Journal of Fundamental and Applied Sciences

ISSN 1112-9867

Available online at http://www.jfas.info

Research Article

Special Issue

PRACTICAL PERFORMANCE ANALYSIS OF REAL-TIME MULTIPROCESSOR SCHEDULING ALGORITHMS

H. Alhussian[1,*], S. J. Abdulkadir[1], N. Zakaria[1], A. Patel[2] and A. Alzahrani[3]

[1]Universiti Technologi Petronas, 32610 Bandar Seri-Iskandar, Perak, Malaysia

[2]Vieira Computer Networks and Security Laboratory (LARCES), State University of Ceará (UECE), Fortaleza, Ceará, Brazil

[3]King Saud University, Riyadh, Saudi Arabia

Published online: 01 February 2018

ABSTRACT

This paper presents a practical performance analysing of two real-time multiprocessor scheduling algorithms, namely, Largest Remaining Execution-Time and Local Time Domain (LRE-TL) and Unfair Semi-Greedy (USG). The analysis is intended to reflect the behind-the-scene time overhead incurred by optimal real-time algorithms such as LRE-TL. The overhead is known to be capable of dismissing the actual optimality of such algorithms in practical applications. Here, the time overhead is measured in terms of the number of scheduler invocations and the time required by the scheduling event handlers. In the implementation of the proposed analysis method, the CPU profiler of Oracle JavaTM VisualVM was used to monitor the executions of LRE-TL and USG. The profiler measured the number of invocations of the scheduling event handlers for each algorithm and the total time required for all the invocations. The results revealed that USG outperformed LRE-TL on both measures, indicating that optimal algorithms may prove to be non-optimal in practical applications.

Keywords: Real-time; Multiprocessor; Scheduling;

Author Correspondence, e-mail: seddig.alhussian@utp.edu.my

doi: http://dx.doi.org/10.4314/jfas.v10i2s.6

1. INTRODUCTION

The correctness of real-time systems is not only determined by the logical results that it produces, but also the physical time at which the results are produced [1-8]. In order to fulfil the timing constraints of a real-time taskset (i.e. deadlines) in a real-time multiprocessor system, an optimal scheduling algorithm must be used. A scheduling algorithm is said to be optimal if it successfully schedules all the tasks of the system without missing any deadline provided that a feasible schedule exists for the tasks [3, 7, 9-11].

Optimal real-time multiprocessor scheduling algorithms consistently attain the highest processor utilisation, which is equal to the number of processors in the system. However, optimality is always achieved at the expense of scheduling overheads in terms of task pre-emptions and migrations, with significant impact on the practicality of the algorithm. This is because the optimality is mostly achieved by adherence to the fairness rule. Accordingly, tasks are forced to progress through their executions in time quanta or at the end of each time slice in a fluid schedule model of the deadlines of all the tasks in the system. The pre-emptions and migrations engender additional overheads, which must be added to the worst-case execution requirements of the tasks. However, theoretical studies of such algorithms ignore the overheads and the results of practical implementations thus fall short of the predictions.

To support this claim, we developed a method for the practical performance analysis of two real-time multiprocessor scheduling algorithms, namely, Local Remaining Execution-Time and Local Time Domain (LRE-TL) as an example of an optimal scheduling algorithm [12], and Unfair Semi-Greedy (USG), which is a non-optimal algorithm with a performance comparable to that of optimal algorithms [13].

The rest of this paper is organised as follows. Section 2 outlines the challenges that were addressed in the present study. Section 3 briefly reviews related works. Section 4 describes the proposed simulation method for evaluating the practical performance of algorithms. Section 5 presents and discusses the results of the implementation of the proposed method, and the conclusions of the study are finally outlined in Section 6.

2. SIGNIFICANCE OF THE PROPOSED COMPARATIVE PERFORMANCE ANALYSIS METHOD

As mentioned above, the proposed performance analysis method is aimed at extracting critical information regarding the time overhead of frequent scheduler invocations, and the time spent by the schedulers in processing the scheduling events. This is imperative because the overheads of most optimal real-time multiprocessor algorithms are ignored in theoretical studies, resulting in the non-delivery of the claimed optimality of such algorithms in practical applications.

