Journal of Computer Sciences

Original Research Paper

Enhanced Preemptive Global Utility Accrual Real Time Scheduling Algorithms in Multicore Environment

Idawaty Ahmad, Mohamed Othman and Zuriati Ahmad Zulkarnain

Department of Communication Technology and Network, Faculty of Computer Science and Information Technology, University Putra Malaysia, 43400 UPM, Serdang, Selangor DE, Malaysia

Article history
Received: 05-11-2015
Revised: 09-02-2016
Accepted: 10-02-2016

Corresponding Author:
Idawaty Ahmad
Department of Communication Technology and Network,
Faculty of Computer Science and Information Technology,
University Putra Malaysia,
43400 UPM, Serdang, Selangor DE, Malaysia
Email: idawaty@upm.edu.my

Abstract: This paper proposed an efficient real time scheduling algorithm using global scheduling paradigm running in multicore environment known as Global Preemptive Utility Accrual Scheduling (GPUAS) algorithm. The existing TUF/UA multiprocessor scheduling algorithms known as Greedy-Global Utility Accrual (G-GUA) and Non Greedy-Global Utility Accrual (NG-GUA) algorithms is seen to overlook the efficiency on its task scheduling algorithm. These algorithms have adapted the task migration attribute considering the load balancing problem in multi core platform. The existing PUAS uniprocessor scheduling algorithm is mapped into the multicore scheduling environment that consists of the global scheduling schemes considering the migration attribute of the executed tasks. The main principal of global scheduling is that it allows the executed tasks to migrate from one processor to the other processors whenever a scheduling event occurs in the system. The proposed GPUAS algorithm inherits the characteristics of PUAS in uniprocessor where it can preempt the highest PUD task at any event that occurs in the system. In this research, the proposed GPUAS algorithm enhanced the existing NG-GUA and G-GUA algorithms. The developed simulator has derived the set of parameter, events and performance metrics according to a detailed analysis of the base model. The proposed GPUAS algorithm achieved the highest accrued utility for the entire load range. The proposed GPUAS algorithm is more efficient than the existing algorithms, producing the highest accrued utility ratio and less abortion ratio making it more suitable and efficient for real time application domain.

Keywords: Real Time System, Utility Accrual Scheduling, Multicore, Discrete Event Simulation

Introduction

In the presence of extremely overloaded tasks traffic, the RTS requires multicore environment with an efficient load sharing capability to accommodate the surplus load. The load sharing mechanism is required in order to migrate the executed tasks across multiple processors. This ensures that no processor is idle while some tasks are waiting to be scheduled on other processors. The load sharing problem in multiprocessor environment can be solved by deploying the task migration attribute in the executed tasks.

Problem Statement

The existing TUF/UA multiprocessor scheduling algorithms known as Greedy-Global Utility Accrual (G-GUA) and Non Greedy-Global Utility Accrual (NG-GUA) algorithms (Garyali *et al*., 2010) is seen to overlook the efficiency on its task scheduling in multicore platform. G-GUA uses a greedy strategy where task whose execution yields the maximum PUD is selected to be scheduled at a particular instance. For load sharing purpose, the requesting task is placed at the queue that has the least remaining execution cost. This requesting task assignment behavior in G-GUA may affect the total utility accrued because it does not consider the value of utility when determining the suitable queue at that particular instance. On the other hand, NG-GUA uses dual metric to overcome overloaded tasks in RTS. During under load it uses deadline and during overloaded condition, task whose execution yields the maximum PUD is selected to be

Science Publications

© 2015 Idawaty Ahmad, Mohamed Othman and Zuriati Ahmad Zulkarnain. This open access article is distributed under a Creative Commons Attribution (CC-BY) 3.0 license.

scheduled at a particular instance. However, NG-GUA immediately aborts the requesting task that has potential to produce zero utility to the system due to its infeasibility. The abortion of a task leading to a zero utility acquired by the requesting task to the system. This will have an effect on the total utility accrued to the system. If the underlying scheduling scheme can reduce the abortion of the requesting task efficiently, then the system will possibly attain a higher utility thus enhancing the overall system's performance.

