger a scheduling event [14].

The LRE-TL algorithm will not preempt a task unless it is absolutely necessary. When a B event occurs, the task that generated the B event will be preempted and replaced by the minimum of heap H_C , the closest task to fire a C event. All tasks that were executing prior to the B event will continue to execute (on the same processor) after the B event is handled, unlike LLREF, which sorts the tasks according to their LLREF and selects the m largest ones to be scheduled for executions. On the other hand, when a C event occurs, the task that fired the C event should be immediately scheduled for execution. This is done by preempting the minimum of heap H_B and replacing it with the task that fired the C event. The preempted task, in turn, will be added to heap H_C [14].

4. Reducing task migrations

As discussed previously, the overheads incurred by global scheduling can potentially be very high, especially when considering the hardware architecture. The fact that jobs can migrate from one processor to another can result in additional communication loads and cache misses, leading to increased worst-case execution times [10]. We mentioned before that LRE-TL starts execution by first initializing the TL-plane, wherein the deadline heap H_D is updated with the deadline of tasks that arrived at time T_{cur} . Then heaps H_B and H_C are populated with tasks selected for execution and tasks that will remain idle until they consume their local laxity, respectively. We realized that if we first sort the tasks with LLREF, before populating heaps H_B and H_C , a significant reduction of event C , which results in task migration, is noticed. The following example clearly explains this.

4.1. Example

In this example we clearly show the effect of sorting tasks with LLREF on the reduction of C events, which result in task migrations.

4.1.1. Case 1: Scheduling without sorting tasks

Table 3 shows 8 tasks with their worst-case execution requirement e_i , period p_i , and local remaining execution l_i for the first TL-plane, which has the interval $[0, 10)$.

As we mentioned previously, LRE-TL does not consider tasks' laxity when selecting them for executions. Hence, the first 4 tasks T_1 , T_2 , T_3 , and T_4 will be selected for executions and added to heap H_B . Remember that for tasks selected for executions, the value of $l_{i,t}$ denotes the time at which the task will end execution. All remaining tasks are instructed to wait until they consume their laxity if none of the running tasks finish execution. In this case, the laxity of each task is calculated before it is added to heap H_C as follows.

Table 3. Sample task set 2.

T_i	e_i	p_i	$l_{i,0} [0, 10) = \frac{e_i}{p_i} \times (10 - 0)$
T_1	8	17	4.7
T_2	10	30	3.3
T_3	5	11	4.5
T_4	8	29	2.8
T_5	1	10	1.0
T_6	11	13	8.5
T_7	3	26	1.2
T_8	15	18	8.3

If we would like to schedule the tasks on a system of 4 processors, then both heaps H_B and H_C would be initialized, as shown in Table 4.

$$\text{laxity of task } T_5 := (t_{f_1} - l_{5,0}) = 10 - 1.0 = 9.0$$

$$\text{laxity of task } T_6 := (t_{f_1} - l_{6,0}) = 10 - 8.5 = 1.5$$

$$\text{laxity of task } T_7 := (t_{f_1} - l_{7,0}) = 10 - 1.2 = 8.8$$

$$\text{laxity of task } T_8 := (t_{f_1} - l_{8,0}) = 10 - 8.3 = 1.7$$

Table 4 illustrates the initialization of heaps H_B and H_C for the first TL-plane.

Table 4. Initialization of H_B and H_C for the first TL-plane in the period $[0, 10)$.

TL-plane 0 $[0, 10)$			
Heap H_B			
T4	T2	T3	T1
2.8	3.3	4.5	4.7
Heap H_C			
T6	T8	T7	T5
1.5	1.7	8.8	9.0

It can be clearly seen from Table 4 that two tasks will fire C events at time $T_{cur} = 1.5$ and $T_{cur} = 1.7$, respectively. The first C event will be fired by task T_6 and will result in the preemption and migration of task T_4 . In this case, task T_6 , which has local remaining execution 8.5, will start execution from time $T_{cur} = 1.5$ and hence will end execution exactly at the end of the first TL-plane at time $T_{cur} = 10$. On the other hand, the preempted task T_4 , which has local execution 2.8, has already consumed 1.5 units of its work, meaning that its remaining execution unit amount is $2.8 - 1.5 = 1.3$. Hence, task T_4 is instructed to wait until time $T_{cur} = (10 - 1.3) = 8.7$, at which it will reach zero laxity as the worst case if none of the executing tasks finish their work.