3. LITERATURE REVIEW

Real-time systems are systems that maintain their correctness by outputting results within specific time constraints referred to as deadlines. Meeting the deadlines of a given real-time taskset cannot be achieved without the use of an optimal scheduling algorithm, which is defined in [14] as ‘one which may fail to meet a deadline only if no other one can’. In other words, an optimal scheduling algorithm successfully schedules all the tasks without missing any deadline for a schedulable taskset [9, 12]. Although there have been numerous proposals of optimal real-time multiprocessor scheduling algorithms over the last few years, many of them are impracticable. A recent study [13] showed that optimal scheduling algorithms have significant amounts of scheduling overheads in terms of the numbers of task pre-emptions and migrations, which substantially affect their practicality [9, 13]. An optimal real-time multiprocessor scheduling algorithm should therefore not only meet the deadline constraints of a schedulable taskset, but also have limited scheduling overheads to ensure practicability. The following subsections highlight some features of the most relevant recently reported real-time multiprocessor scheduling algorithms.

P-Fair [15]: This quantum-based algorithm is very strictly and known as Fair Scheduling. It strictly follows the fluid schedule model each time quantum and hence; forces each task to make progress in its execution every time quantum. Therefore, it is known to be computationally expensive and thus; causes a very high overhead in practice due to the frequent task preemptions and migrations.

Boundary Fair (BF) [16, 17]: This was the first algorithm to utilise the Deadline Partitioning (DP) technique. Like P-Fair, it uses quantum-based timing, but enforces fairness at the deadlines of tasks. It, however, also has the disadvantages of complexity and a high overhead in dealing with round-off issues regarding the amount of allocated work.

Largest Local Remaining Execution First (LLREF) [18]: This was the first quantum based DP-Fair algorithm. It performs unnecessary work and then resorts all the jobs during each scheduling event, completing those with the least laxity. It incorporates ‘T-L Plane’ visualisation.

LRE-TL [19]: Funk and Nadadur [19] proposed the LRE-TL algorithm as an optimal improved version of LLREF. The key feature of the algorithm is the elimination of the need to select tasks for execution based on the largest local remaining execution time within each time slice. In fact, any task with a remaining local execution time will do. This significantly decreases the number of migrations per time slice compared to LLREF. Funk and Nadadur [19] also showed how LRE-TL could be applied to sporadic tasksets and proved its optimality for sporadic tasksets with implicit deadlines. Funk [12] also extended LRE-TL to the support of sporadic tasks with unconstrained deadlines and proved the optimality of the algorithm for the application. A modified version of LRE-TL that significantly decreases the number of task migrations without affecting the optimality of the algorithm has also be developed [20].

USG [13]: USG is a semi-optimal real-time multiprocessor scheduling algorithm that decreases the number of task pre-emptions and migrations and enables the achievement of high levels of schedulability. The main idea behind USG algorithm is relaxing the fairness rule totally, and hence avoiding the enormous number of scheduling overheads in-terms of task preemptions and migrations it generates, though it ensures the optimality of the algorithm. The algorithm (USG) uses a global tasks queue ordered in an increasing laxity. The tasks with zero laxity have higher priority and are always scheduled for immediate execution. If new tasks with less laxity arrive while tasks with more laxity are being executing, the former are instructed to wait until they reach zero laxity, at which time they are considered for execution. However, as a penalty of relaxing the fairness rule totally, USG can sometimes miss a few deadlines, though these are sufficiently few to be tolerated to achieve the benefits of the significantly reduced task pre-emptions and migrations [13].

