Computers and Electrical Engineering 50 (2016) 143–165

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

An unfair semi-greedy real-time multiprocessor scheduling algorithm☆

Hitham Alhussian[a,*], Nordin Zakaria[a], Ahmed Patel[b,c]

[a] Universiti Teknologi Petronas, 32610 Bandar Seri Iskandar, Perak Darul Ridzuan, Malaysia
[b] Computer Networks Dept., Faculty of Computer Science & Information System, Jazan University, Saudi Arabia
[c] School of Computing and Information Systems, Faculty of Science, Engineering & Computing, Kingston University, Kingston upon Thames KT1 2EE, United Kingdom

ARTICLE INFO

Article history:
Received 20 November 2014
Revised 3 July 2015
Accepted 3 July 2015
Available online 26 July 2015

Keywords:
Real-time
Multiprocessor
Scheduling
Preemption
Migration
Semi-greedy

ABSTRACT

Most real-time multiprocessor scheduling algorithms for achieving optimal processor utilization, adhere to the fairness rule. Accordingly, tasks are executed in proportion to their utilizations at each time quantum or at the end of each time slice in a fluid schedule model. Obeying the fairness rule results in a large number of scheduling overheads, which affect the practicality of the algorithm. This paper presents a new algorithm for scheduling independent real-time tasks on multiprocessors, which produces very few scheduling overheads while maintaining high schedulability. The algorithm is designed by totally relaxing the fairness rule and adopting a new semi-greedy criterion instead. Simulations have shown promising results, *i.e.* the scheduling overheads generated by the proposed algorithm are significantly fewer than those generated by state-of-the-art algorithms. Although the proposed algorithm sometimes misses a few deadlines, these are sufficiently few to be tolerated in view of the considerable reduction achieved in the scheduling overheads.

1. Introduction

Real-time systems maintain their correctness by producing output results within specific time constraints called deadlines [1]. The deadlines of a given real-time taskset cannot be met without the use of an optimal scheduling algorithm unless some constraints are imposed. [1] An optimal scheduling algorithm, with regard to a system and a task model, can be defined as one which can successfully schedules all of the tasks without missing any deadline for any schedulable taskset [2–4].

Optimal real-time multiprocessor scheduling algorithms always achieve high processor utilization that is equal to the number of processors in the system. Most of these algorithms achieve optimality by adhering to the fairness rule completely or partially. Under the fairness rule, tasks are forced to make progress in their executions in proportion to their utilizations. An example of an algorithm that strictly follows the fairness rule is P-fair [5], which forces all tasks to advance their executions in proportion to their utilizations at each time quantum. DP (Deadline Partitioning) algorithms such as LLREF (Largest Local Remaining Executions First), LRE-TL (Largest Remaining Execution-Time and Local time domain) and DP-Wrap

☆ Reviews processed and recommended for publication to the Editor-in-Chief by Guest Editor Dr. Yingpeng Sang.

* Corresponding author.
E-mail addresses: halhussian@gmail.com (H. Alhussian), nordinzakaria@gmail.com (N. Zakaria), whinchat2010@gmail.com (A. Patel).

[1] For example, the RM (Rate Monotonic) scheduling algorithm is optimal for the scheduling of tasks with implicit deadlines scheduled with fixed task priorities on uniprocessor platforms but is not optimal for the scheduling of tasks with constrained deadlines or if dynamic priorities are allowed [4].

http://dx.doi.org/10.1016/j.compeleceng.2015.07.003
0045-7906/

(Deadline Partitioning-Wrap) [3,6,7] partially follow the fairness rule by forcing tasks to make progress in their executions in proportion to their utilizations at the end of each TL-plane (time slice) in a fluid schedule model, which corresponds to the deadline of tasks in the system. Although adhering to the fairness rule always ensures optimality, it produces a large number of scheduling overheads in terms of task preemptions and migrations which adversely affect the practicality of the algorithm [2,7] because the processors will be busy executing the scheduler itself rather than executing the actual work [2]. In fact, the empirical study in [8] confirmed that preemption and migration delays could be as high as 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 consider a reduction in the scheduling overheads in order to be practically implemented.

To further explain the problem of following the fairness rule, consider the taskset shown in Table 1 [6] to be scheduled on a system of 4 processors. In DP algorithms, such as LLREF, LRE-TL, and DP-Wrap, the fairness rule is always ensured at the deadline of tasks; they divide the time into TL-planes, *i.e.* time slices as mentioned previously, which are bounded by two successive deadlines, and the end of each TL-plane corresponds to the deadline of a task in the system. Hence, tasks are marshalled in the intervals [0,5), [5,7), [7,10), [10,14), [14,15), [15,16), [16,17), [17,19), [19,20), [20,21), [21,25), [25,26), [26,28), and [28,29), which correspond to the first 14 TL-planes, after which all tasks would finish at least one period of their executions. This means that at the beginning of each TL-plane, all tasks have to be allocated local executions proportional to their utilizations and marshalled until the end of the time slice at which they must all be preempted. This will result in numerous preemptions as well as migrations. For example, although task T_2 has worst-case execution requirements of 1 and period of 16, it is forced to make progress in its executions in each TL-plane even though it can wait for 15 units of time before it become critical. The same case holds for task T_5 (worst-case execution requirements of 2 and period of 26) which can wait for 24 units of time before it become critical, however, it is also forced to make progress in its execution in each TL-plane. This means that task T_1 will be preempted 6 times before it reaches its deadline, and similarly, task T_5 will be preempted 11 times before it reaches its deadline.

In this paper, we present an efficient global real-time multiprocessor scheduling algorithm, namely, USG (**U** nfair **S** emi-**G** reedy). It is "Unfair" because we have totally relaxed the fairness rule, and it is "Semi-Greedy" because we have employed two policies: the Non-Preemptability policy to avoid the problem of greedy schedulers as well as to reduce the scheduling overheads, and the Zero-Laxity policy to maintain the criticality of the system as well as to increase the schedulability of the algorithm.

The remainder of this paper is organized as follows. Section 2 briefly reviews related studies. Section 3 describes the task model and defines the terms that will be used in this paper. Section 4 presents the proposed algorithm and illustrates its underlying mechanism with examples. Section 5 analyses the deadline misses under the proposed algorithm. Section 6 discusses the run time analysis of the proposed algorithm. Section 7 presents and discusses the results obtained using the proposed algorithm. Finally, Section 8 states the conclusions.

2. Related work

LLF (Least Laxity First) [9], initially introduced as the least slack algorithm, is a fully dynamic scheduling algorithm, *i.e.* the priorities of jobs change dynamically according to their laxity which in turn changes over time. Although this dynamicity of LLF can increase its schedulability, it has a negative impact because it generates a large number of preemptions and migrations, which adversely affect its practicality. Therefore, LLF has not attracted much research attention even though its optimality has been proven for uniprocessor systems.

The authors in [10] developed a new schedulability test for LLF based on its dynamicity of laxity values that changes over time. They showed that the new LLF schedulability dominates the state-of-the-art EDZL (Earliest Deadline until Zero Laxity) schedulability tests. In their future work, they plan to develop variants of LLF in order to reduce the large number of scheduling overheads generated by the algorithm.

On the other hand, the authors in [11] showed that despite the simplicity of the Zero-Laxity policy, it is quite effective in handling general task systems on multiprocessors, as it integrates both urgency and parallelism. They also claimed that any work-conserving preemptive algorithm employing the Zero-Laxity policy dominates the original algorithm itself. A good example of a laxity-based algorithm is EDZL (Earliest Deadline first until Zero Laxity) [12], which extends the EDF (Earliest Deadline First) by giving tasks with zero laxity the highest priority and executes them until completion without interruption.

The P-fair (Proportionate fair) algorithm is the first optimal real-time multiprocessor scheduling algorithm to be proposed [5]. P-fair is defined for periodic tasks with implicit deadlines. It executes tasks in proportion to their utilization, based on the concept of fluid scheduling, by dividing the timeline into quanta of equal length. The algorithm allocates tasks to the processors at every time quantum t, such that the accumulated processor time allocated to each task t_i will be either $\lceil tu_i \rceil$ or $\lfloor tu_i \rfloor$.[2] P-fair can achieve an optimal utilization bound $U \leq m$ [4,5]. However, it is difficult to implement P-fair in practice because it makes scheduling decisions at each time quantum, resulting in numerous task preemptions and migrations. Many subsequent versions of the P-fair algorithm have been proposed (e.g. PD (Pseudo-Deadline) [13], PD^2 (Pseudo-Deadline 2)

[2] Remember that $\lceil tu_i \rceil$ refers to the ceiling function, while $\lfloor tu_i \rfloor$ refers to the flooring function. For example $\lceil 3.2 \rceil = 4$ while $\lfloor 3.2 \rfloor = 3$.

Table 1
A feasible taskset on 4 processors.

Task	Execution requirement (E)	Period (P)
T_1	3	7
T_2	1	16
T_3	5	19
T_4	4	5
T_5	2	26
T_6	15	26
T_7	20	29
T_8	14	17

[14], and ER-PD (Early Released Pseudo-Deadline) [15]). However, all of them suffer from numerous preemptions and migrations because they ensure the fairness property at each time quantum.

Instead of ensuring the fairness rule at each time quantum t, a new technique known as DP (Deadline Partitioning) is introduced, in which time is partitioned into slices [4,7,16]. 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 in proportion 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 number of preemptions and migrations is reduced significantly as compared to the P-fair algorithms. Based on DP, a family of algorithms, known as DP-Fair (Deadline Partitioning Fair) [7] or Boundary Fair [16], has been proposed; it includes BF (Boundary Fair) [16], LLREF [6], LRE-TL [3], and DP-Wrap [7].

The EKG (**E**arliest deadline first with task splitting and **K** processors in a **G**roup) [17] algorithm uses a parameter k, $1 \leqslant k \leqslant m$, to split tasks among a group of k processors. When $k = 1$, EKG uses partitioned EDF to schedule tasks with a limited utilization bound. When $k = m$, EKG schedules tasks in a way that is very similar to DP-Wrap, with utilization equal to m. The authors in [18] improved EKG to reduce the number of generated preemptions and migrations by decreasing the number of time slices needed to ensure that deadlines are met on the one hand, and by using a swapping algorithm to exchange the execution time between tasks and time slices on the other hand.

In [19], a new optimal algorithm, namely, RUN (**R**eduction to **Un**iprocessor), 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. Further, it has been shown that RUN achieves better performance than existing optimal algorithms in terms of task preemptions with no more than 3 preemptions per job; however, it cannot handle scheduling of sporadic tasks.

In [2], the following conclusion has been drawn from a study of the above-mentioned algorithms: whenever the fairness rule is relaxed, the number of preemptions and migrations decreases. This can be clearly observed in the case of P-fair algorithms, which produce many more scheduling overheads than BF or DP-Fair, which in turn produce more scheduling overheads than EKG. According to this observation, the authors of [2] proposed the U-EDF (**U**nfair-**E**arliest **D**eadline **F**irst) algorithm in which they released the fairness property and incorporated EDF rules in the scheduler. As a result, their algorithm shows better performance in terms of the number of preemptions and migrations. However, their pre-allocation algorithm has a run-time complexity of $O(n \times m)$ per task release, which seems to be quite high [2].

In [20], a new algorithm, namely, QPS (Quasi-Partitioned Scheduling), is introduced. It is able to schedule any feasible taskset with implicit deadlines on identical processors. QPS uses a hybrid method that alternates between partitioned and global scheduling according to the system overloads. Although QPS has been proved to be optimal, it does not perform as well as its competitors, RUN and U-EDF, in terms of the number of preemptions and migrations. For systems that do not need full utilization, QPS performance becomes comparable with that of RUN and exceeds that of U-EDF.