Objective

The objective of this research is to enhance the efficiency of the existing TUF/UA scheduling algorithms in multiprocessor environment i.e., NG-GUA and G-GUA to accommodate the overloaded tasks traffic so that the maximum total utility accrued to the system. This paper proposed the GPUAS algorithm that considered an efficient task request location with migration task attribute for solving the overloaded situation. If the underlying scheduling scheme places the task's request among the processor's queue efficiently and reduce the abortion problems, then the system will gain higher utility thus enhancing the overall system's performance.

Materials and Methods

Approach

A discrete event simulation is used as a methodology to verify the performances of GPUAS and the existing algorithms. In order to precisely remodel and further enhance the algorithms, DES written in C language in Visual C++ environment is the best method to achieve this objective. Figure 1 shows the deployed simulation framework.

Simulation Framework

The simulation of multicore infrastructure consists of a source, task entities and an array of *utlist* queues and resources to represent the various numbers of processors in the system.

Task Model

Tasks have one of two types of migration characteristics. The migration type of a task is denoted by *Migration* $\in$ {NON_MIGRATE, MIGRATE}. Tasks that are not allowed to migrate among processors possess the NON_MIGRATE attribute while those with a MIGRATE attribute can be migrated among the processors and are considered only in the global scheduling paradigm.

TUF Model

The general dominant task attribute is associated to using timing constraint which is denoted as deadline. The timing constraint of a task is designed using the step and arbitrary TUF model in this study (Li *et al*., 2006). A TUF describes a task utility contribution to the system as a function of its completion time. The TUF shape of a task is denoted by *Shape* $\in$ {STEP, ARBITRARY}. The step TUF model used in the simulator is shown in Fig. 2. The maximum utility that could possibly be gained by a task is denoted as *MaxAU*. The random value of *MaxAU* abides normal distribution (10, 10) i.e., the mean value and variance is set 10 to conform to the benchmark. The *InitialTime* is the starting time for which the function is defined. The *TerminationTime* is the last time for which the function is defined. That is, *MaxAU* is defined in within the time interval of [*InitialTime, TerminationTime*]. The completion of a task at an instance i.e., *sclock* within this interval will yield a random positive utility denoted as *Utility* which is equal to the *MaxAU* for step TUF model as shown in Fig. 2a. The completion of a task breaching the stipulated deadline causes the value of *Utility* and *MaxAU* to become zero. If the *TerminationTime* is reached and the task has not finished its execution, it accrues zero utility to the system.

The arbitrary shape TUF is represented as a continuous and derivable polynomial equation as shown in Fig. 2b. The maximum utility that could possibly be gained by a task is denoted as *MaxAU*. The random value of *MaxAU* abides normal distribution (10, 10) i.e., the mean value and variance is set 10 to conform to the benchmark. For arbitrary TUF, the completion of a task within the *InitialTime* and *TerminationTime* interval will yield a random positive utility denoted as *Utility* as shown in Fig. 2b.

Task Assignment Algorithm

All tasks are assigned to processors by task assignment algorithm as shown in Fig. 3. The number of processors is checked before the task is assigned to a specific processor. The *Tgen.cpuid* parameter is used to identify the assigned processor ID. In uniprocessor environment the value of *Tgen.cpuid* is equal to 0. This indicates that the processor ID zero is assigned to task *Tgen*. In multiprocessor environment, the generated task *Tgen* is assigned according to the value of *ShortestCPU* parameter. This parameter captures the processor ID that has the smallest value of *TotalExec[cpuid]]* value in their respective *cpuid* queue. The *TotalExec[cpuid]* measures the execution time of all requests that are currently pending in the *cpuid* queue. The *TotalExec* parameter is increased every time a request of a task is inserted into the *cpuid* queue. The *TotalExec* parameter is reduced every time a request of a task is deleted from the *cpuid* queue.