4. METHODOLOGY

The experimental simulation procedure for both LRE-TL and USG is shown in Figure 1. The process is as follows. Firstly, the required number of tasks, n, is set to 4, 8, 16, 32, or 64, and the required number of processors, m, is accordingly set to 2, 4, 8, 16, or 32; i.e., $n = 2m$. A while loop then begins to read the tasksets from the specified task file, one-by-one. It should be noted that the task file contains 100000 tasksets, and if n is set to 4, a set of 4 tasks would be read from the file at a time. The scheduler then sets the current time Tcur to zero and initialises the scheduling queues [13, 21]. The scheduler of each algorithm is executed for the first 1000 time units for tractability because the task periods are chosen within the range [1, 100]. Hence, the second while loop executes until the current time Tcur reaches 1000. As time progresses, the second while loop checks for scheduling events, and if an event is found, the corresponding handler is called upon to handle the fired event, with the statistical matrices updated accordingly. For example, if a Zero-Laxity (Z) event occurs, there would be a pre-emption and migration, and the corresponding pre-emption and migration matrices would be updated. The scheduler also checks whether the pre-empted task will miss its deadline; if so, the deadline matrix is additionally updated. Thereafter, the scheduler instructs each processor to execute its designated task, and the time is advanced. When the second while loop completes its execution, the outer while loop continues with the next tasksets, and this is repeated until all the tasksets in the file are simulated. When the outer while loop finally completes its execution, the collected results in the statistical matrices are printed.