With regard to optimal real-time multiprocessor scheduling algorithms, both RUN and U-EDF outperform all other existing algorithms [2,19] in terms of the generated scheduling overheads. In this paper, we show how our novel algorithm achieves better performance in terms of the number of preemptions and migrations.

3. Model and term definitions

In this paper, we consider the problem of scheduling n independent periodic tasks with implicit deadlines (deadlines equal to periods, *i.e.* $d_i = p_i$) on a platform of m symmetric SMPs (Shared-Memory Multiprocessors). In real-time systems, a periodic task is one that is released periodically at a constant rate. Usually, two parameters are used to describe a periodic task T_i: its worst-case execution time e_i and its period p_i. An instance of a periodic task (*i.e.* release) is known as a job and is expressed as $T_{ij} = (e_{ij}, p_{ij})$, where $j = 1, 2, 3, \ldots, e_{ij}$ denotes the worst-case execution requirement of job T_{ij}, and p_{ij} denotes its period. The deadline of a job is the arrival time of its successor. For example, the deadline of the job T_{ij} would be the arrival time of job $T_{i(j+1)}$, *i.e.* at $(j+1)p_i$. We use the term $e_{ij,t}$ to denote the remaining execution of job T_{ij} at time t. Consequently, the laxity L_{ij} of a job T_{ij} at time t, given by Eq. (1), is the time for which T_{ij} can remain idle before its execution should commence, *i.e.* when it reaches zero laxity.

$$l_{ij,t} = p_{ij} - e_{ij,t} - t \tag{1}$$

One more important parameter that is used to describe a task T_i is its utilization. The utilization u_i of a task T_i, given by Eq. (2), is the portion of the time that it needs to execute after it has been released and before it reached its deadline.

$$u_i = \frac{e_i}{p_i} \tag{2}$$

U_{sum} denotes the total utilization of a given taskset T whereas U_{max} denotes its maximum utilization. A periodic taskset T is said to be schedulable on m identical multiprocessors *iff*

$$U_{sum}(T) <= m, \quad \text{and} \quad U_{\max}(T) \leq 1 \tag{3}$$

We use the term $n_i(t)$, given by Eq. (4), to refer to the number of instances or releases of a task T_i during time t.

$$n_i(t) = \left\lceil \frac{t}{p_i} \right\rceil \tag{4}$$

The remaining utilization of the jth job of task T_i at time t is given by Eq. (5).

$$r_{i_j,t} = \frac{e_{i_j,t}}{n_i(t)\,p_i - t} \tag{5}$$

Hence, the total remaining utilization of a taskset T at time t is

$$R_t = \sum r_{i,t} \tag{6}$$

If T is a schedulable taskset, *i.e.* $U \leqslant m$, then at any time t

$$R_t \leq U \tag{7}$$

Table 2 summarizes the terms and notations used in this paper, along with their definitions.

4. The proposed algorithm

The key concept underlying the proposed algorithm is the total relaxation of the fairness rule in order to avoid a large number of scheduling overheads in the form of task preemptions and migrations. However, totally relaxing the fairness rule leads to greedy schedulers which fail to schedule some tasksets, as explained in [7]. A greedy scheduler is one in which a job is executed according to a specific priority, e.g. according to the job's deadline or laxity. EDF and LLF are well-known examples of greedy schedulers [7].

To mitigate the adverse effects of completely relaxing the fairness rule as well as to alleviate the problem of greedy schedulers, we have proposed a semi-greedy algorithm, namely, USG (***U**nfair **S**emi-**G**reedy*). A semi-greedy scheduler can be defined as one in which the priority rule can sometimes be violated or loosely forced. As stated previously, the concept underlying the semi-greedy property is the use of the *Non-Preemptability* policy bounded by the *Zero-Laxity* policy.

Although non-preemptive scheduling has not been studied as extensively as preemptive scheduling owing to its poor performance in real-time systems, it is widely used in real-world applications [21–23]. It may even be preferred to preemptive scheduling [23] because it is easier to implement and has fewer runtime overheads as compared to its counterparts. On the other hand, it is far more difficult to predict the overheads of preemptive scheduling algorithms because of inter-task interference caused by caching and pipelining; the latter can become even more difficult to predict on multiprocessor platforms. Non-preemptive scheduling can alleviate this problem. To maintain the system's criticality and to avoid the adverse effects of non-preemptive scheduling, we have employed the Zero-Laxity policy [11]. Thus, tasks are allowed to execute until an awaiting task raises a Zero-Laxity event. This allows tasks with greater laxity to advance their executions before the ones with the least laxity become critical, *i.e.* reach zero laxity. Thus, the proposed algorithm significantly reduces the scheduling overheads in the form of task preemptions and migrations as compared to the state-of-the-art algorithms. However, as a result of totally relaxing the fairness rule, the proposed algorithm may sometimes miss a few deadlines; nonetheless, these misses are sufficiently few to be tolerable in view of the considerable reduction in task preemptions and migrations. In the examples that appear later in this paper, we show the difference between the proposed algorithm and greedy schedulers.

The proposed algorithm maintains a global queue ordered by increasing laxity. The algorithm is a work-conserving one, *i.e.* whenever a task completes its execution and there are available tasks in the waiting queue, the one with the least laxity is picked up and executed on the same processor as the completed task. The algorithm preempts a task if and only if a waiting task reaches zero laxity, *i.e.* the algorithm can allow tasks with greater laxity to continue their executions even if tasks with the least laxity are waiting. Moreover, if a task reaches zero laxity, the algorithm preempts the task with the maximum laxity from the running list. As will be shown later, preempting a task with maximum laxity significantly reduces task migrations and increases the schedulability of the algorithm as compared to preempting a task with minimum remaining work, as has been done in [24].

Table 2
Summary of terms and notations.

Term	Definition	Term	Definition
m	Number of processors	l_i	Laxity of task T_i
n	Number of tasks	l_{ij}	Laxity of the jth job of task T_i
T	Set of tasks $\{T_1, T_2, \ldots, T_n\}$	$l_{ij,t}$	Remaining laxity of the jth job of task T_i at time t
T_i	The ith task	u_i	Utilization of task T_i
T_{ij}	The jth job of task T_i	U, U_{sum}	Total utilization of T
p_i	Period (minimum inter-arrival time) of task T_i	U_{max}	Maximum utilization in T
p_{ij}	Period (minimum inter-arrival time) of the jth job of task T_i	$n_i(t)$	The number of instances or releases of a task T_i during time t
e_i	Worst-case execution requirement of task T_i	$r_{i_j,t}$	The remaining utilization of the jth job of task T_i at time t
e_{ij}	Worst-case execution requirement of the jth job of task T_i	R_t	The total remaining utilization of a taskset T at time t
$e_{ij,t}$	Remaining worst-case execution requirement of the jth job of task T_i at time t		

4.1. Data structure of the proposed algorithm

We propose the use of the heap data structure as in [3] to implement our algorithm. A heap is a data structure which maintains the minimum item (*key*) in its root node [25]. The algorithm includes three heaps, as shown in Fig. 1. The first heap is H_R, which is used to implement the list of "Running" tasks. The second heap is H_W, which is used to implement the list of "Waiting" tasks. The third heap is H_{NA}, which is used to implement the list of "Not Active" tasks.

The value of the *key*, *i.e.* the minimum item in the heap, differs from one heap to another. In heap H_R, the value of the key represents the time at which the task will finish its execution. In heap H_W, the value of the key represents the time at which the task will be critical, *i.e.* when it reaches zero laxity. In heap H_{NA}, the value of the *key* represents the time at which the task will be active, *i.e.* when it arrives again.

4.2. Task states

We propose the use of three task states in our new algorithm: *Running*, *Waiting*, and *Not Active*. Fig. 2 shows the proposed task states as well as the possible transitions between them. The tasks traverse between the proposed states with the advancement of time. In the following subsections, we discuss each of these states in detail.

4.2.1. The running state

A task is in the running state if it is already assigned to a processor and is currently executing. When a task is currently executing, its remaining execution decreases with the increase in time, while its laxity remains constant. A task T_i can arrive at the running state either from the waiting state or from the not active state. There are two options for T_i to arrive at the running state from the waiting state. The first option is when T_i reaches zero laxity (see Section 4.3.2). The second option is when a running task ends execution and releases its processor; hence, the scheduler should select one of the waiting tasks to start/resume its work on the released processor. In both options, T_i's key is calculated using Eq. (8) to denote the time at which it will end execution by adding the current time t to the remaining execution requirement of T_i

$$T_i \cdot key = T_i \cdot p - T_i \cdot key \ MOD \ T_i \cdot p + t \tag{8}$$

On the other hand, task T_i can arrive at the running state from the not active state *iff* its utilization is equal to one, *i.e.* its $u_i = 1$, which means that the task cannot wait any more and its execution should commence immediately. Otherwise, it will miss its deadline. In this case, T_i's key is calculated using Eq. (9) to denote the time at which it will end execution by adding the current time t to its execution requirement.[3]

$$T_i \cdot key = T_i \cdot e + t \tag{9}$$

4.2.2. The waiting state

A task is in the waiting state if it is currently idle. The maximum time for a task to remain idle is until it reaches zero laxity. When a task is idle, its laxity decreases with the increase in time, while its remaining execution remains constant. A task T_i can arrive at the waiting state either from the running state when it is preempted or from the not active state if it has laxity, *i.e.* its $u_i < 1$. Before T_i is preempted, its key represents the time at which it will end execution, as mentioned before. However, when T_i is preempted, we need to change its key to denote the time at which it will reach zero laxity (see Section 4.3.2), and this is accomplished using Eq. (8) again. Actually, Eq. (8) is always used to switch the value of the key from the time at which T_i will end execution to the time at which it will reach zero laxity and vice versa.

[3] Recall that the *key* property holds the value of the root node in the heap, as mentioned in Section 4.1; hence, retrieving its value requires only $O(1)$ time. Therefore, no time overheads were observed in Eqs. (8)–(10) during the simulations.

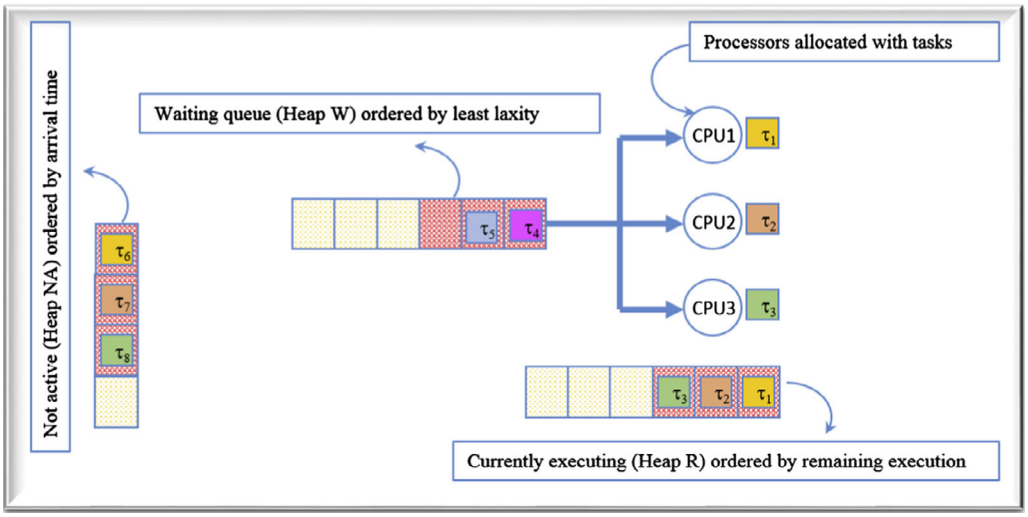

Fig. 1. The infrastructure of the proposed algorithm.

