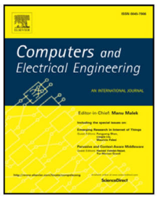
Computers and Electrical Engineering 50 (2016) 143–165

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/© 2015 Elsevier Ltd. All rights reserved.

(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 $\lfloor 3.2 \rfloor = 3$.