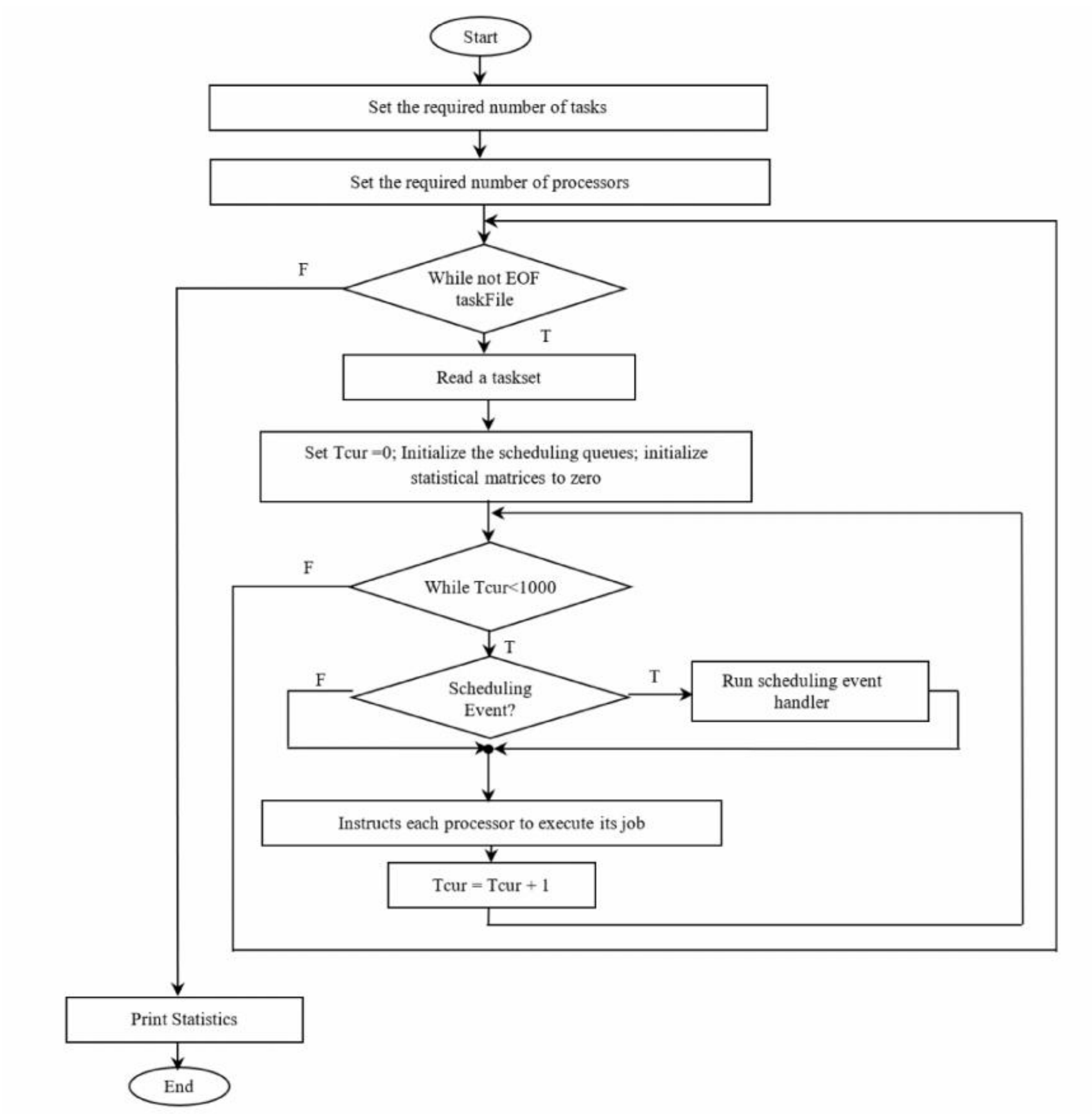

Start
Set the required number of tasks
Set the required number of processors
While not EOF taskFile
F
T
Read a taskset
Set Tcur =0; Initialize the scheduling queues; initialize statistical matrices to zero
While Tcur<1000
F
T
Scheduling Event?
F
T
Run scheduling event handler
Instructs each processor to execute its job
Tcur = Tcur + 1
Print Statistics
End

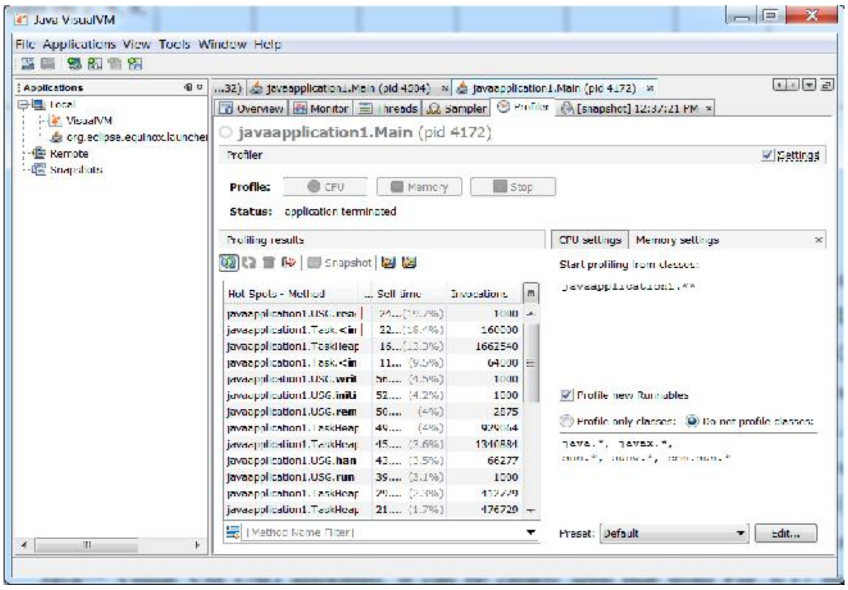
Fig.1. Experimental Simulation Procedure for LRE-TL and USG

5. RESULTS AND DISCUSSION

The CPU profiler of Oracle JavaTM VisualVM [1] was used to monitor the execution of USG against LRE-TL. Figure 2 shows the interface of Oracle JavaTM VisualVM. The two algorithms were executed on 2, 4, 8, 16, and 32 processors, respectively, using the same above-mentioned tasksets, where were generated with full utilisation. Both algorithms were run for the first 100 time units. For each group of generated tasksets, 1000 samples were executed. The CPU profiler of JavaTM VisualVM measured the number of invocations for each procedure (method) and algorithm, as well as the total time required for all the invocations.

Figures 3 and 4 respectively show the CPU profiler results for USG and LRE-TL on 2 processors. It can be clearly seen from the figures that the run procedures of both algorithms were executed 1000 times according to the number of tasksets used. USG invoked the initialise procedure 1000 times, implying once per taskset, as noted in Section 4. The total time required for all the invocations of USG was only 82.8 ms. In the case of LRE-TL, in which the initialisation procedure (*TL_Plane_Initialize*) was invoked at the beginning of each TL-plane, there were a total of 6943 invocations for the 1000 tasksets, implying about 7 invocations per taskset. A total time of 2182 ms was required for the invocations.