The second C event will be fired by task T_8 , at time $T_{cur} = 1.7$ as mentioned previously and will result in the preemption and migration of task T_2 . In this case, task T_8 , which has local execution 8.3, will start execution from time $T_{cur} = 1.7$ and hence will end execution exactly at the end of the first TL-plane at time $T_{cur} = 10$. On the other hand, the preempted task T_2 , which has remaining execution 3.3, has already consumed 1.7 units of its work, meaning that its remaining execution unit amount is $3.3 - 1.7 = 1.6$. Hence, task T_2 is instructed to wait until time $T_{cur} = (10 - 1.6) = 8.4$, at which it will reach zero laxity as the worst case if none of the executing tasks finish their work.

Figure 2 illustrates the schedule generated by LRE-TL for the first TL-plane, which has the period $[0, 10)$.

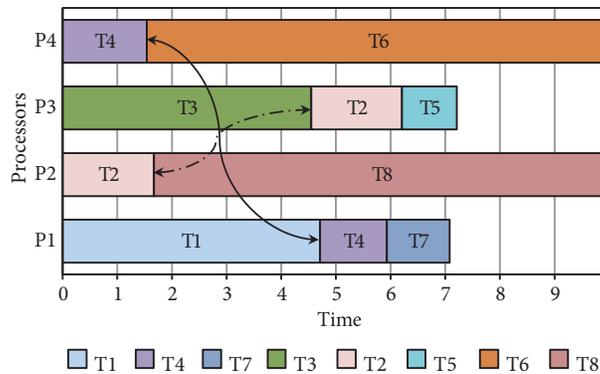


Figure 2. The schedule generated by LRE-TL for the first TL-plane $[0, 10)$ without sorting the tasks.

It can be clearly seen that task T_4 is preempted from processor P_4 at time $T_{cur} = 1.5$ and resumed later on processor P_1 at time $T_{cur} = 4.7$, i.e. after the end of task T_1 . Note that task T_1 is resumed before task T_7 since its laxity when preempted is set to 8.7, which is less than the laxity of task T_7 , which is 8.8.

On the other hand, task T_2 is preempted from processor P_1 at time $T_{cur} = 1.7$ and resumed later on processor P_3 at time $T_{cur} = 4.5$, after the end of task T_3 . Also note that task T_2 is resumed before task T_5 since its laxity when preempted is set to 8.4, which is less than the laxity of task T_5 , which is 9.0.

4.1.2. Case 2: Scheduling with task Sorted using LLREF

On the other hand, when tasks are sorted with their local remaining executions according to LLREF, we get the order shown in Table 5.

Table 5. Task set of Table 3 after sorting with LLREF.

T_i	e_i	p_i	$l_{i,0} [0, 10) = \frac{e_i}{p_i} \times (10 - 0)$
T_6	11	13	8.5
T_8	15	18	8.3
T_1	8	17	4.7
T_3	5	11	4.5
T_2	10	30	3.3
T_4	8	29	2.8
T_7	3	26	1.2
T_5	1	10	1.0

In this case we can see from Table 6 that no C event will be fired since all tasks of heap H_B will finish their executions before tasks of heap H_C consume their local laxity. Therefore, no task migration will happen.

Table 6. Initialization of H_B and H_C for the first TL-plane for task set in Table 5.

TL-plane 0 [0, 10)			
Heap H_B			
T3	T1	T8	T6
4.5	4.7	8.3	8.5
Heap H_C			
T2	T4	T7	T5
6.7	7.2	8.8	9.0

Figure 3 illustrates the generated schedule when tasks are sorted according to LLREF for the first TL-plane, which has the period [0, 10).