DOI: 10.3844/jcssp.2015.1099.1107

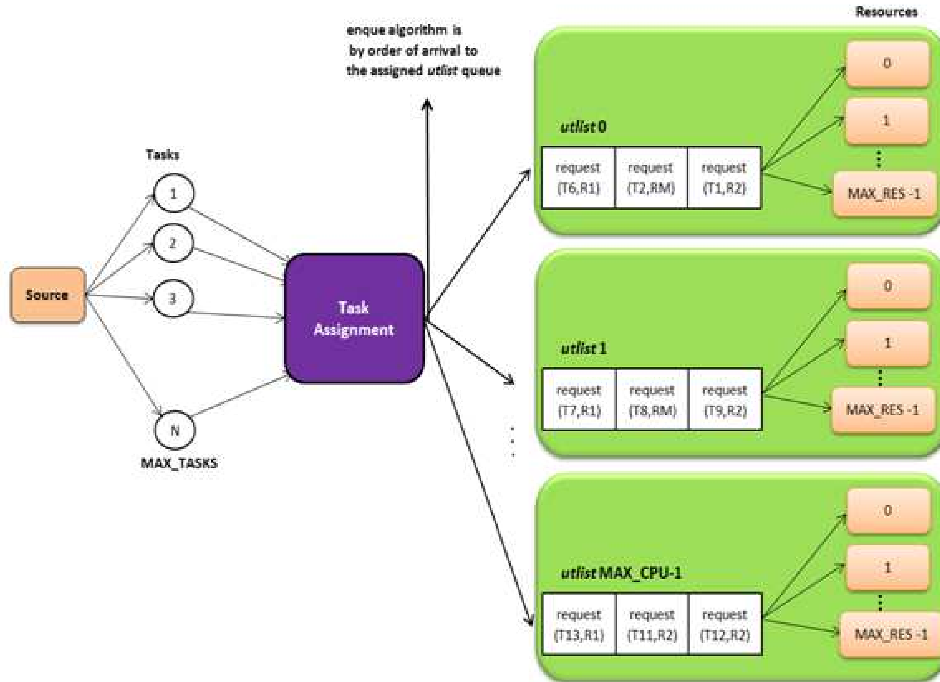

Fig. 1. Simulation framework (Idawaty *et al*., 2012)

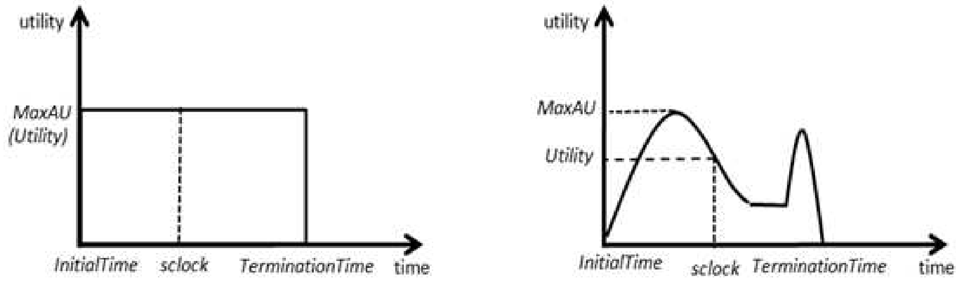

Fig. 2. TUF model

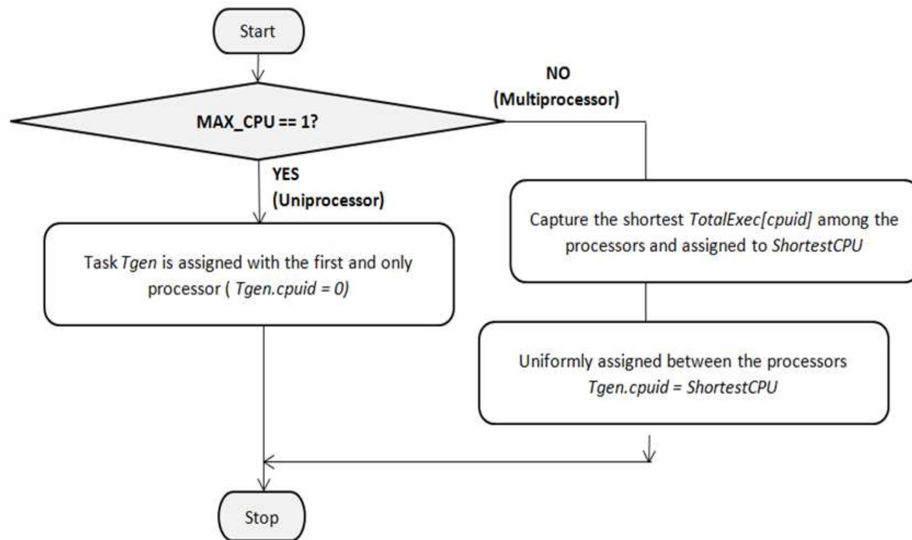

Fig. 3. Task assignment algorithm

DOI: 10.3844/jcssp.2015.1099.1107

Existing Algorithms

The features of the existing G-GUA algorithm are simplified as follows:

1. G-GUA uses the PUD metric i.e., task whose execution yields the maximum PUD over others is scheduled in the system. Once a scheduling event is triggered, a new schedule is created that may result in the task executing on the other processors to be preempted.
2. G-GUA allows the request of a task to migrate to the idle resource in another processor in the system. It also allows the request of a task to be inserted into a processor's queue with the least of total remaining execution cost for the respective resource.

The existing NG-GUA algorithm is a TUF/UA multiprocessor real time scheduling algorithm that allows tasks to be subject to run-time uncertainties, overloads and global migration (Garyali *et al*., 2010). The none greediness in the name of this algorithm describes the tendency of the algorithm to accrue as much total utility during overload situation while in under load situation it uses deadline as its metric and schedule task with the earliest deadline first. The features of the existing NG-GUA are simplified as follows:

1. NG-GUA uses two metrics for task scheduling depending on the overload condition in the system. During overloaded, the PUD metric i.e., task whose execution yields the maximum PUD over others is scheduled in the system. In under load situation, the deadline metric i.e., task with the earliest deadline is scheduled in the system. Once a scheduling event is triggered, a new schedule is created that may result in the task executing on the other processors to be preempted.
2. NG-GUA allows the request of a task to migrate to the idle resource in another processor in the system. It also allows the request of a task to be inserted into a processor's queue with the least of total remaining execution cost for the respective resource.

Proposed Algorithm

The different approach of GPUAS is in the decision for queuing the requesting task with the lowest PUD in the system. GPUAS uses the lowest PUD metric as opposed to G-GUA that used the least sum of total remaining execution cost for the respective resource. GPUAS uses only the PUD metric to schedule the incoming task as opposed to the existing algorithms that uses dual metrics i.e., PUD and deadline to schedule tasks in the system according to the current load in the system. Figure 4 shows the different between these algorithms to identify the most suitable queue to insert the requesting task i.e., *Treq* when it cannot be instantaneously scheduled to use a resource in the system. There are two parameters being used by the scheduling algorithms on the decision to locate task *Treq* as follows:

i. NG-GUA, G-GUA and GPUAS measure the least sum of remaining execution cost for resource rid among the *utlist* queues. The *LeastTime* parameter is used to represent the least sum of remaining execution cost. Additionally, the *LeastUtlist* parameter is used to represent which *utlist* queue possess the least sum of remaining execution cost. The remaining execution cost for a resource i.e., *rid* in a queue is captured from the remaining *HoldTime* parameter of each task that requesting for resource *rid* in the respective *utlist* queue. The sum of the remaining *HoldTime* of each identified request is accumulated in the *TotalCost* parameter. Therefore, before the searching procedure for a request in any *utlist* queue is executed, the value of *TotalCost* is initialized to 0.0000 as depicted in Fig. 4.

ii. GPUAS additionally measures the highest PUD of task request for the resource *rid* in the respective *utlist* queue. The *HighestPUD* parameter is used to represent the highest PUD among the task request in the respective queue. Initially, this parameter is set to 0.0000 as shown in Fig. 4.

Referring to Fig. 4, the *TotalCost* and *HighestPUD* parameter is initialized to 0.0000. The identification of which *utlist* queue for the respective *rid* is depicted in the in *res[rid].cpuid* parameter. The scheduler search for a request for resource *rid* in a queue. Two additional pointers are needed for the searching procedure i.e., the *work* and *prev_utlist* pointers. The *work* pointer moves from one element to the next element starting from the *head_utlist* to the *tail_utlist* searching for the corresponding request. The *prev_utlist* points to the previous element before the currently processing element that is shown by the *work* pointer. Initially both of these pointers point to the first element in the utlist queue as depicted in Fig. 4. The work pointer then checks the resource *rid* of the first element in the *utlist* queue. If it does not discover the request for resource *rid*, it will search for the next element. This is repeated to the subsequent elements until the end of the *utlist* queue.