Further, the procedure for handling the scheduling events in USG (*handleEOrZEvens*) was invoked 8489 times for the 1000 tasksets, requiring 38.1 ms. The corresponding procedure in LRE-TL (*handleBOrCEvent*) was invoked 27749 times, requiring 266 ms. The trend was similar for the helper procedures (*removeMin* and *removeMaxLaxity* in USG, and *removeMin* in LRE-TL), which are used to insert and remove tasks from the scheduling queues (see Table 1).



javaapplication1.Main (pid 4172)

Profiler

Profile:

Status: application terminated

Profiling results

Hot Spots - Method	Self time	Invocations
javaapplication1.USG.rea	24... (19.7%)	1000
javaapplication1.Task.<in	22... (18.4%)	160000
javaapplication1.TaskHeap	16... (13.3%)	1662540
javaapplication1.Task.<in	11... (9.5%)	64000
javaapplication1.USG.writ	56... (4.5%)	1000
javaapplication1.USG.initi	52... (4.2%)	1000
javaapplication1.USG.rem	50... (4%)	2875
javaapplication1.TaskHeap	49... (4%)	929064
javaapplication1.TaskHeap	45... (3.6%)	1340884
javaapplication1.USG.han	43... (3.5%)	66277
javaapplication1.USG.run	39... (3.1%)	1000
javaapplication1.TaskHeap	29... (2.3%)	412229
javaapplication1.TaskHeap	21... (1.7%)	476729

Start profiling from classes:
javaapplication1.**

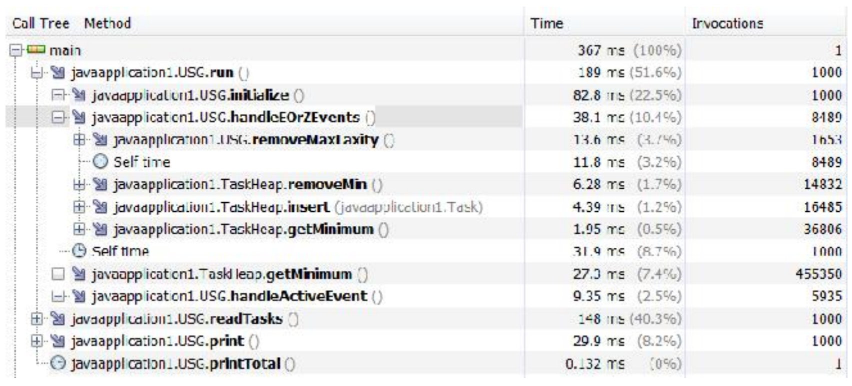
[✓] Profile new Runnables

Profile only classes: [● Do not profile classes:]