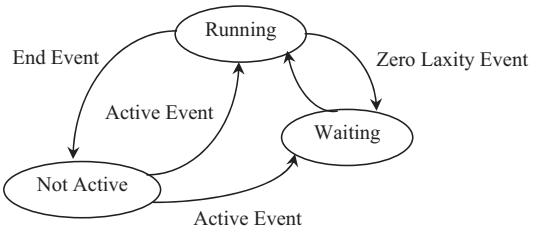

Fig. 2. Proposed task states.

When T_i arrives at the waiting state from the not active state, its key is calculated using Eq. (10) to denote the time at which it will reach zero laxity (see Section 4.3.2).

$$T_i \cdot key = T_i \cdot p - T_i \cdot e + t \tag{10}$$

4.2.3. The not active state

In contrast to the running and waiting states, the not active state has only one entry, *i.e.* when a running task ends its execution. A task T_i arrives at the not active state only when its execution ends, and accordingly, its key is updated to represent the time at which it will be active. In this case, T_i's key is calculated using Eq. (11), where t denotes the current time.

$$T_i \cdot key = \begin{cases} t, & T_i \cdot key \ MOD \ T_i \cdot p = 0 \\ T_i \cdot p - T_i \cdot key \ MOD \ T_i \cdot p + t, & T_i \cdot key \ MOD \ T_i \cdot p > 0 \end{cases} \tag{11}$$

4.3. Scheduling events

A task changes from one state to another as the result of a scheduling event. We propose the use of three scheduling events: the End (E) event, the Zero Laxity (Z) event, and the Active (A) event. In the following subsections, we discuss each of these events in detail.

*4.3.1. The End (*E*) event*

The End (E) event is fired whenever a task ends its execution, *i.e.* when it completes all its execution requirements (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, *i.e.* heap H_{NA}. In this case, the scheduler selects the task with the least laxity from the waiting list, heap H_W, for execution on the same processor occupied by the task just completed, after setting its key to the time at which it will end execution. The completed task will then be added to the not active list, heap H_{NA}, after setting its key to the time at which it will be active again. Accordingly, the state of the ended task will be changed from running to not active, and similarly, the resumed task's state will be changed from waiting to running.

4.3.2. The Zero-Laxity (Z) event

The Zero-Laxity (Z) event is fired whenever a task becomes critical, *i.e*. when it reaches zero laxity, meaning that it cannot wait any longer and it has to be immediately scheduled for execution, or it will miss its deadline. In this case, the scheduler preempts the task with the maximum laxity from the running list and schedules the critical task on the processor of the preempted one. This is done after setting its key to the time at which it will end execution. The preempted task will then be added to the waiting list, *i.e*. heap H_W, by setting its key to the time at which it will become critical, *i.e*. when it reaches zero laxity. Accordingly, the state of the preempted task will be changed from running to waiting, and the resumed task's state will be changed from waiting to running.

4.3.3. The Active (A) event

The Active (A) event is fired whenever a task is released, *i.e*. when it becomes active. When a task becomes active, the scheduler initially searches for any processor left idle; if such a processor is available, the task is immediately scheduled for execution and assigned to that processor. If no processor is left idle, *i.e*. all processors are busy, a choice will be made according to the task's laxity. If the task's laxity is equal to 0, then it will be immediately scheduled for execution by preempting the task with the maximum laxity from the running list. Then, the preempted task will be added to the waiting list, *i.e*. heap H_W, after setting its key to the time at which it will become critical again, *i.e*. when it reaches zero laxity. If the released task has a laxity of greater than 0, it will be forced to wait until it becomes critical by setting its key to the time at which it will reach zero laxity before adding it to heap H_W. Depending on whether the arrived task is critical or not, its state will be changed from not active to either running or waiting as described above. If its state is changed to running, the preempted task's state will accordingly be changed from running to waiting.

4.4. The algorithm procedures

The algorithm consists of four major procedures and one helper procedure. The first is the *main* procedure from which all other procedures are called. The second is the *initialize* procedure, which initializes the heaps. The third is the *handleEorZEvents* procedure, which handles both E and Z events. The fourth is the *handleAEvent* procedure, which handles the A event. The helper procedure is the *extractMaxLaxity* procedure used by the *handleEorZEvents* procedure to preempt the task with the maximum laxity from the running list as a result of firing a Z event. Table 3 summarizes the variable definitions used in all the procedures.

4.4.1. The main procedure

The *main* procedure, shown in Fig. 3, 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, our algorithm calls this procedure once when the system starts to initialize the heaps. After the *initialize* procedure ends, the system starts checking for all types of events as time advances (lines 3–8). First, the system checks for E or Z events (line 3), and whenever one of them occurs, the system responds by calling the *handleEorZEvents* procedure (line 4). Then, the system checks for the A event (line 6),[4] and if a task becomes active, *i.e*. it is released, the system responds by calling the *handleAEvent* procedure (line 7). Finally, all processors are instructed to execute their assigned jobs (line 9), and the system waits until the next time an event will be fired (line 10).

4.4.2. The initialize procedure

The main functionality of the *initialize* procedure is to set the current time to zero and initialize the running list, heap H_R, as well as the waiting list, heap H_W. Fig. 4 shows the algorithm of the *initialize* procedure. The procedure starts by setting the current time, *Tcur*, to zero in line 1. Then, the procedure populates the waiting list, heap H_W, with all the tasks after updating their keys with the laxity of each task (lines 2–5). In lines 7–14, the procedure uses a while loop to extract the first m tasks from heap H_W, updates their keys to the time at which they will fire an E event, *i.e*. end their executions, and adds them to the running list, heap H_R. These tasks will continue to execute until either they end execution or another task(s) from heap H_W becomes critical (reaches zero laxity), as we will see later in the *hadleEorZEvents* procedure. The purpose of this step (populating heap H_W with all tasks, extracting the first m tasks, and adding them to heap H_R) is to ensure that the tasks with the least laxity will be scheduled for execution first, thereby reducing the probability of firing a Z event(s), which leads to a task migration cost. After the while loop in line 7 is terminated, all other tasks remain in heap H_W until they fire a Z event. In lines 16–19, the remaining processors, if any, are set to null, *i.e*. set to idle, and they are pushed into the *idleProcSTK* stack, which keeps track of idle processors to simplify the process of retrieving them later.

4.4.3. The handleEorZEvents procedure

The *handleEorZEvents* procedure, which is shown in Fig. 5, is called whenever an E or Z event is fired. The procedure starts by first handling all E event(s) (lines 2–18). The *while* loop in line 2 checks heap H_R for any task that ends execution, *i.e*. its *key* is equal to the current time. If such a task exists, it is extracted from heap H_R (line 3), and the *Id* of the processor

[4] Remember that the scheduler is monitoring three heaps (queues); hence, after checking for E and Z events, the scheduler also needs to check for an A event in case any task arrives. Further, note that the handlers of E and Z events are combined in one procedure to reduce the number of passes incurred by successive procedure calls.

Table 3
Variable definitions.

Variable	Definition	Variable	Definition
N	Denotes number of tasks	T_{na}	Denotes the task extracted from heap H_{NA}
M	Denotes number of processors	T	Denotes a temporary task
T_S	Denotes a taskset of n tasks	*taskId*	Denotes the ID of the task
P	Denotes a list of m processors	*procId*	Denotes the ID of the processor
H_R	The running heap of size m, holds currently executing tasks	*pq*	Denotes a priority queue
H_W	The waiting heap of size n, holds waiting (idle) tasks	*Tcur*	Denotes the current time
H_{NA}	The not active heap of size n, holds not active tasks	*z*	Denotes the processor ID of the preempted task
T_r	Denotes the task extracted from heap H_R	*IdleProcSTK*	Denotes a stack of idle processors
T_w	Denotes the task extracted from heap H_W		

```
1.  call initialize()                      7.      call handleAEvent()
2.  while true                             8.    end if
3.    if a E or Z event occurs             9.    instructs each processor to execute its job
4.      call handleEorZEvents()            10.   sleep until next time an event is fired
5.    end if                               11. end while
6.    if an A event occurs
```

Fig. 3. The *main* procedure.

n: number of tasks.
m: number of processors.
H_W: Waiting heap of size equal to the number of tasks n.

H_R: Running heap of size m.
P: List of processors of size m.
z: A temporary variable to hold the processor ID of a preempted task.

```
1.  Tcur = 0;
2.  for each task T in T_S
3.    T.key = T.p -T.e;
4.    H_W.insert(T);
5.  end for
6.  z = 0;
7.  while ((H_W.size()>0) and (z<m))
8.    T = H_W.extractMinimum ();
9.    T.key = T.p- T.key+Tcur;
10.   T.procId = z;
11.   P[z].taskId = T.id;
12.   H_R.insert(T);
13.   z = z + 1;
14. end while
15. //Null all remaining processors
16. for i=z to P.length-1
17.   P[i].taskId = 0;
18.   idleProcSTK.push(i);
19. end for
```

Fig. 4. The *initialize* procedure.

in which it was running is assigned to a local variable (line 4). The procedure then checks if there are any waiting tasks available in heap H_W to replace the ended task (line 5). If available tasks in heap H_W exist, the task with the minimum laxity is extracted from H_W (line 6). Then, its *key* is updated to the next time at which it will fire an *E* event (line 7), assigned to the same processor as the ended task (line 8), and added to heap H_R (line 10). If no tasks are available in heap H_W (line 13), the processor occupied by the ended task is set to idle (line 12) and pushed into the idle processor stack *idleProcSTK* (line 13). The ended task will then be added to the not active list, heap H_{NA} (line 17), after updating its *key* to the next time at which it will be released (line 16). The *while* loop in line 2 then continues until all *E* events are handled.

After handling all *E* events, the procedure then continues to check for any *Z* events (lines 20–32). The *while* loop in line 21 checks heap H_W for any task(s) that reaches zero laxity, *i.e.* its *key* is equal to the current time. If such a task exists, it is immediately scheduled for execution by preempting the task with the maximum laxity for it from heap H_R. In line 22, the task with the maximum laxity is extracted from heap H_R, and in line 23, the task that reaches zero laxity is extracted from heap H_W. In lines 24–25, Eq. (8) is used to switch the value of the *key* in both tasks (*Tr*, *Tw*) from the remaining time until the end of execution to the remaining time until zero laxity is reached, and vice versa. In lines 26–27, the processor of the preempted task (*Tr*) is retrieved and assigned to the task with zero laxity (*Tw*). In line 28, the task *Id* of the processor is updated to the *Id* of the zero laxity task (*Tw*). Now, both tasks are ready to be replaced, and thus, task (*Tw*) is added to heap H_R, while task (*Tr*) is added to heap H_W (lines 29–30). The *while* loop in line 21 then continues until all *Z* events are handled.