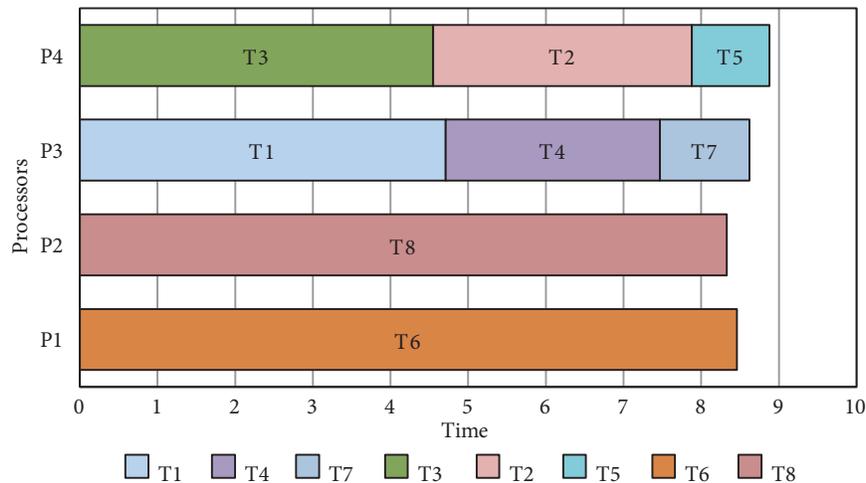


Figure 3. The schedule generated by LRE-TL for the first TL-plane [0, 10) when tasks are sorted according to LLREF.

It can be clearly seen that all tasks successfully completed their executions and none of them are being preempted and resumed later in a different processor, i.e. none of the tasks are preempted and migrated.

5. The proposed solution

We supposed that, as mentioned before, the sorting of tasks would increase the complexity of the TL-plane initializer procedure [14]. To overcome this problem, we utilized the indirect built-in sort functionality of heaps. Since heaps are used to maintain the minimum (or maximum element) in its root node [17,18], we can retrieve the elements from a heap in ascending (or descending) order by extracting items from the heap one by one. In LRE-TL, heap H_C is used to hold tasks that are waiting to be scheduled for execution until they fire C event, i.e. the minimum element in H_C is the task that has the minimum key (laxity), which means it is the closest task to fire a C event. Hence, we propose to do the following: first, we populate heap H_C with all available tasks in the task set T after calculating their local laxity (lines 3–7 in Figure 3). Second, after heap H_C is populated, we extract the first m tasks from it, add them to heap H_B , and assign them to the m processors (lines 9–16 in Figure 3). Since heap H_C maintains the element with the minimum key (laxity) at the top, the tasks extracted back from it will be ordered accordingly to their least laxity first, which is also equivalent to the LLREF order. In this case the complexity of the TL-plane initialization procedure will remain the same and will not be affected. Figure 4 shows the original LRE-TL initializer procedure.

m : number of processors (2, 4, 8, 16, 32)
T: set n of tasks (4, 8, 16, 32, 64)

```

1.   Start
2.   Update the deadline heap with tasks that arrived at  $T_{cur}$ 
3.    $z=1$ 
4.   for all tasks in T
5.      $l = u_i(T_f - T_{cur})$ 
6.     if  $z \leq m$  then
7.        $T_i.key = T_{cur} + l$ 
8.        $T_i.proc-id = z$ 
9.        $z.task-id = T_i$ 
10.     $H_B.insert(T_i)$ 
11.     $z=z+1$ 
12.  else
13.     $T_i.key = T_f - l$ 
14.     $H_c.insert(T_i)$ 
15.  end if
16. end for
17.  $z' = z + 1$ 
18. While  $z' \leq m$  //Null all remaining processors if any
19.    $z'.task-id=NULL$ 
20. end while
21. End

```

Figure 4. The original LRE-TL initialize procedure.

Figure 5 shows the original LRE-TL *initialize* procedure in flowchart form. The proposed LRE-TL initializer procedure is shown in Figure 6.