java.*, javax.*,
sun.*, sunw.*, com.sun.*

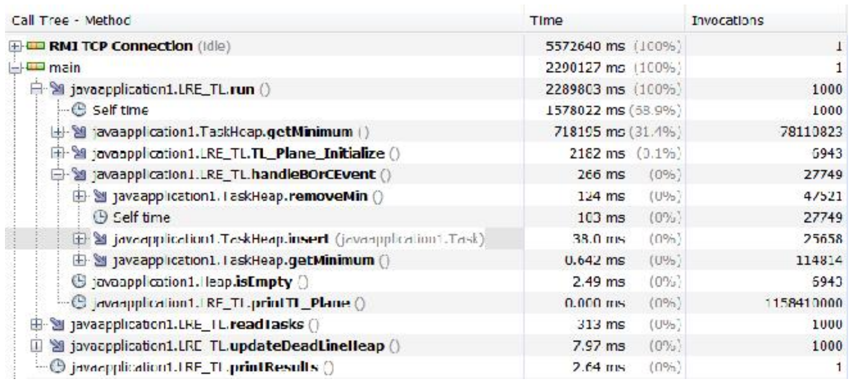
Preset: Default

Fig.2. Java VisualVM Interface



```
Call Tree   Method                                              Time                 Invocations
main                                                             367 ms  (100%)                1
  javaapplication1.USG.run ()                                    189 ms (51.6%)             1000
    javaapplication1.USG.initialize ()                          82.8 ms (22.5%)             1000
    javaapplication1.USG.handleEOrZEvents ()                    38.1 ms (10.4%)             8489
      javaapplication1.USG.removeMaxLaxity ()                   13.6 ms  (3.7%)             1653
      Self time                                                 11.8 ms  (3.2%)             8489
      javaapplication1.TaskHeap.removeMin ()                    6.28 ms  (1.7%)            14832
      javaapplication1.TaskHeap.insert (javaapplication1.Task)  4.39 ms  (1.2%)            16485
      javaapplication1.TaskHeap.getMinimum ()                   1.95 ms  (0.5%)            36806
    Self time                                                   31.9 ms  (8.7%)             1000
    javaapplication1.TaskHeap.getMinimum ()                     27.3 ms  (7.4%)           455350
    javaapplication1.USG.handleActiveEvent ()                   9.35 ms  (2.5%)             5935
  javaapplication1.USG.readTasks ()                              148 ms (40.3%)             1000
  javaapplication1.USG.print ()                                 29.9 ms  (8.2%)             1000
  javaapplication1.USG.printTotal ()                           0.132 ms    (0%)                1
```

Fig.3. CPU Profiler Results for USG on 2 Processors



```
Call Tree - Method                                                 Time                     Invocations
RMI TCP Connection (idle)                                          5572640 ms (100%)                  1
main                                                               2290127 ms (100%)                  1
  javaapplication1.LRE_TL.run ()                                   2289803 ms (100%)               1000
    Self time                                                      1578022 ms (58.9%)              1000
    javaapplication1.TaskHeap.getMinimum ()                         718195 ms (31.4%)          78110823
    javaapplication1.LRE_TL.TL_Plane_Initialize ()                    2182 ms  (0.1%)              6943
    javaapplication1.LRE_TL.handleBOrCEvent ()                         266 ms    (0%)             27749
      javaapplication1.TaskHeap.removeMin ()                           124 ms    (0%)             47521
      Self time                                                        103 ms    (0%)             27749
      javaapplication1.TaskHeap.insert (javaapplication1.Task)        38.0 ms    (0%)             25658
      javaapplication1.TaskHeap.getMinimum ()                        0.642 ms    (0%)            114814
    javaapplication1.Heap.isEmpty ()                                  2.49 ms    (0%)              6943
    javaapplication1.LRE_TL.printTL_Plane ()                         0.000 ms    (0%)        1158410000
  javaapplication1.LRE_TL.readTasks ()                                 313 ms    (0%)              1000
  javaapplication1.LRE_TL.updateDeadLineHeap ()                       7.97 ms    (0%)              1000
  javaapplication1.LRE_TL.printResults ()                             2.64 ms    (0%)                 1
```

Fig.4. CPU Profiler Results for LRE-TL on 2 Processors

Table 1 summarises and compares the CPU profiler results for USG and LRE-TL on 2 processors.

Table 1. CPU Profiler Results for USG and LRE-TL on 2 Processors

Procedure	Initialisation		Scheduling events		Removal		Insertion	
	No. of invocations	Time (ms)	No. of invocations	Time (ms)	No. of invocations	Time	No. of Invocations	Time (ms)
USG	1000	82.8	8489	38.1	1653 + 14832 = 16485	13.6 + 6.28 = 19.88	16485	4.39
LRE-TL	6943	2182	27749	266	47521	124	25658	38.0

The CPU profiler results for both USG and LRE-TL on 4 processors are summarised in Table 2. In this case, the total time required for all the invocations of the initialisation procedure in USG was only 99.1 ms. Conversely, the LRE-TL initialisation procedure (*TL_Plane_Initialise*) was invoked 12525 times for the 1000 tasksets, implying about 13 invocations per taskset, with a total time of 6993 ms required. Further, the procedure for handling the scheduling events in USG was invoked 15821 times for the 1000 tasksets, requiring a total time of 69.9 ms. The corresponding procedure in LRE-TL was invoked 100024 times, with a total time of 933 ms required.