4.4.4. *The* handleAEvent *procedure*

The *handleAEvent* procedure, which is shown in Fig. 6, is called whenever an *A* event is fired. The procedure starts with the *while* loop in line 1, which checks for any task that becomes active, *i.e.* its *key*, which represents its arrival time, is equal to the current time. If a task becomes active, it is extracted from the not active list, heap H_{NA} (line 2). The procedure then checks if any idle processor is available (line 6). If an idle processor is available, it is retrieved from the idle processor stack

n: Number of tasks.
m: number of processors.
H_W: Waiting heap of size n.
H_R: Running heap of size m.
H_{NA}: Not active heap of size n.
P: List of processors of size m.

```
1.  //Handle E Event
2.  while (($H_R$.getSize()>0) and ($H_R$.getMinimum().key == Tcur))
3.    Task $T_r$ = $H_R$.extractMinimum();
4.    int z = $T_r$.procId;
5.    if ($H_W$.getSize()>0)
6.      Task $T_w$ = $H_W$. extractMinimum ();
7.      $T_w$.key =$T_w$.p - $T_w$.key%$T_w$.p + Tcur;
8.      $T_w$.procId = z;
9.      P[z].taskId = $T_w$.id;
10.     $H_R$.insert($T_w$);
11.   else
12.     P[z].taskId = 0;
13.     idleProcSTK.push(z);
14.   end if
15.   //Add $T_r$ to Not Active heap
16.   $T_r$.key = ($T_r$.key%$T_r$.p==0)? Tcur: ($T_r$.p - $T_r$.key%$T_r$.p + Tcur);
17.   $H_{NA}$.insert($T_r$);
18. end while
19. //Handle Z Event
20. if ($H_W$.getSize()>0)
21.  while ($H_W$.getMinimum().key==Tcur)
22.    Task $T_r$ = extractMaxLaxity($H_R$);
23.    Task $T_w$ = $H_W$.extractMinimum();
24.    $T_r$.key = $T_r$.p -$T_r$.key%$T_r$.p + Tcur;//change to laxity
25.    $T_w$.key = $T_w$.p - $T_w$.key%$T_w$.p + Tcur;//change to key
26.    int z = $T_r$.procId;
27.    $T_w$.procId = z;
28.    P[z].taskId = $T_w$.id;
29.    $H_R$.insert($T_w$);
30.    $H_W$.insert($T_r$);
31.  end while
32. end if
```

Fig. 5. The *handleEorZEvents* procedure.

idleProcSTK (line 8) and assigned to the arrived task (line 9). Similarly, the task *Id* is assigned to the processor (line 10), the task *key* is updated to the next time at which it will fire an *E* event (line 11), and finally, the task is added to heap H_R (line 12). If no idle processor is available (line 13), the procedure checks whether the task is critical, *i.e.* if it has zero laxity (line 14). If the task is not critical, its *key* is set to the next time at which it will fire a *Z* event (line 16), and it is added to heap H_W (line 17) even though its laxity may be less than that of the currently executing tasks. Here, the *Non-Preemptability* policy plays its role in our algorithm; tasks with less laxity are able to advance their executions before the greedy ones reach zero laxity. This is the reason for the semi-greedy concept underlying the algorithm. On the other hand, if the task is critical (line 18), the procedure preempts the task with the maximum laxity for it. In line 19, the task with the maximum laxity is extracted from heap H_W. In line 20, the *key* of the preempted task (T_r) is updated to denote the next time at which it will reach zero laxity, and similarly, the *key* of task (T_{na}) is updated (line 22) to denote the time at which it will end execution. In lines 23–24, the processor *Id* of task (T_r) is retrieved and assigned to task (T_{na}), and accordingly, in (line 25), the task *Id* of the processor is changed to the *Id* of task (T_{na}). Finally, in lines 26–27, task (T_{na}) is added to heap H_R and task (T_r) is added to heap H_W.

4.4.5. *The* extractMaxLaxity *procedure*

The *extractMaxLaxity* is a helper procedure used by the *handleEorZEvents* procedure when a *Z* event is fired, to locate and extract the task with the maximum laxity from the running heap H_R as a prelude to preemption. Fig. 7 shows the algorithm of the *extractMaxLaxity* procedure. The procedure uses a *priority* queue that is ordered by decreasing laxity, *i.e.* a task with the maximum laxity will be at the front of the queue. The procedure uses two *while* loops to extract the task with the maximum laxity from heap H_R. The first *while* loop (lines 1–4) extracts all tasks from the running heap H_R (line 2) and adds them to the *priority* queue (*pq*) (line 3) one by one. Once this step is completed, the maximum laxity task will be at the front of the *priority* queue (*pq*), and consequently, it is extracted in line 5. In lines 6–9, the second *while* loop extracts the remaining tasks from the *priority* queue (*pq*) (line 7) and adds them back to the running heap H_R (line 8). After the second *while* loop is completed, the procedure returns the maximum laxity task (line 9) to the caller procedure.

4.5. *Example 1: USG versus fair scheduling algorithms*

In this example, we will use the taskset in Table 1 to compare the schedule generated by the LRE-TL and LLREF algorithms [3,6] with that generated by the proposed USG algorithm.

Fig. 8 shows the schedule generated by LRE-TL, while Fig. 9 shows the schedule generated by the proposed USG algorithm. We have considered the schedule for the interval [0,5), which corresponds to the first TL-plane in LRE-TL.

Owing to the fairness rule followed by LRE-TL, all tasks have to be preempted at the end of each TL-plane. Therefore, as shown in Fig. 8, the schedule generated by LRE-TL shows 8 preemptions and one migration for task T_6. There are many problems associated with the LRE-TL scheduling algorithm as well as other algorithms that follow the fairness rule in general. Next, we identify these problems clearly and show how the proposed USG algorithm overcomes them:

- The first problem is that sometimes, we have consecutive deadlines with only one unit of time, such as [14,15), [15,16), [16,17), [19,20), and [20,21). This will result in numerous preemptions and migrations. For example, for the first 14 TL-planes which are in the time interval [0,29), we have a total of 112 preemptions and 14 task migrations. In contrast, in the same time interval, the proposed algorithm shows only 4 preemptions that result in 4 task migrations.

n: Number of tasks. m: number of processors. H_W: Waiting heap of size n.	H_R: Running heap of size m. H_{NA}: Not active heap of size n. P: List of processors of size m.
1. *while* (H_{NA}.getMinimum().key == Tcur)	**15.** //add Tna to heapW
2. Task T_{na} = H_{NA}. extractMinimum ();	**16.** T_{na}.key = T_{na}.p-T_{na}.e + Tcur;
3. Task T_r;	**17.** H_W.insert(T_{na});
4. int z;	**18.** *else*//preempt one for T_{na}
5. //if H_R size is less than m then add Tna to heapR	**19.** T_r = extractMaxLaxity(H_R);
6. *if* (H_R.getSize()<m)	**20.** T_r.key = T_r.p - (T_r.key%T_r.p)+ Tcur;
7. //get any idle processor	**21.** //change T_r.key to laxity
8. int idleProcessor=idleProcSTK.pop();	**22.** T_{na}.key = T_{na}.e + Tcur;
9. T_{na}.procId = idleProcessor;	**23.** z = Tr.procId;
10. P[idleProcessor].taskId = T_{na}.id;	**24.** T_{na}.procId = z;
11. T_{na}.key = T_{na}.e + Tcur;	**25.** P[z].taskId = Tna.id;
12. H_R.insert(T_{na});	**26.** H_R.insert(T_{na});
13. *else* //H_R is full, check to see if T_{na} should go for // heapW or preempt one for it	**27.** H_W.insert(T_r);
14. *if* (T_{na}.e<T_{na}.p)	**28.** *end if*

Fig. 6. The *handleAEvent* procedure.

n: number of tasks. m: number of processors.	H_R: Running heap of size m. pq: Priority queue of size m.
1. *while* (H_R.getSize()>0)	**6.** *while* (!pq.isEmpty())
2. Task Tb = extractMinimum	**7.** Task Tb = pq.poll();
3. pq.offer(Tb);	**8.** H_R.insert(Tb);
4. *end while*	**9.** *end while*
5. Task TmaxLaxity = pq.poll();	**10.** *return* TmaxLaxity;

Fig. 7. The *extractMaxLaxity* procedure.

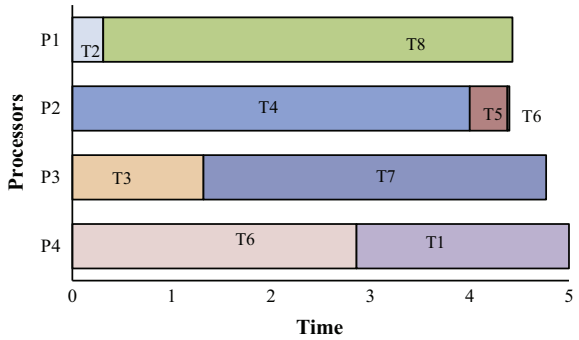

Fig. 8. The schedule generated by LRE-TL for the first TL-plane in the interval [0,5).

- The second problem is that some tasks such as task T_2, which has execution requirements of 1 and a period of 16, as well as task T_5, which has execution requirements of 2 and a period of 26, are forced to execute in each TL-plane although they can wait for 15 and 24 units of time, respectively, before they become critical. Surely, this results from obeying the fairness rule. In our case, no such task is executed unless it becomes critical; thus, our algorithm produces far fewer preemptions and migrations.
- The third problem is that the initialization procedure has to be invoked at the beginning of each TL-plane to initialize the local remaining execution times for the active tasks and populate both heaps H_B and H_C. In our case, the initialization procedure is called once, before the scheduler starts running, to populate both heap H_R and heap H_W, and it is never called again. This makes our scheduler execution lighter than that of LRE-TL.
- The last problem is that, when calculating the local remaining executions of tasks within each TL-plane, we usually get the values as floating-point numbers, which may be periodic ones; therefore, we have to round off the numbers as required, and this may lead to the loss of some execution units for some tasks. In this case, the scheduler timer should be fine-grained so that it can fire the scheduling events at the right time, and surely, this will extend the execution time

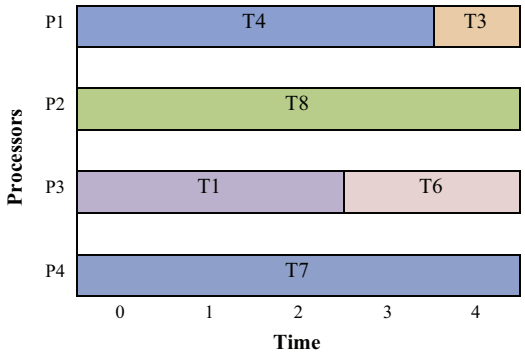

Fig. 9. The schedule generated by the proposed USG algorithm for the interval [0,5).

of the scheduler because it has to loop many times to reach the end of the TL-plane (e.g. if the rounding mode is set to 2 digits after the decimal point, we have to adjust the scheduler timer to start from 0.00 with steps of 0.01, and in order to reach the end of the first TL-plane in the above example, which is 5, the scheduler has to run 500 hundred times from 0.00 to 4.99).

4.6. USG versus greedy schedulers

Next, two examples are presented to show the difference between USG and greedy schedulers.

4.6.1. Example 2

In this example, we show how USG schedules the feasible taskset $\{T_1 = (9,10), T_2 = (9,10), T_3 = (7,40)\}$, for 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], and [29,30] before it can be prioritized over T_1 and T_2 at time $t = 36$, when it will have zero laxity. Therefore, T_3 will be executed to completion before its deadline at time $t = 40$. On the other hand, both T_1 and T_2 still have 3 remaining units each, *i.e.* a total of 6 units; however, only 3 units of time are remaining on the other processor before their deadline at $t = 40$. Thus, either T_1 or T_2 will miss its deadline. The schedule generated by LLF and EDZL is shown in Fig. 10.

In the following scenario, we explain how USG can successfully schedule the above-mentioned taskset. As explained previously, when the scheduler starts, it will first call the *initialize* procedure. The *initialize* procedure will then populate heap H_R with tasks T_1 and T_2 because they have the least laxity. Then, task T_3 will be added to heap H_W. When the initialize procedure completes execution, the system checks for any of the three types of events, *E*, *Z*, or *A*. As we can see from the schedule shown in Fig. 11, T_1 and T_2 continue to execute until time $t = 9$; then, both of them will fire an *End E* event. As a result, T_3 will be selected for execution on one processor and the other one will be set to idle. T_3 will continue to execute until time $t = 10$, at which both T_1 and T_2 will arrive. Thus, T_1 will be assigned to the idle processor; however, because T_2 is not critical, *i.e.* its laxity is not zero, it will be added to heap H_W even though it has less laxity than T_3. Thus, T_3 would have a chance to make progress in its execution by one more unit of time before T_2 reaches zero laxity and preempts it. This clearly explains the semi-greedy property of the algorithm. The scenario then continues as shown in Fig. 11, and a feasible schedule is generated.