As shown in Fig. 4, in the case the *work* pointer discovers a request for *rid*, the information of the task request is obtained from the *tid* element of the *work* pointer i.e., *work-tid.* For the purpose of clarification, the respective task is known as *Twait* in the figure. The execution mode of task *Twait* in *Twait.Mode* is checked. Subsequently, the execution mode for task *Twait* is classified into the NORMAL or ABORT mode as stated below:

DOI: 10.3844/jcssp.2015.1099.1107

i. Task *Twait* is currently executing in the NORMAL mode. In this case, the scheduler measures the least sum of remaining execution cost for resource *rid* in the respective *utlist* queue. The remaining execution cost for resource *rid* in a *utlist* queue is captured from the remaining *HoldTime* parameter obtained from the *work->HoldTime* parameter. The sum of remaining *HoldTime* of each identified request is accumulated in the *TotalCost* parameter. Referring to Fig. 4, the PUD of this task request is calculated as *Twait.PUD.* The calculation formula of PUD is elaborated in (Jensen *et al*., 1985; Li *et al*., 2006). Subsequently, the PUD is then compared with the *HighestPUD* parameter that contains the value that is currently producing highest PUD among the tasks in the respective *utlist* queue. Initially, the value of largest PUD is set to 0.0000. If task *Twait* produces a larger PUD than the value currently in *HighestPUD*, the *Twait.PUD* is considered as the highest PUD so far. Thus, the value of *HighestPUD* is updated to be equal to the *Twait.PUD*.

ii. Task *Twait* is currently executing in the ABORT mode. In this case, the request of task *Twait* for resource *rid* in the respective *utlist* queue can be delayed. Thus, the *AbortTime* is not accumulated in the *TotalCost* parameter. The PUD of *Twait* is equal to 0.0000.The scheduler then proceed searching for the next element in the respective *utlist*. The above mentioned procedure is repeated to the subsequent elements until to the end of the *utlist.* The outcome of this procedure is to obtain the value of *TotalCost* and *HighestPUD* parameters.

The measurement in GPUAS and the existing NG-GUA and G-GUA algorithms differ as follows:

i. NG-GUA and G-GUA algorithms use the measured *TotalCost* parameter and compares to the *LeastTime* parameter. Note that the *LeastTime* parameter is used to represent the least sum of remaining execution cost among the *utlist* queues for resource *R.* If the value of *LeastTime* exceeds the *TotalCost* parameter, the *LeastTime* is updated to reflect the least sum of total remaining execution cost among the *utlist* queues in the system. The *LeastUtlist* parameter specifies which *utlist* that contained the least sum of total remaining execution cost i.e., *res[rid].cpuid.*

ii. GPUAS uses two parameters i.e., the *TotalCost* and *HighestPUD* to decide which *utlist* queue to locate task *Treq.*