Table 2. CPU profiler Results for USG and LRE-TL on 4 Processors

Procedure	Initialisation		Scheduling events		Removal		Insertion	
	No. of invocations	Time (ms)	No. of invocations	Time (ms)	No. of invocations	Time (ms)	No. of invocations	Time (ms)
USG	1000	99.1	15821	69.9	2695 + 30248 = 32943	34.5 + 12.2 = 46.7	32943	6.67
LRE-TL	12525	6993	100024	933	178168	592	106104	118

Table 3 summarises the results of the CPU profiler for both USG and LRE-TL on 8 processors. Here, the time spent on all the invocations of the initialisation procedure in USG was only 143 ms. conversely, the LRE-TL initialisation procedure (*TL_Plane_Initialise*) was invoked 22060 times for the 1000 tasksets, implying 23 invocations per taskset, with a total time of 36643 ms required. Further, the procedure for handling the scheduling events in USG was invoked 27292 times for the 1000 tasksets, requiring a total time of 131 ms. The corresponding procedure in LRE-TL was invoked 351941 times, with the total time spent being 1307 ms.

Table 3. CPU Profiler Results for USG and LRE-TL on 8 Processors

Procedure	Initialisation		Scheduling events		Removal		Insertion	
	No. of invocations	Time (ms)	No. of invocations	Time (ms)	No. of invocations	Time (ms)	No. of invocations	Time (ms)
USG	1000	143	27292	131	320 + 58553 = 61759	54.5 + 32 = 86.5	61759	1.5
LRE-TL	22060	9772	351941	1175	633365	469	385374	162

Table 4 summarises the results of the CPU profiler for both USG and LRE-TL on 16 processors. In this case, all the invocations of the initialisation procedure in USG required only 193 ms. Conversely, the LRE-TL initialisation procedure (*TL_Plane_Initialise*) was invoked 22060 times for the 1000 tasksets, implying about 23 invocations per taskset, requiring a total time of 9772 ms. The procedure for handling the scheduling events in USG was invoked 44233 times for the 1000 tasksets, requiring a total time of 134 ms. The corresponding procedure in LRE-TL was invoked 351941 times, with a total time of 1175 ms required.

Table 4. CPU Profiler Results for USG and LRE-TL on 16 Processors

Procedure	Initialisation		Scheduling events		Removal		Insertion	
	No. of invocations	Time (ms)	No. of invocations	Time (ms)	No. of invocations	Time (ms)	No. of invocations	Time (ms)
USG	1000	193	44233	134	3279+112580= 115859	97.2	115859	10.7
LRE-TL	22060	9772	351941	1175	633365	469	385374	162

Table 5 summarises the CPU results for both USG and LRE-TL on 32 processors. Here, the time spent on all the invocations of the initialisation procedure in USG is only 330 ms. Conversely, the LRE-TL initialisation procedure (*TL_Plane_Initialise*) was invoked 55518 times for the 1000 tasksets, implying about 26 invocations per taskset, with a total time of 48475 ms required. Further, the procedure for handling the scheduling events in USG was invoked 66277 times for the 1000 tasksets, requiring a total time of 340 ms. The corresponding procedure in LRE-TL was invoked 3456556 times, with a total time of 7306 ms required.

Table 5. CPU Profiler Results for USG and LRE-TL on 32 Processors

Procedure	Initialisation		scheduling events		remove		insert	
	Invoked	Time (ms)	Invoked	Time	Invoked	Time	Invoked	Time
USG	1000	330	66277	340	2875+221903= 224778	251	224778	31.9
LRE-TL	55518	48475	3456556	7306	6480930	2793	4105416	644

6. CONCLUSION

This paper presented a practical method for analysing the performance of optimal real-time and semi-optimal multiprocessor scheduling algorithms. Using LRE-TL and USG as representative examples of the two types of algorithms, respectively, the proposed method was implemented on machines equipped with 2 and 4 processors, respectively. Both algorithms were executed on the same tasksets and monitored by the CPU profiler of Oracle JavaTM VisualVM. The results of the CPU profiler revealed that USG outperformed LRE-TL with respect to the number of invocations of the procedures for handling the scheduling events and the total time spent on the invocations. This indicates the practical applicability of USG, unlike optimal algorithms, which fall short of their theoretical optimality in practical applications.

7. REFERENCES

[1] H. Kopetz, Real-Time Systems, 2 ed. New York, NY, USA: Springer, 2013.

[2] A. Burns and A. J. Wellings, Real-Time Systems and Programming Languages, 4 ed. Canada: Pearson Education, 2009.