4.6.2. Example 3

In this example, we show another simple case which greedy schedulers fail to schedule; however, it is schedulable by USG. Consider the feasible taskset $T = \{T_1(2,3), T_2(2,3), T_3(6,10)\}$ on two processors.

Fig. 12 shows the schedule generated by LLF. It can be clearly seen that at time $t = 8$, task T_2 reaches zero laxity. However, at the same time, both T_3 and T_1 are having zero laxity; hence, one of the three tasks will miss its deadline.

Fig. 13 shows the schedule generated by EDZL. It can be clearly seen that at time $t = 8$, task T_2 reaches zero laxity. However, at the same time, both T_3 and T_1 are having zero laxity; hence, one of the three tasks will miss its deadline.

Fig. 14 shows a successful schedule for the tasks generated by USG, and all the tasks meet their respective deadlines.

5. Deadline misses under USG

In this section, we show how and when USG can miss deadlines and fail to schedule tasksets.

Theorem 1. *Let $T = \{T_1, T_2, \ldots, T_n\}$ be a schedulable taskset, i.e. $U \leq m$ and $U_{\max} \leq 1$. Then, USG can fail to schedule such a taskset iff the number of tasks that reach zero laxity at the same time is $> m$.*

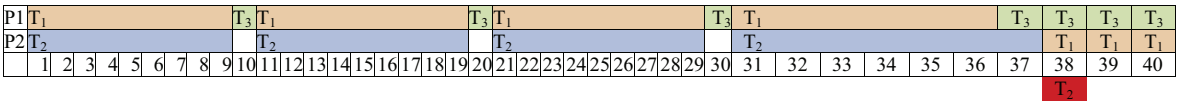

Fig. 10. LLF and EDZL schedule for the taskset of Example 2.

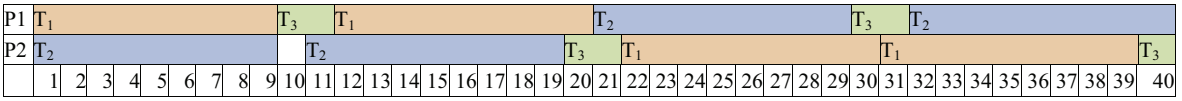

Fig. 11. USG schedule for the taskset of Example 2.

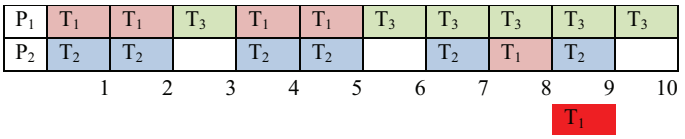

Fig. 12. Schedule generated by LLF for the taskset of Example 3.

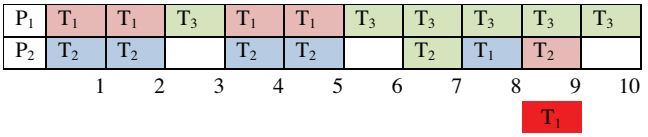

Fig. 13. Schedule generated by EDZL for the taskset of Example 3.

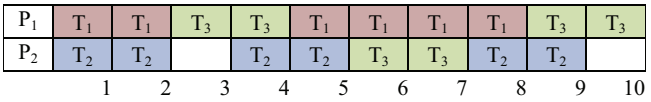

Fig. 14. Schedule generated by USG for the taskset of Example 3.

Proof. Suppose that task T_x missed its deadline at time $t = x$; then, the following conditions hold for task T_x:

- T_x has zero laxity at time $t = x$.
- T_x has been preempted by another task T_i that also reaches zero laxity at the same time $t = x$.
- T_x still has remaining execution units.

USG preempts the task with the maximum laxity when a task fires a Zero-Laxity event; this means that when T_x is preempted at time $t = x$, it is the task with the maximum laxity. Now, if the task with the maximum laxity is having zero laxity, all the remaining tasks in heap H_R should also have zero laxity. Thus, the number of tasks that reach zero laxity at time $t = x$ would be at least $m + 1$. □

Theorem 2. *According to Theorem 1, the total remaining utilization when a task T_i misses a deadline at time $t = x$ is $R_x > U$.*

Proof. From Theorem 1, we know that when a task misses a deadline at time $t = x$, we would have more than m tasks having zero laxity at the same time $t = x$. The task that reaches zero laxity would have remaining utilization

$$r_{i,t} = \frac{e_{i,t}}{n_i(t)\,p_i - t} = 1 \tag{12}$$

where

$$n_i(t) = \left\lceil \frac{t}{p_i} \right\rceil \tag{13}$$

Then, the total remaining utilization at time $t = x$ is given by

$$R_x = m + 1 + \sum_{i=m+2}^{n} r_{i,x} > m \tag{14}$$

Because T is a schedulable taskset, *i.e.* $U \leqslant m$, the remaining utilization at any time t, R_t, should never exceed the total utilization U which is less than or equal to m. However, from Eq. (14), we get $R_x > m$, which means that $R_x > U$.

It is clear that Theorem 2 follows from Theorem 1, and in this case, the taskset T is deemed unschedulable by USG. In Section 7.3, we discuss the schedulability of USG and show the percentage of tasksets that are deemed schedulable and unschedulable by USG for different task utilizations. □

6. The complexity of USG

The complexity, *i.e.* the running time analysis, of USG depends on the event handler procedure being called. In the following subsections, we discuss the running time of each of the event handler procedures in USG. Table 4 summarizes the complexity of the USG procedures.

6.1. The complexity of the initialize procedure

The USG *initialize* procedure has two loops. The first loop iterates n times, *i.e.* it takes $O(n)$ time to add all tasks to heap H_W. Then, the first m tasks are extracted from heap H_W and added to heap H_R. Because the insertion operation in a heap takes $O(\log n)$ time and extracting the minimum element also takes $O(\log n)$ time in a heap of size n [25], the complexity of the *initialize* procedure is

$$\sum_{i=1}^{n} \log n + \sum_{i=1}^{m} (\log(n) + \log m) = O(n \log n) + O(m \log(n)) + O(m \log m) = O(n \log n) \tag{15}$$

The *initialize* procedure is called only one time, when the algorithm starts in order to initialize the heaps.

6.2. The complexity of the handleEorZEvents procedure

The *handleEorZEvent* procedure has two loops. The first loop handles E events, and the second loop handles Z events.

6.2.1. Handling an E event

Handling an E event requires the following operations:

- The completed task is extracted from heap H_R, which takes $O(\log m)$ time.
- If heap H_W is not empty, the minimum element is extracted and added to heap H_R, and this takes $O(\log(n-m)) + O(\log m)$ time.
- The completed task is added to the not active list, heap H_{NA}, which takes $O(\log n)$ time.

The worst-case scenario for this loop is when all tasks complete their executions at the same time, which means that the above operations will be executed m times, *i.e.* the size of heap H_R. Hence, the complexity of the first loop is

$$\sum_{i=1}^{m} (\log m + \log(n-m) + \log m + \log n) = O(2m \log m) + O(m \log(n-m)) + O(m \log n) = O(m \log n) \tag{16}$$

If the number of active tasks is less than m, heap H_W is empty and tasks are removed only from H_R, which takes $O(\alpha \log \alpha)$ time, where α is the number of active tasks.

Table 4
Complexity of USG algorithm.

Procedure		Complexity
initialize		$O(n \log n)$
handleEorZEvent	*Handling E event*	$O(m \log n)$
	Handling Z event	$O(m^2 \log m)$
extractMaxLaxity		$O(m \log m)$
handleAEvent		$O(n \log n)$

6.2.2. Handling a Z event

Handling a *Z* event requires the following operations:

- Extracting the zero laxity task from heap H_W, which takes $O(\log(n-m))$ time.
- Extracting the maximum laxity task from heap H_R, which takes $O(m \log m)$ time.
- Adding the zero laxity task to heap H_R, which takes $O(\log m)$ time.
- Adding the maximum laxity task to heap H_W, which takes $O(\log(n-m))$ time.

The worst-case scenario for this loop is when m tasks reach zero laxity at the same time (if more than m tasks reach zero laxity at the same time, the taskset is unschedulable). Hence, the complexity of the second loop is

$$\sum_{i=1}^{m}(\log(n-m)+m\log m+\log m+\log(n-m)) = O(m\log(n-m))+O(m^2\log m)+O(m\log m)+O(m\log(n-m)) = O(m^2\log m) \quad (17)$$

If the number of active tasks is less than m, this loop will not be executed because heap H_W will be empty.

Note that because the maximum laxity task is removed when a Zero-Laxity event is fired, the complexity of this procedure increases up to $O(m^2\log m)$. Instead, we can remove the task with the minimum remaining work, which takes $O(\log m)$ time, because we only need to remove the minimum of heap H_R. In this case, the complexity of this procedure will be reduced to either $O(m\log(n-m))$ if $m < n/2$ or $O(m\log m)$ if $m \geqslant n/2$. However, as shown in Section 7.6, removing the task with the maximum laxity not only reduces the number of task preemptions and migrations significantly but also increases the schedulability of USG.

6.3. The complexity of the extractMaxLaxity *procedure*

This procedure is called in order to extract the task with the maximum laxity from heap H_R when a Zero-Laxity event is fired. However, the value of the *key* in heap H_R represents the time at which the task will end execution and not the laxity. Hence, removing the task with the maximum laxity requires further processing. The procedure uses a priority queue ordered by decreasing laxity to determine the task with the maximum laxity. For this purpose, the following operations are required:

- All tasks are extracted from heap H_R and added to the priority queue. Extracting a task from heap H_R takes $O(\log m)$ time, while adding it to the priority queue takes $O(\log m)$ time, which is the same as the time taken to add a task to a heap, where m in this case is the size of the priority queue.
- Extracting the top (peak) of the priority queue, *i.e.* the maximum laxity task, which takes $O(\log m)$ time.
- Extracting all remaining tasks from the priority queue, which takes $O((\mathrm{m}-1)\log(m-1))$ time, and adding them back to heap H_R, which also takes the same time, *i.e.* $O((\mathrm{m}-1)\log(m-1))$.

The worst-case scenario for this loop is when there are m tasks running when the Zero-Laxity event occurs. Hence, the complexity of this procedure is

$$\sum_{i=1}^{m}(\log m+\log m)+\log m+\sum_{i=1}^{m-1}(\log(m-1)+\log(m-1)) = O(2m\log m)+O(2(m-1)\log(m-1))+\log m = O(m\log m) \quad (18)$$

6.4. The complexity of the handleAEvent *procedure*

The *handleAEvent* procedure has a single loop that checks for and handles *A* events. Handling an *A* event requires the following operations:

- Extracting the arrived task from heap H_{NA}, which takes $O(\log n)$ time.
- Adding the arrived task to either heap H_R, which takes $O(\log m)$ time, or heap H_W, which takes $O(\log(n-m))$ time.