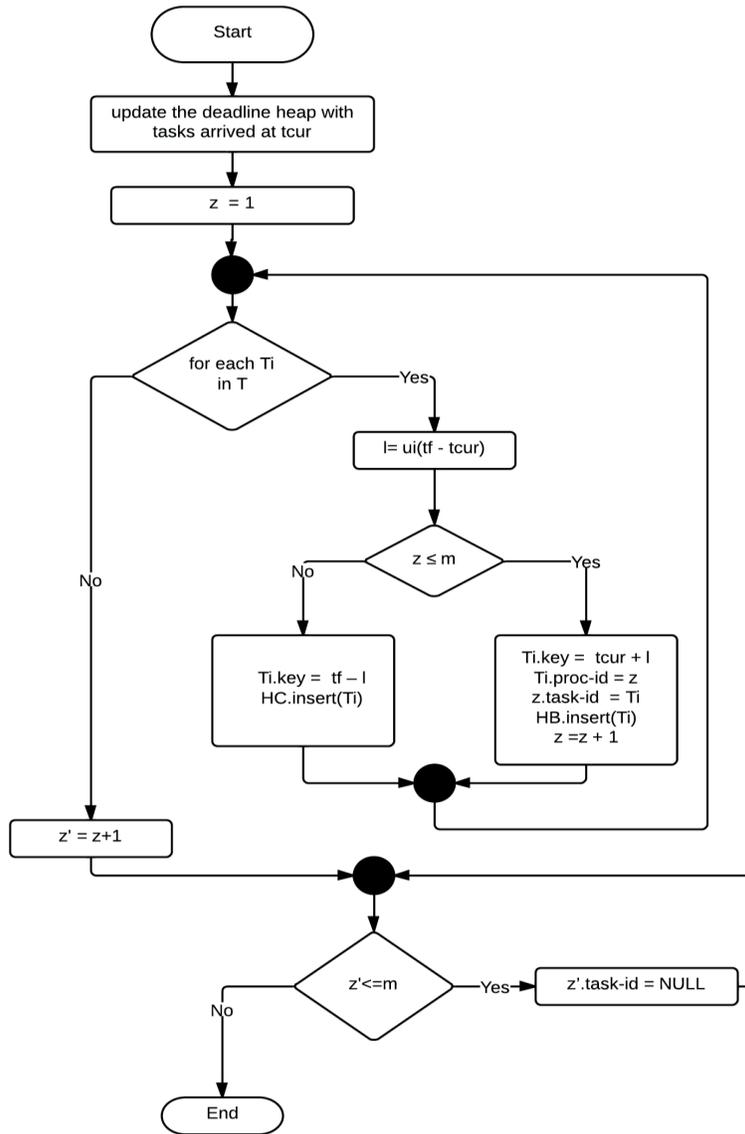


Figure 5. The flowchart of the original LRE-TL initializer procedure.

Figure 7 shows the proposed LRE-TL *initialize* procedure in flowchart form.

6. Results and discussion

In order to test the modified LRE-TL algorithm, we have conducted extensive experimental work. We have tested the algorithm using task sets of size 4, 8, 16, 32, and 64 tasks generated with random utilization using a uniform integer distribution. For each task set we have generated 1000 samples; for example, for the first random set of tasks (i.e. 4 tasks on 2 processors), we have generated 1000 samples, and similarly for the remaining sets. Figure 8 shows the difference between the total task migrations for the first TL-plane for all sets of tasks. It can be clearly seen from Figure 8 that task migrations are greatly reduced when using the modified LRE-TL algorithm.

```

m : number of processors (2, 4, 8, 16, 32)
T: set n of tasks (4, 8, 16, 32, 64)

```

1. Start
2. Update the deadline heap with tasks that arrived at T_{cur}
3. for all active tasks
4. $l = u_i(T_f - T_{cur})$
5. $T_i.key = T_f - l$
6. $H_C.insert(T_i)$
7. end for
8. $z=1$
9. while ($z \leq m$ and NOT $H_C.isEmpty()$)
10. $T = H_C.extract-min()$
11. $T.key = T_f - T.key + T_{cur}$
12. $T.proc-id = z$
13. $z.task-id = T$
14. $H_B.insert(T)$
15. $z = z + 1$
16. end while
17. $z' = z + 1$
18. while $z' \leq m$ //Null all remaining processors if any
19. $z'.task-id = NULL$
20. end while
21. End

Figure 6. The modified LRE-TL initialize procedure.