[3] G. C. Buttazzo, Hard Real-Time Computing Systems, 3 ed. vol. 24. New York, NY, USA: Springer, 2013.

[4] C. M. Krishna, Real-Time Systems: Wiley Online Library, 1999.

[5] P. A. Laplante and S. J. Ovaska, Real-Time Systems Design and Analysis, 4 ed. New York, NY, USA: Wiley, 2011.

[6] I. Lee, J. Y. Leung, and S. H. Son, Handbook of Real-time and Embedded Systems: CRC Press, 2007.

[7] J. W. S. Liu, Real-Time Systems, 1 ed. New Jersey, NJ, USA: Prentice Hall, 2000.

[8] R. Mall, Real-Time Systems: Theory and Practice: Pearson Education, 2009.

[9] G. Nelissen, V. Berten, V. Nélis, J. Goossens, and D. Milojevic, ‘U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks,’ in 24th Euromicro Conference on Real-Time Systems (ECRTS), Pisa, Italy, 2012, pp. 13-23.

[10] S. Funk, G. Levin, C. Sadowski, I. Pye, and S. Brandt, ‘DP-Fair: a unifying theory for optimal hard real-time multiprocessor scheduling,’ Real-Time Syst, vol. 47, pp. 389-429, 2011.

[11] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings, ‘Real-time system scheduling,’ in Predictably Dependable Computing Systems, B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, Eds., ed Berlin, Heidelberg, Germany: Springer, 1995, pp. 41-52.

[12] S. Funk, ‘LRE-TL: An optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines,’ Real-Time Syst, vol. 46, pp. 332-359, 2010.

[13] H. Alhussian, N. Zakaria, and A. Patel, ‘An unfair semi-greedy real-time multiprocessor scheduling algorithm,’ Computers & Electrical Engineering, vol. 50, pp. 143-165, 2016.

[14] M. L. Dertouzos and A. K. Mok, ‘Multiprocessor online scheduling of hard-real-time tasks,’ Software Engineering, IEEE Transactions on, vol. 15, pp. 1497-1506, 1989.

[15] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, ‘Proportionate progress: A notion of fairness in resource allocation,’ ALGORITHMICA, vol. 15, pp. 600-625, 1996.

[16] D. Zhu, X. Qi, D. Mossé, and R. Melhem, ‘An optimal boundary fair scheduling algorithm for multiprocessor real-time systems,’ Journal of Parallel and Distributed Computing, vol. 71, pp. 1411-1425, 2011.

[17] D. Zhu, D. Mossé, and R. Melhem, ‘Multiple-Resource Periodic Scheduling Problem: how much fairness is necessary?,’ in Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE, 2003, pp. 142-151.

[18] H. Cho, B. Ravindran, and E. D. Jensen, ‘An optimal real-time scheduling algorithm for multiprocessors,’ in 27th IEEE Real-Time Systems Symposium, Rio de Janeiro, Brazil, 2006, pp. 101-110.

[19] S. Funk and V. Nanadur, ‘LRE-TL: An Optimal Multiprocessor Scheduling Algorithm for Sporadic Task Sets,’ in 17th International Conference on Real-Time and Network Systems, 2009, pp. 159-168.

[20] H. S. A. Alhussian, M. N. B. ZAKARIA, and F. A. B. HUSSIN, ‘Minimizing scheduling overhead in LRE-TL real-time multiprocessor scheduling algorithm,’ Turkish Journal of Electrical Engineering & Computer Sciences, vol. 25, pp. 263-277, 2017.

[21] H. Alhussian, N. Zakaria, and F. A. Hussin, ‘Minimizing scheduling overhead in LRE-TL real-time multiprocessor scheduling algorithm,’ Turk J Elec Eng & Comp Sci, 2015.

[22] Oracle. (2013). VisualVM - All-in-One Java Troubleshooting Tool. Available: https://visualvm.java.net

How to cite this article:
Alhussian H, Abdulkadir S J, Zakaria N, Patel A, Alzahrani A. Practical performance analysis of real-time multiprocessor scheduling algorithms. J. Fundam. Appl. Sci., 2018, *10(2S), 60-73.*