The worst-case scenario for this loop is when all tasks on the system arrive at the same time, which means that the loop will be executed n times. Thus, the complexity of this procedure is

$$\sum_{i=1}^{n}(\log n+\log m+\log(n-m)) = O(n\log n)+O(n\log m)+O(n\log(n-m)) = O(n\log n) \quad (19)$$

7. Results and discussion

To test the performance of USG in terms of scheduling overheads as well as schedulability, we used a standard procedure for generating the real-time tasksets. This procedure has been used in many recent works [2,7,19]. The data (tasksets) has to be generated randomly, as it is generally not easy to obtain real-time data for multiprocessors, especially data with hard real-time constraints. This is attributable to at least three factors. First, hard real-time systems are normally targeted at a specific hardware platform with specific sensors and actuators, and hence, it is difficult to closely regenerate the data in a lab setting. Second, embedded and real-time systems are usually made privately and rarely shared with researchers; these systems are considered as trade secrets. Third, existing real-time data are yet to fully exploit multiprocessors, as the adoption of multicore systems in the embedded and real-time industry remains in its nascent stages [26].

The task periods p_i have been generated using uniform integer distribution, a random number distribution that produces integer values according to the probability mass function given by [27]

$$P(i|a,b) = \frac{1}{b-a+1}, a \le i \le b \tag{20}$$

This distribution produces random integers in the range $[a,b]$, where each possible value has an equal likelihood of being produced.

To generate the task execution times e_i, first, a random real number, x, is uniformly generated in the range $(0,1]$ using uniform real distribution. Then, the task execution time is calculated as

$$e_i = \lfloor x \times p_i \rfloor \tag{21}$$

The generated tasksets are of size 4, 8, 16, 32, and 64 tasks to be executed on 2, 4, 8, 16, and 32 processors, respectively, *i.e.* the number of tasks is twice the number of processors ($n = 2m$), as in [2,7,19]. For each taskset, we have generated two sets (groups), each with 100,000 samples. The first set is generated with full utilization $\sum u_i = m$, while the second is generated with random utilization $\sum u_i \le m$. The task periods as well as worst-case execution requirements were uniformly chosen in the period $[1,100]$.

For example, to generate a taskset with 4 tasks to be executed on 2 processors, the scenario proceeds as follows. First, we start by initializing the total utilization of the taskset, U, to zero, and then, we generate two random numbers r_1, r_2 using uniform integer distribution where $r_1 \leqslant r_2$. Here, r_1 represents the worst-case execution requirement (e_i) for the first task, while r_2 represents its period (p_i). Then, we calculate the utilization u_i of this task and add it to the total utilization U of the taskset. This process is repeated until we reach the number of tasks that we need to generate, which is 4 in this case. On completion, we test whether the total utilization of the tasks is equal to the number of processors, *i.e.* 2 in this case. If the total utilization is equal to the number of the processors, we save the current taskset and proceed to generate the next one. Otherwise, we simply ignore the taskset and restart the process to get a new taskset. Two procedures have been used to generate the tasksets: the *main* procedure and the *generate* procedure. Next, we briefly discuss each of them.

7.1. USG scheduling overhead

The scheduling overhead results of USG in terms of task preemptions and migrations were encouraging. In the following subsections, we discuss and compare these results with those of state-of-the-art algorithms.

7.1.1. Preemption overhead

Fig. 15 shows compares USG with state-of-the-art optimal real-time multiprocessor scheduling algorithms in terms of the average number of preemptions produced per job. It can be clearly seen that USG produced very few preemptions per job. For example, the average number of preemptions per job produced by USG on 2 processors was only 0.24, which was far lower than that produced by RUN (1.25), the state-of-the-art algorithm in this field. On 4 processors, USG produced an average of 0.21 preemptions per job versus 1.25 produced by RUN. On 8 processors, USG produced an average of 0.11 preemptions per job versus 1.6 produced by RUN. On 16 processors, USG produced an average of 0.06 preemptions per job versus 1.9 produced by RUN. Lastly, on 32 processors, USG produced an average of 0.024 preemptions per job versus 1.9 produced by RUN. An interesting observation made in Fig. 15 is that the average number of preemptions per job produced by USG decreases as the number of processors increases, as opposed to the case of the other algorithms. This is because the other algorithms enforce artificial deadlines for tasks in order to maintain the optimality of the schedule, *i.e.* a task is subject to some preemptions even before it reaches its actual deadline, not because of criticality, *i.e.* a waiting task becoming critical, but rather to maintain optimality. However, in USG, no such artificial deadlines are applied because the fairness is totally relaxed. Hence, unless a waiting task becomes critical, *i.e.* reaches zero laxity, no preemptions are forced. Therefore, USG has very few preemption overheads as compared to the state-of-the-art algorithms.

7.1.2. Migration overhead

Fig. 16 compares USG with state-of-the-art optimal real-time multiprocessor scheduling algorithms in terms of the average number of migrations per job. It can be clearly seen that USG recorded the fewest migrations per job on average. For example, the average number of migrations per job produced by USG on 2 processors was only 0.24, which was far lower

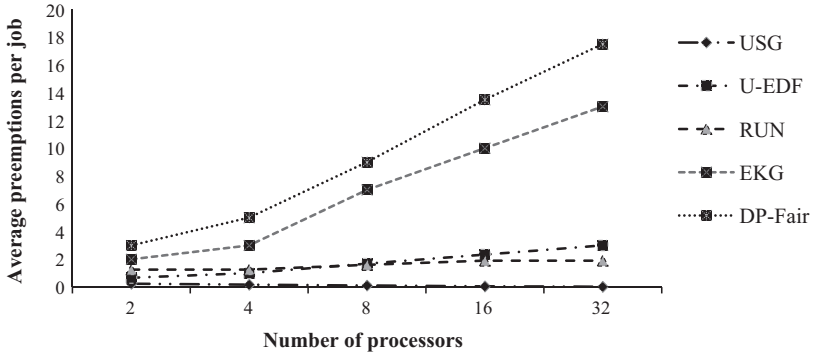

Fig. 15. Average preemptions per job: USG vs. state-of-the-art algorithms ($U = m$).

than that produced by RUN (0.8), the state-of-the-art algorithm in this field. On 4 processors, USG produced an average of 0.21 migrations per job versus 1.25 produced by RUN. On 8 processors, USG produced an average of 0.11 migrations per job versus 1.6 produced by RUN. On 16 processors, USG produced an average of 0.06 migrations per job versus 2.1 produced by RUN. Lastly, on 32 processors, USG produced an average of 0.024 migrations per job versus 2.35 produced by RUN. This is because USG never preempts a task unless it is absolutely necessarily, *i.e.* when a waiting task reaches zero laxity, as mentioned previously. Hence, when a waiting task reaches zero laxity, one of the executing tasks, *i.e.* the one with the maximum laxity, has to be preempted to allow the execution of the critical task. The execution of the preempted task is then continued later in a different processor and it is considered as a migratory task. To reduce the number of preemptions and migrations, USG uses a global job queue ordered by increasing laxity. Thus, tasks with the least laxity, which have a higher probability of firing Zero-Laxity events (resulting in task preemptions and migrations), are always scheduled for execution first. Accordingly, when the number of resources, *i.e.* processors, increases, many of the least laxity tasks have the opportunity to execute first, and similarly, many of them are expected to release their processors before the waiting tasks reach zero laxity. The number of Zero-Laxity events is eventually reduced, and accordingly, task preemptions and migrations are also reduced.

7.2. USG deadline misses

Because USG has not been optimized thus far, we conducted extensive simulations to show the average number of deadlines that USG can miss per job. In the following subsections, we show the results of these simulations under both full and random utilizations.

7.2.1. Deadline misses under full utilization

Fig. 17 shows the average number of deadline misses per job for USG, EDZL, and G-EDF using tasksets that are generated with full utilization, *i.e.* $\sum u_i = m$. It can be clearly seen that USG misses very few deadlines per job on average—sufficiently few to be tolerated in view of the considerable reduction achieved in task preemptions and migrations. For example, as shown in Fig. 17, the average number of deadlines that USG missed per job on 2 processors was 0.016, which was far lower than that missed by EDZL (0.196) and G-EDF (0.399). On 4 processors, USG missed an average of 0.000415 deadlines per job versus 0.195 missed by EDZL. On 8 processors, USG missed an average of 0.000000625 deadlines per job versus 0.178 by EDZL. On 16 as well as 32 processors, USG did not miss any deadlines, whereas EDZL missed an average of 0.166 and 0.159 deadlines, respectively.

An important observation that can be clearly made in Fig. 17 is that as the number of processors increases, the average number of deadlines per job missed by USG decreases. This can be explained by two factors. First, because USG uses a global job queue that is ordered by increasing laxity, the least laxity tasks are always considered for execution first. On the other hand, EDZL and G-EDF order the waiting queue according to the earliest deadlines (recall that laxity is based on both the deadline and the remaining work). Second, USG always preempts the task with the maximum laxity whenever a Zero-Laxity event is fired, unlike EDZL and G-EDF, which preempt the task with the maximum deadline. It is obvious that the availability of a running task with greater laxity reduces the probability of missing the deadline of the preempted task, which in turn contributes towards the schedulability of the taskset. Thus, an increase in the number of processors means that more tasks can end their work before others reach zero laxity. This indicates the availability of a running task to be preempted with sufficient laxity for the schedule to run smoothly without missing a deadline.

7.2.2. Deadline misses under random utilizations

Fig. 18 shows the average number of deadline misses per job for USG, EDZL, and G-EDF for tasksets generated with random utilizations, *i.e.* $\sum u_i \leq m$. It can be clearly seen that the average number of deadlines that USG misses per job is

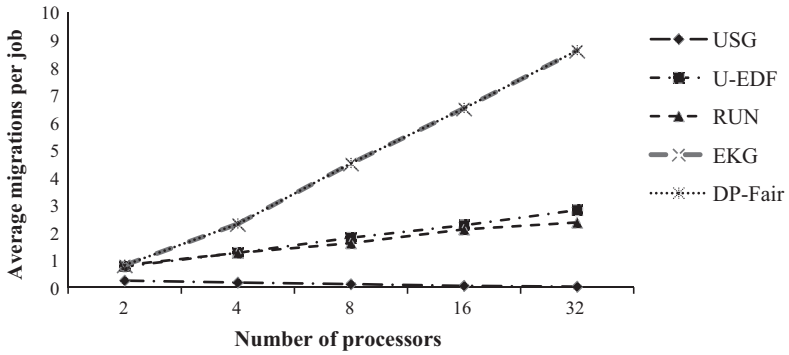

Fig. 16. Average migrations per job: USG vs. state-of-the-art algorithms ($U = m$).

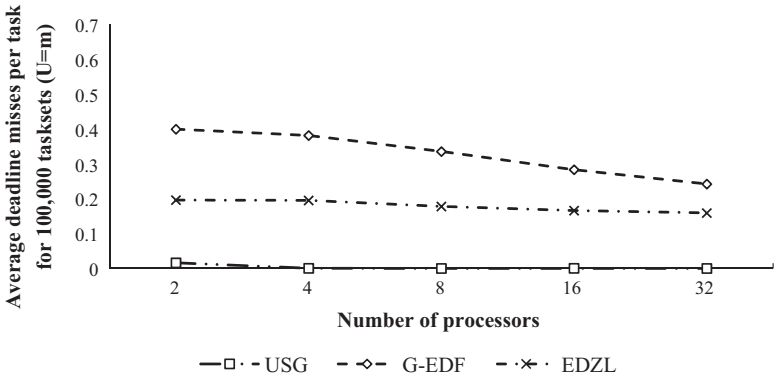

Fig. 17. Average deadline misses per job for USG, EDZL, and G-EDF ($U = m$).

significantly lower than that shown in Fig. 17. This is because more laxity is now available for USG to utilize, because the tasksets are generated with random utilization. For example, the average number of deadlines that USG missed per job on 2 processors was 0.0046, which was far lower than that missed by EDZL (0.094) and G-EDF (0.358). On 4 processors, USG missed an average of 0.000094 deadlines per job versus 0.138 missed by EDZL. On 8 processors, USG missed an average of 0.00000063 deadlines per job versus 0.150 missed by EDZL. On 16 as well as 32 processors, USG did not miss any deadlines, whereas EDZL missed an average of 0.1519 and 0.152 deadlines, respectively.