To verify that the obtained results of the modified LRE-TL algorithm are statistically significant, we have conducted an independent-samples t-test. We have compared the results of task migrations of the modified LRE-TL against the results obtained by the original LRE-TL algorithm.

The stated hypotheses of the t-test are:

- 1) The null hypothesis, denoted H_0 , which states that the difference in task migrations between the modified LRE-TL and the original LRE-TL algorithm is not significant.
- 2) The alternative hypothesis, denoted H_1 , which states that the difference in task migrations between the modified LRE-TL and the original LRE-TL algorithm is significant.

Table 7 summarizes the obtained t-test results. Note that the \bar{X} column refers to the average number of migrations, the Std column refers to the standard deviations, the t-value column refers to the obtained t-test value, and the P-value column refers to the probability of the obtained t-test result. The significance of the t-test results depends on whether the obtained P-value is less than the stated significance level, i.e. $\alpha = 0.01$ or $\alpha = 0.05$. If the P-value is greater than the stated value of α then the t-test accepts the null hypothesis, H_0 , and rejects the alternative hypothesis, H_1 , and hence no significance is reported. Otherwise, the t-test

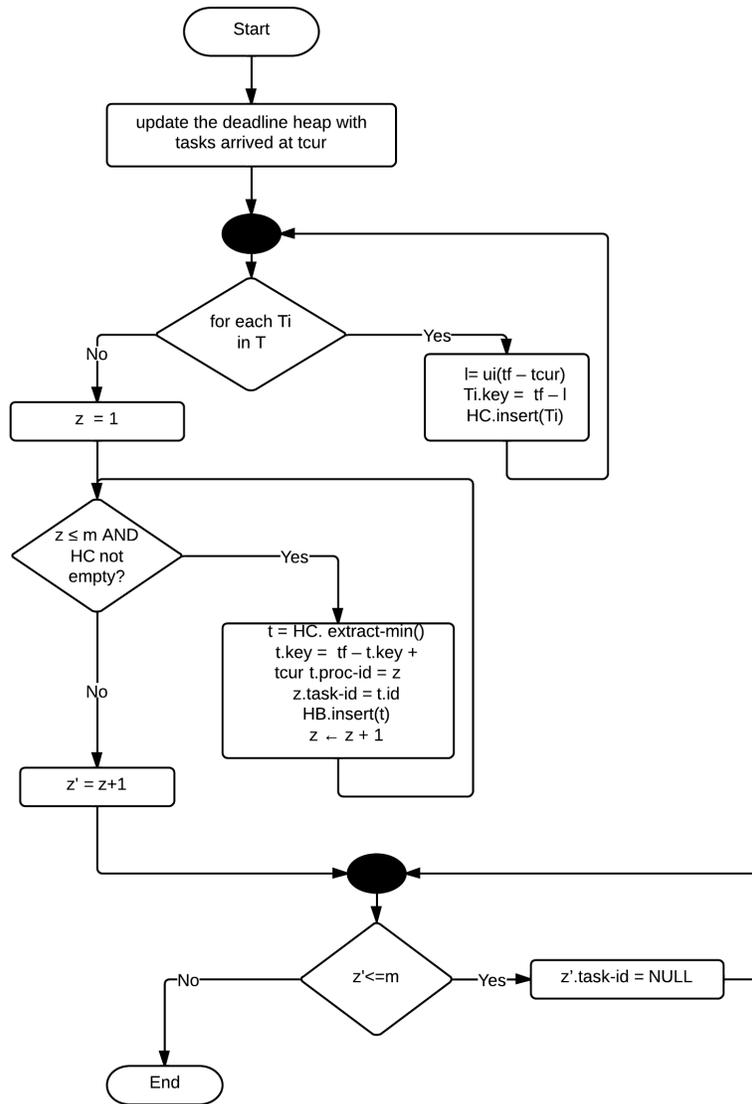


Figure 7. The flowchart of the proposed LRE-TL initializer procedure.

rejects the null hypothesis, H_0 , and accepts the alternative hypothesis H_1 , which indicates the significance of the obtained results.

As can be clearly seen from Table 7 the obtained P-values of the t-test are all less than the stated value of α (0.01) and hence the accepted hypothesis is H_1 , which means that the obtained results are of significant difference.