7.3. USG schedulability

In the following subsections, we present further simulation results to show the percentage of scheduled and unscheduled tasksets out of 100,000 tasksets for each group of the generated tasksets under full and random utilizations.

7.3.1. Schedulability under full utilization

Fig. 19 shows the percentages of scheduled tasksets out of 100,000 tasksets for USG, EDZL, and G-EDF for all groups of the generated tasksets under full utilization, *i.e.* $\sum u_i = m$. It can be clearly seen that USG achieves high levels of schedulability. For example, on 2 processors, USG was able to schedule 94.24% of the 100,000 tasksets successfully, whereas EDZL reported only 23% of the tasksets as schedulable. On 4 processors, USG reported 97.74% of the tasksets as schedulable versus only 1% reported as schedulable by EDZL. On 8 processors, USG reported 99.999% of the tasksets as schedulable, and on 16 and 32 processors, all the tasksets were reported as schedulable. On the other hand, EDZL failed to schedule any taskset on 8, 16, and 32 processors.

The low performance of both G-EDF and EDZL corresponds to the fact that real-time scheduling algorithms extended from their uniprocessor counterparts, such as Earliest Deadline First (EDF) and Rate Monotonic (RM) , are known to perform very well on uniprocessor systems, but they do not perform well on multiprocessors [28]. This can be clearly seen from the performance of G-EDF, which was able to schedule only 30% of the tasksets generated with random utilizations on 2 processors, and only 3% on 4 processors. On 8, 16, and 32 processors, G-EDF was not able to schedule any taskset. The performance of G-EDF is further degraded when the tasksets are generated with full utilization. On the other hand, EDZL, which is an enhanced version of G-EDF, performed better and was able to schedule 63% of the tasksets generated with random utilization

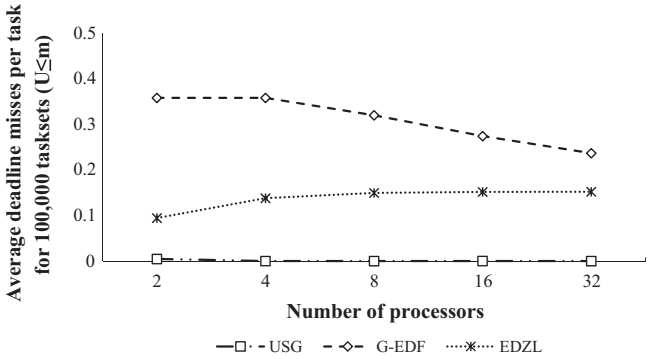

Fig. 18. Average deadline misses per job for USG, EDZL, and G-EDF ($U \leqslant m$).

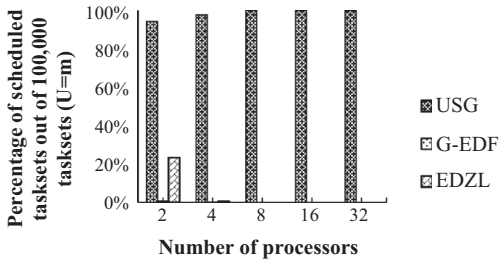

Fig. 19. Percentage of scheduled tasksets for USG, EDZL, and G-EDF ($U = m$).

on 2 processors. However, because of the EDF policy extended from G-EDF to EDZL, the performance of EDZL is degraded as the number of processors increases. Its performance is further degraded for tasksets generated with full utilization.

From these results, we believe that the levels of schedulability achieved by USG are comparable with optimal scheduling algorithms. The high schedulability is due to the three integrated policies of USG: Non-Preemptability, Zero-Laxity, and preemption of the maximum laxity task when a critical (Zero-Laxity) event is fired.

For greater clarity, Fig. 20 shows the percentage of unschedulable tasksets for USG, EDZL, and G-EDF under full utilization.

7.3.2. Schedulability under random utilizations

Fig. 21 shows the schedulability of USG, EDZL, and G-EDF using tasksets generated with random utilizations, i.e. $\sum u_i \leq m$. In this case, USG was able to achieve higher schedulability levels as compared to the results obtained using tasksets generated with full utilization. For example, on 2 processors, USG achieved 98.38% schedulability (compared to 94.24% achieved with full utilization). On the other, hand EDZL achieved 63% schedulability. On 4 processors, USG achieved 99.93% schedulability, while EDZL achieved 18% schedulability. Further, on 8 processors, USG reported 99.999% of the tasksets as schedulable, while on 16 and 32 processors, all tasksets were reported as schedulable. On the other hand, EDZL was able to schedule 1% of the tasksets on 8 processors, while it failed to schedule any of them on 16 and 32 processors.

For greater clarity, Fig. 22 shows the percentage of unschedulable tasksets for USG, EDZL, and G-EDF under random utilizations.

7.4. USG complexity

The worst-case complexity of USG is when a task is being preempted in response to a Zero-Laxity (Z) event. In this case, the *extractMaxLaxity* procedure is called to remove the task with the maximum laxity. The complexity of this procedure is $m \log m$. However, if m tasks reaches zero laxity at the same time, *i.e.* they fire Z events at the same time, the complexity of handling these events will be $m^2 \log m$. The worst-case complexity of USG is less than that of U-EDF, which has a complexity of nm per task release, *i.e.* if n tasks are released at the same time, the complexity of U-EDF increases up to $n^2 m$, which is greater than $m^2 \log m$. The complexity of USG is also less than that of RUN, which has complexity $O(jn \log m)$, where j is the number of jobs; if j reaches n, the complexity of RUN will increase up to $O(n^2 \log m)$, which is greater than the complexity of USG.

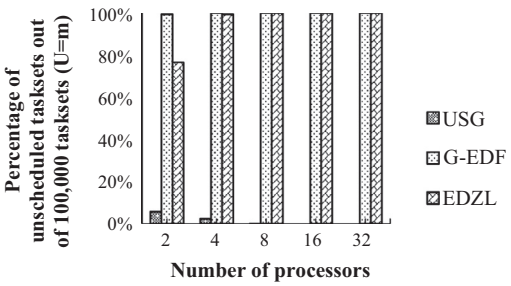

Fig. 20. Percentage of unscheduled tasksets for USG, EDZL, and G-EDF ($U = m$).

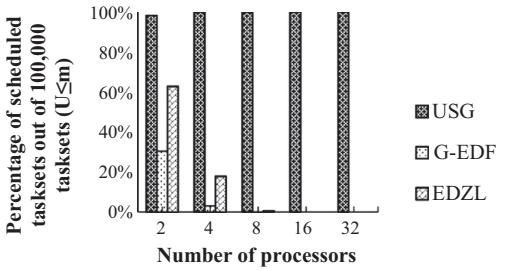

Fig. 21. Percentage of scheduled tasksets for USG, EDZL, and G-EDF ($U \leqslant m$).

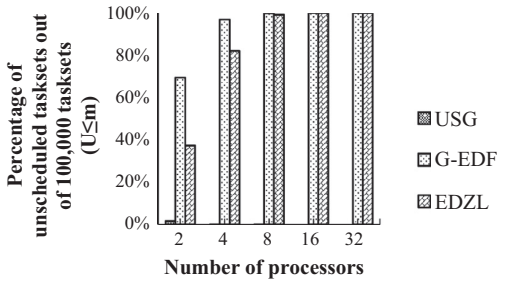

Fig. 22. Percentage of unscheduled tasksets for USG, EDZL, and G-EDF ($U \leqslant m$).

7.5. Practical schedulability analysis of USG

We conducted further simulations to show exactly at what utilizations the USG algorithm starts to fail to schedule tasksets and how many tasksets are deemed unschedulable. For each set of processors, *i.e.* 2, 4, 8, 16, and 32, we generated 100,000 tasksets with utilizations that range in the periods $[(m-1).9-m]$, $[(m-1).8-(m-1).9)$, $[(m-1).7-(m-1).8)$, … and so on, until we reached the range beyond which the algorithm deems all tasksets schedulable. For example, for $m = 2$, we generated 100,000 tasksets with utilizations in the periods [1.9–2.0], [1.8–1.9), [1.7–1.8), [1.6–1.7), [1.5, 1.6), … and so on. For $m = 2$, when we reached the utilization period [1.6–1.7), the algorithm reported only one taskset as unschedulable, and for utilizations less than or equal to 1.5, all tasksets were reported as schedulable. Figs. 23–27 show the utilizations at which the schedulability starts to drop, as well as the percentage of the schedulability at the corresponding utilization for $m = 2$, 4, 8, 16, and 32, respectively. It can be observed from these figures that the schedulability increases with the number of processors. The justification for this is that whenever the number of processors increases, which means more tasks as well, the probability of finding a task with sufficient laxity to be preempted when a Zero-Laxity event is fired increases significantly. This in turn considerably reduces the probability of getting more than m tasks that reach zero laxity at the same time, at which point the algorithm fails. This is why the algorithm reported all tasks for $m = 16$ and $m = 32$ as schedulable. This also reinforces our claim that the algorithm really misses very few deadlines—sufficiently few to be tolerated.

7.6. Preemption in USG (minimum remaining work vs. maximum laxity)

In Section 7.4, we discussed the complexity of USG. We showed that preempting the task with the maximum laxity when a Zero-Laxity event is fired increases the complexity of the *handleEorZEvents* procedure up to $O(m^2 \log m)$. On the

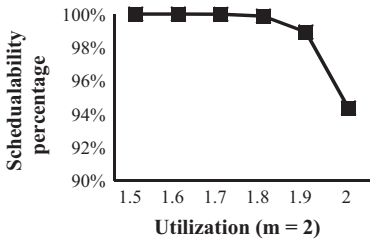

Fig. 23. USG schedulability for $m = 2$.

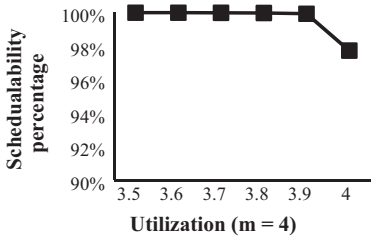

Fig. 24. USG schedulability for $m = 4$.

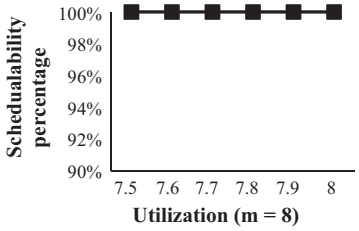

Fig. 25. USG schedulability for $m = 8$.

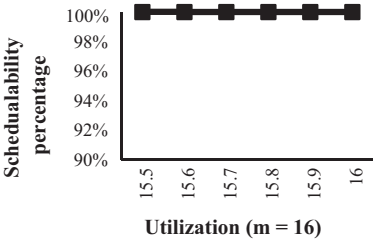

Fig. 26. USG schedulability for $m = 16$.

other hand, we showed that preempting the task with the minimum remaining work reduces this complexity to either $O(m\log(n-m))$ if $m < n/2$ or $O(m\log m)$ if $m \geqslant n/2$. However, preempting the task with the maximum laxity significantly improved the performance of USG in terms of scheduling overheads as well as schedulability. Fig. 28 compares the scheduling overheads in terms of task preemptions and migrations for USG when preempting the task with the minimum remaining work (USG_{Min_Work}) versus the one with the maximum laxity (USG_{Max_Laxity}). It can be clearly seen from that preempting the task with the maximum laxity significantly reduces the average number of task preemptions and migrations per job.

Consider Figs. 29 and 30. Fig. 29 shows the percentage of scheduled tasks for USG when preempting the task with the minimum remaining work versus the one with the maximum laxity when $U = m$. Fig. 30 shows the percentage of scheduled tasks when $U \leqslant m$. It can be clearly seen from Fig. 29 that preempting the task with the maximum laxity significantly improves the schedulability of USG. For example, the schedulability of USG when preempting the task with the minimum remaining work on 2 processors and under full utilization, *i.e.* $U = m$, is around 52%. However, when preempting the task with the maximum laxity, the schedulability increases considerably, *i.e.* up to 94%. In Fig. 30, where $U \leqslant m$, the schedulability reached 98% when preempting the task with the maximum laxity, versus 80% when preempting the task with the minimum remaining work.

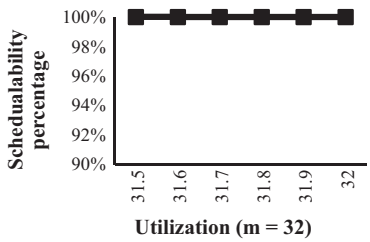

Fig. 27. USG schedulability for $m = 32$.

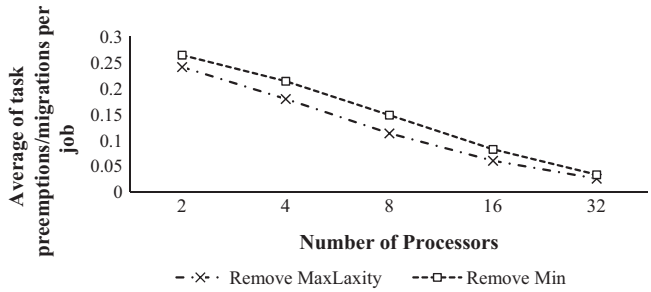

Fig. 28. Average number of job preemptions/migrations in USG when preempting the task with the minimum remaining work vs. the one with the maximum laxity.

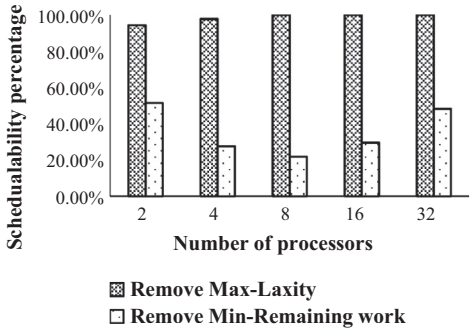

Fig. 29. Percentage of scheduled tasks when preempting the task with the minimum remaining work vs. the task with the maximum laxity in USG ($U = m$).

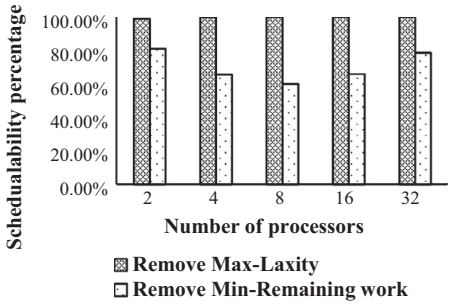

Fig. 30. Percentage of scheduled tasks when preempting the task with the minimum remaining work vs. the task with the maximum laxity in USG ($U \leqslant m$).

7.7. A brief summary of the obtained results

With regard to scheduling overheads in terms of task preemptions and migrations, the average number of task preemptions and migrations that USG produced per job was 0.24 and 0.024 for 2 and 32 processors, respectively, which was far

better than the results of the best-known algorithms in this field: U-EDF (0.67 and 3 preemptions; 0.75 and 2.8 migrations) and RUN (1.25 and 1.9 preemptions; 0.8 and 2.35 migrations).

Because USG has not been optimized thus far, it was shown to miss a few deadlines. However, these can be tolerated in view of the considerable reduction achieved in task preemptions and migrations. For example, the average number of deadlines that USG missed per job for 2 processors was 0.00456 when $U \leq m$ and 0.016 when $U = m$, which was far better than the results of the best-known non-optimal algorithms in this field: G-EDF (0.358 and 0.3997) and EDZL (0.094 and 0.196). Moreover, it was shown that USG schedulability increased with the number of processors. For example, USG achieved 100% schedulability for 16 and 32 processors.

The practical schedulability analysis of USG showed exactly at what utilizations USG started to miss deadlines. It was shown that for 2 processors, USG started to miss deadlines when the utilization increased beyond 1.5, *i.e.* when $U > 1.5$, which meant that USG can schedule tasksets without missing any deadline on 2 processors provided that $U \leq 1.5$. On 4 processors, USG started to miss deadlines when the utilization increased beyond 3.6. This meant that USG can schedule tasksets without missing any deadline on 4 processors provided that $U \leq 3.6$. The analysis also showed that on 8 processors, USG started to miss deadlines when the utilization reached 8, *i.e.* when $U \leq 8$. This meant that USG can schedule tasksets without missing any deadline on 8 processors provided that $U < 8$. With regard to 16 as well as 32 processors, USG was able to schedule all tasksets.

8. Conclusion

This paper presented an efficient real-time multiprocessor scheduling algorithm for reducing the scheduling overheads in terms of the number of task preemptions and migrations while maintaining high levels of schedulability. The key concept underlying the algorithm is the total relaxation of the fairness rule in order to reduce scheduling overheads. Even though this ensures the optimality of the algorithm, the overheads have a significant impact on the algorithm's practicality. The proposed algorithm uses a global job queue ordered by increasing laxity. Tasks with zero laxity have higher priority and they are always scheduled for execution immediately. If tasks with less laxity arrive while other tasks with greater laxity are executing, the former are instructed to wait until they reach zero laxity. Thereafter, they are scheduled for execution. Because USG has not been optimized thus far, it was shown to miss a few deadlines. However, these misses can be tolerated in view of the significant reduction in task preemptions and migrations. Hence, we believe that USG can be practically implemented in real-time applications that can tolerate a few deadline misses. This paper also presented a comparison between two preemption methods in USG, the minimum remaining work versus the maximum laxity, which showed that preempting the task with the maximum laxity outperforms preempting the task with the minimum remaining work in terms of reducing the scheduling overheads (task preemptions and migrations) and improving schedulability.

In the future, we plan to work towards optimizing USG and extending it to other real-time task models. We also plan to study the possibility of extending USG to supercomputing and cloud computing, as has been presented in [29,30].

References

[1] Liu JWS. Real-time systems. Prentice Hall; 2000.
[2] 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 (ECRTS); 2012. p. 13–23.
[3] Funk S. LRE-TL: an optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. Real-Time Syst 2010;46:332–59.
[4] Davis RI, Burns A. A survey of hard real-time scheduling for multiprocessor systems. ACM Comput Surv (CSUR) 2011;43:35.
[5] Baruah SK, Cohen NK, Plaxton CG, Varvel DA. Proportionate progress: a notion of fairness in resource allocation. Algorithmica 1996;15:600–25.
[6] Cho H, Ravindran B, Jensen ED. An optimal real-time scheduling algorithm for multiprocessors. In: 27th IEEE real-time systems symposium; 2006. p. 101–10.
[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] Bastoni A, Brandenburg BB, Anderson JH. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In: 31st IEEE real-time systems symposium; 2010. p. 14–24.
[9] Leung JY-T. A new algorithm for scheduling periodic, real-time tasks. Algorithmica 1989;4:209–19.
[10] Lee J, Easwaran A, Shin I. Laxity dynamics and LLF schedulability analysis on multiprocessor platforms. Real-Time Syst 2012;48:716–49.
[11] Lee J, Easwaran A, Shin I, Lee I. Zero-laxity based real-time multiprocessor scheduling. J Syst Software 2011;84:2324–33.
[12] Lee SK. On-line multiprocessor scheduling algorithms for real-time tasks. In: IEEE region 10's ninth annual international conference; 1994. p. 607–11.
[13] Baruah SK, Gehrke JE, Plaxton CG. Fast scheduling of periodic tasks on multiple resources. In: International symposium on parallel processing; 1995. p. 280–8.
[14] Chen S-Y, Hsueh C-W. Optimal dynamic-priority real-time scheduling algorithms for uniform multiprocessors. In: 29th IEEE real-time systems symposium; 2008. p. 147–56.
[15] Anderson JH, Srinivasan A. Early-release fair scheduling. In: 12th IEEE euromicro conference on real-time systems; 2000. p. 35–43.
[16] Zhu D, Qi X, Mossé D, Melhem R. An optimal boundary fair scheduling algorithm for multiprocessor real-time systems. J Parallel Distr Comput 2011;71:1411–25.
[17] Andersson B, Tovar E. Multiprocessor scheduling with few preemptions. In: 12th IEEE international conference on embedded and real-time computing systems and applications; 2006. p. 322–34.
[18] Nelissen G, Funk S, Goossens J. Reducing preemptions and migrations in EKG. In: 18th IEEE international conference on embedded and real-time computing systems and applications; 2012. p. 134–43.
[19] Regnier P, Lima G, Massa E, Levin G, Brandt S. Multiprocessor scheduling by reduction to uniprocessor: an original optimal approach. Real-Time Syst 2013:1–39.
[20] Massa E, Lima G, Regnier P, Levin G, Brandt S. OUTSTANDING PAPER: optimal and adaptive multiprocessor real-time scheduling: the quasi-partitioning approach. In: 26th IEEE euromicro conference on real-time systems; 2014. p. 291–300.

[21] Guan N, Yi W, Gu Z, Deng Q, Yu G. New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms. In: 29th IEEE real-time systems symposium; 2008. p. 137–46.
[22] Baruah SK. The non-preemptive scheduling of periodic tasks upon multiprocessors. Real-Time Syst 2006;32:9–20.
[23] Li W, Kavi K, Akl R. A non-preemptive scheduling algorithm for soft real-time systems. Comput Electr Eng 2007;33:12–29.
[24] Alhussian H, Zakaria N, Hussin FA. An efficient real-time multiprocessor scheduling algorithm. JCIT 2014;9:136–47.
[25] Weiss MA. Data structures and algorithms analysis in Java. 3rd ed. Prentice Hall; 2011.
[26] Brandenburg BB. Scheduling and locking in multiprocessor real-time operating systems. University of North Carolina; 2011.
[27] Kuipers L, Niederreiter H. Uniform distribution of sequences. Courier Dover Publications; 2012.
[28] Baker TP. Multiprocessor EDF and deadline monotonic schedulability analysis. In: 23rd IEEE real-time systems symposium; 2003. p. 120.
[29] Mahafzah BA, Jaradat BA. The hybrid dynamic parallel scheduling algorithm for load balancing on Chained-Cubic Tree interconnection networks. J Supercomput 2010;52:224–52.
[30] Gu X, Liu P, Yang M, Yang J, Li C, Yao Q. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng 2012;38:785–800.

Hitham Alhussian received his BSc and MSc in Computer Science from the School of Mathematical Sciences, Khartoum University, Sudan. He obtained his PhD from Universiti Teknologi Petronas, Malaysia. Currently, he is a Postdoctoral Researcher in the High-performance Computing Center in Universiti Teknologi Petronas. His main research interests include real-time Systems and parallel and distributed Systems.

Nordin Zakaria obtained his PhD from Universiti Sains Malaysia in 2007, working in the field of computer graphics. Since then, his areas of research have been diverse, encompassing high-performance computing, quantum computing, and motion capture and visualization. He is currently heading the High-performance Computing Center in Universiti Teknologi Petronas.

Ahmed Patel received MSc/PhD degrees from Trinity College, Dublin, specializing in packet-switched networks. Currently, he is a Full Professor at Jazan University, Saudi Arabia. His research covers networking, security, forensic computing, and distributed systems. He has authored 260 publications and co-authored several books. He is a member of the Editorial Advisory Board of International Journals and has participated in Irish, Malaysian, and European funded research projects.