On the other hand, the sign of the obtained t-value indicates the direction of the difference in sample means. Since the signs of the obtained t-values are all negative, this means that the mean of the first sample, i.e. the modified LRE-TL algorithm, is less than the mean of the second sample, i.e. the original LRE-TL algorithm. Hence, we can conclude that the results of the t-test indicate a significant reduction in the obtained task migration results at the level of $\alpha = 0.01$. These results suggest that the modified LRE-TL algorithm really does have an effect on task migrations. Specifically, our results suggest that when the modified LRE-TL algorithm is used, task migrations are reduced significantly.

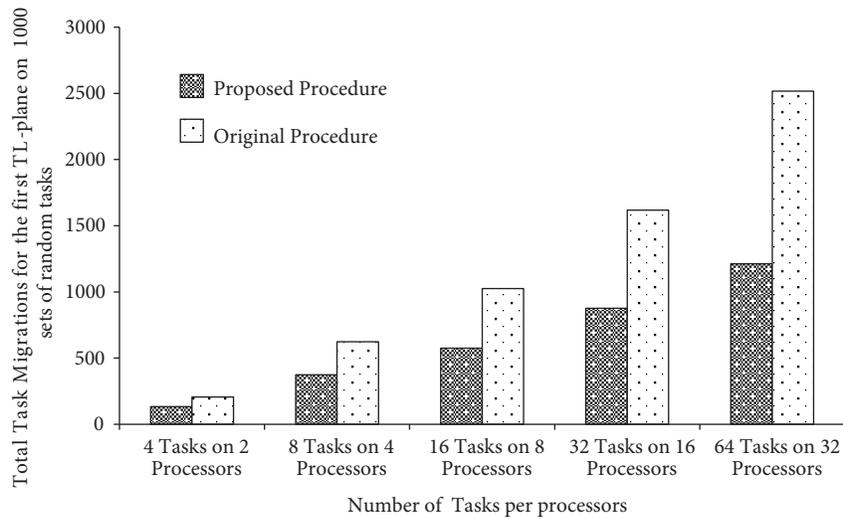


Figure 8. Total task migrations: modified vs. original LRE-TL ($U \leq m$).

Table 7. The t-test results.

Test sample	The modified LRE-TL algorithm		The original LRE-TL algorithm		t-value	P-value	Accepted hypothesis
	\bar{X}	Std	\bar{X}	Std			
4 tasks on 2 processors	0.132	0.339	0.206	0.405	-4.435	9.7282e-06	H ₁
8 tasks on 4 processors	0.373	0.699	0.624	0.812	-7.409	1.8809e-13	H ₁
16 tasks on 8 processors	0.575	1.151	1.026	1.364	-7.989	2.3084e-15	H ₁
32 tasks on 16 processors	0.876	1.931	1.617	2.321	-7.762	1.3462e-14	H ₁
64 tasks on 32 processors	1.212	3.255	2.517	4.044	-7.950	3.1690e-15	H ₁

We have also tested the algorithm using random task sets generated with full utilization, i.e. $\sum_{i=1}^n u_i = m$.

Figure 9 shows the difference between the total task migrations for the first TL-plane for all generated task sets. In this case, the modified algorithm also outperforms the original one, as can be seen from Figure 9 and the reported reduction in task migrations.

Furthermore, we have implemented the modified algorithm as well as the original one using the task set example given in Table 3 earlier from time $t = 0$ until time $t = 29$, i.e. the first 10 TL-planes. The implementation has been conducted on a machine with a Core I7 processor equipped with 4 cores. We have used Java Visual VM of Oracle (<https://visualvm.java.net>) to trace the tasks. Figure 10 shows the results of the CPU profiler of Java Visual VM for the original LRE-TL.

In Figure 11, we show the results of the CPU profiler of Java Visual VM for the modified LRE-TL.

The results achieved by the modified algorithm can be seen in the reduction of the number of invocations of the procedure *handleBorCEvent* that handles both scheduling events B and C, as well as the helper procedure used to manage the heaps.

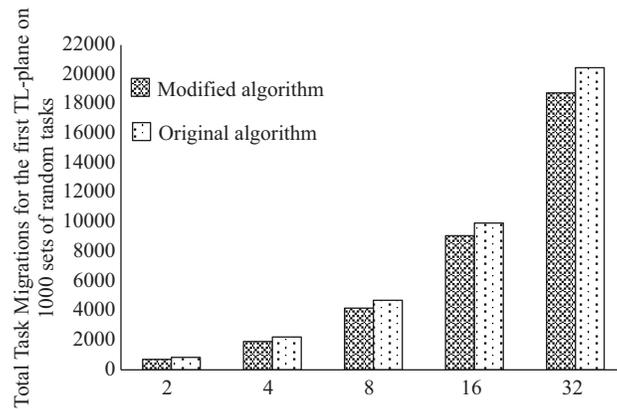


Figure 9. Total task migrations: modified vs. original LRE-TL ($U = m$).

Hot Spots - Method	Self time [%] ▼	Self time	Invocations
javaapplication1.Task.run ()		1040982 ms (100%)	4
javaapplication1.LRE_TL.handleBOrCEvent ()		22.2 ms (0%)	90
javaapplication1.TaskHeap.getMinimum ()		19.6 ms (0%)	9694
javaapplication1.LRE_TL.TL_Plane_Initialize ()		15.7 ms (0%)	10
javaapplication1.Task.myWakeUp ()		7.92 ms (0%)	9965
javaapplication1.TaskHeap.isEmpty ()		7.64 ms (0%)	9854
javaapplication1.TaskHeap.printHeap ()		4.43 ms (0%)	20
javaapplication1.Task.<init> ()		1.75 ms (0%)	80
javaapplication1.TaskHeap.siftDown (int)		0.729 ms (0%)	240
javaapplication1.TaskHeap.siftUp (int)		0.577 ms (0%)	290
javaapplication1.Task.proceed ()		0.419 ms (0%)	60
javaapplication1.TaskHeap.insert (javaapplication1...		0.379 ms (0%)	160
javaapplication1.TaskHeap.removeMin ()		0.371 ms (0%)	160

Figure 10. The result of CPU profiler of Java Visual VM for the original algorithm.

Hot Spots - Method	Self time [%] ▼	Self time	Invocations
javaapplication1.Task.run ()		946921 ms (100%)	4
javaapplication1.LRE_TL.TL_Plane_Initialize ()		20.5 ms (0%)	10
javaapplication1.TaskHeap.getMinimum ()		20.3 ms (0%)	9892
javaapplication1.LRE_TL.handleBOrCEvent ()		16.5 ms (0%)	80
javaapplication1.Task.myWakeUp ()		8.69 ms (0%)	9925
javaapplication1.TaskHeap.isEmpty ()		7.76 ms (0%)	10012
javaapplication1.TaskHeap.printHeap ()		4.41 ms (0%)	20
javaapplication1.Task.<init> ()		2.75 ms (0%)	120
javaapplication1.TaskHeap.siftDown (int)		0.456 ms (0%)	160
javaapplication1.TaskHeap.siftUp (int)		0.403 ms (0%)	190
javaapplication1.TaskHeap.<init> (int)		0.392 ms (0%)	20
javaapplication1.LRE_TL\$.compare (Object, Obj...		0.345 ms (0%)	170
javaapplication1.Task.proceed ()		0.330 ms (0%)	80
javaapplication1.TaskHeap.insert (javaapplicatio...		0.292 ms (0%)	120
javaapplication1.TaskHeap.removeMin ()		0.267 ms (0%)	120

Figure 11. The result of CPU Profiler of Java Visual VM for the modified algorithm.

7. Conclusion

One of the major issues that affect the practicality of optimal real-time multiprocessor scheduling algorithms is the large amount of scheduling overhead they generate. Hence, this paper presented a modified version of the LRE-TL algorithm aimed at reducing task migration overheads. LRE-TL does not consider tasks with the largest local remaining execution to be scheduled for execution first. We have discovered that such tasks with largest local remaining execution always have the minimum laxity, which means that not selecting them for execution first will increase their probability of firing a C event, which in turn results in task migration. For example, the simulation showed that on 2 processors, the achieved reduction in task migrations was 64%. On 4 processors, the achieved reduction was 59%. On 8 processors, the achieved reduction was 56%. On 16 processors, the achieved reduction was 54%. On 32 processors, the achieved reduction was 48%. The statistical t-test conducted showed a significant reduction of task migrations when using the modified version of LRE-TL against the original one.

Although the modified LRE-TL algorithm presented in this paper reduced the amount of task migrations by 56% on average, the incurred overhead still seems to be quite high. For example, for the task set generated with full utilization (Figure 9), the average number of task migrations on 32 processors achieved was 19 migrations per TL-plane, i.e. more than half of the number of processors. This initiates the need for new approaches rather than fairness to schedule real-time tasks even though it ensures the optimality of the algorithm.

References

- [1] Kopetz H. *Real-Time Systems*. 2nd ed. New York, NY, USA: Springer, 2013.
- [2] Burns A, Wellings AJ. *Real-Time Systems and Programming Languages*. 4th ed. Toronto, Canada: Pearson Education, 2009.
- [3] Buttazzo GC. *Hard Real-Time Computing Systems*. 3rd ed. New York, NY, USA: Springer, 2013.
- [4] Laplante PA, Ovaska SJ. *Real-Time Systems Design and Analysis*. 4th ed. New York, NY, USA: Wiley, 2011.
- [5] Liu JWS. *Real-Time Systems*. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall, 2000.
- [6] Nelissen G, Berten V, Nélis V, Goossens J, Milojevic D. U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In: *24th Euromicro Conference on Real-Time Systems; 11–13 July 2012; Pisa, Italy*. New York, NY, USA: IEEE. pp. 13-23.
- [7] Funk S, Levin G, Sadowski C, Pye I, Brandt S. DP-Fair: A unifying theory for optimal hard real-time multiprocessor scheduling. *Real-Time Syst* 2011; 47: 389-429.
- [8] Audsley N, Burns A, Davis R, Tindell K, Wellings A. Real-time system scheduling. In: Randell B, Laprie J-C, Kopetz H, Littlewood B, editors. *Predictably Dependable Computing Systems*. Berlin, Germany: Springer, 1995, pp. 41-52.
- [9] Nelissen G, Berten V, Goossens J, Milojevic D. Reducing preemptions and migrations in real-time multiprocessor scheduling algorithms by releasing the fairness. In: *IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications; 29–31 August 2011; Toyama, Japan*. New York, NY, USA: IEEE. pp. 15-24.
- [10] Davis RI, Burns A. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput Surv* 2011; 43: 1-44.
- [11] Bastoni A, Brandenburg BB, Anderson JH. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In: *31st IEEE Real-Time Systems Symposium; 30 November–3 Dec 2010; San Diego, CA, USA*. New York, NY, USA: IEEE. pp. 14-24.
- [12] Bastoni A, Brandenburg BB, Anderson JH. Is semi-partitioned scheduling practical? In: *23rd Euromicro Conference on Real-Time Systems; 6–8 July 2011; Porto, Portugal*. New York, NY, USA: IEEE. pp. 125-135.

- [13] Liu CL, Layland JW. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J ACM* 1973; 20: 46-61.
- [14] Funk S. LRE-TL: An optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. *Real-Time Syst* 2010; 46: 332-359.
- [15] Baruah SK, Cohen NK, Plaxton CG, Varvel DA. Proportionate progress: a notion of fairness in resource allocation. *Algorithmica* 1996; 15: 600-625.
- [16] Cho H, Ravindran B, Jensen ED. An optimal real-time scheduling algorithm for multiprocessors. In: 27th IEEE Real-Time Systems Symposium; 5–8 December 2006; Rio de Janeiro, Brazil. New York, NY, USA: IEEE. pp. 101-110.
- [17] Goodrich M, Tamassia R, Goldwasser M. *Data Structures and Algorithms in Java*. 6th ed. New York, NY, USA: Wiley, 2014.
- [18] Ma W. *Data Structures and Algorithms Analysis in Java*. 3rd ed. London, UK: Pearson, 2011.