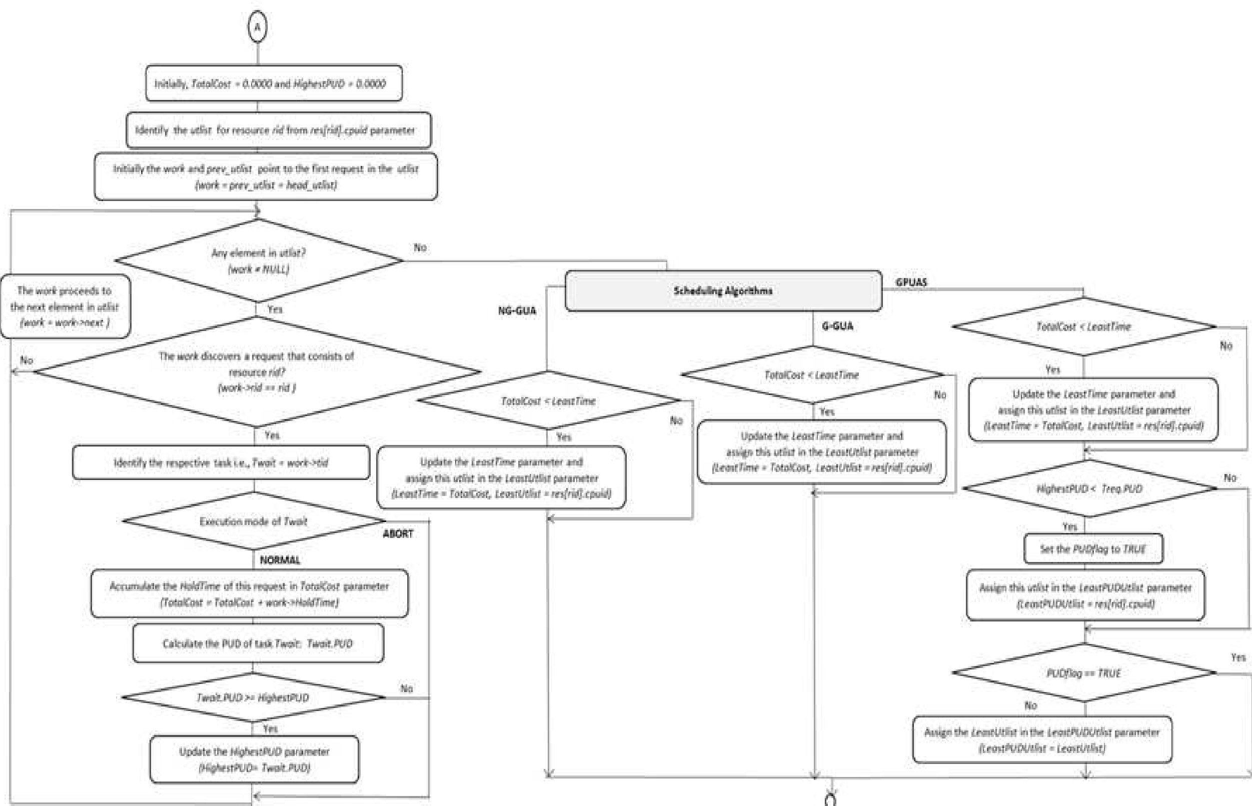

Fig. 4. Scheduling algorithms

There are three (3) conditions to validate each *utlist* as stated below:

a. GPUAS firstly identifies the *LeastUtlist* queue that is identical to the above mentioned G-GUA. The calculated *TotalCost* parameter is compared to the *LeastTime* parameter. If the value of *LeastTime* exceeds the *TotalCost* parameter, the *LeastTime* is updated to reflect the least sum of total remaining execution cost among the *utlist* queues in the system. The *LeastUtlist* parameter specifies which *utlist* that contained the least sum of total remaining execution cost i.e., *res[rid].cpuid.*
b. GPUAS compares the highest PUD in this *utlist* (i.e., *HighestPUD*) with the PUD of the requesting task *Treq* (i.e., *Treq.PUD).* If the value in *Treq.PUD* exceeds the *HighestPUD* parameter, the *PUDflag* parameter is tagged as *TRUE* to reflect the existence of a *utlist* with a lower PUD compared to task *Treq*. The *LeastPUDUtlist* parameter specifies which *utlist* that contained a lower PUD compared to *Treq* i.e., *res[rid].cpuid.* Referring to Fig. 4, in the case the value in *Treq.PUD* is less than or equal to the *HighestPUD* parameter, the *PUDflag* value remained unchanged.
c. GPUAS considers the *LeastUtlist* parameter in the case none of the task in *utlist* queue produced a lower PUD as compared to the PUD of task *Treq*. In this case, GPUAS considers the *LeastUtlist* as its *LeastPUDUtlist.*

The rationale to select the *LeastPUDUtlist* queue that has a lower PUD as compared to the requesting task i.e., *Treq* in GPUAS is to ensure that if task *Treq* is to be inserted into the *utlist* queue, the condition in the *LeastPUDUtlist* will ensure that task *Treq* can use the resource *rid* soon after the owner task has releases it.

Experimental Setting

The performances of real time scheduling algorithms are measured by the metrics which rely on the respective application specifications. The Accrued Utility Ratio (AUR) metric defined in (Jensen *et al*., 1985) has been extensively utilized in the existing TUF/UA scheduling algorithms and is considered as the standard metric in this domain (Wu *et al*., 2004; Li *et al*., 2006).

AUR is defined as the ratio of accrued aggregate utility to the maximum possibly attained utility. Equation 1 shows that each task i has its maximum value of utility which is denoted as MaxAU(i). After a task i has completed its execution, it will yield a value denoted as Util(i). These values are then accumulated for all tasks i.e., MAX_TASKS. The AUR is calculated as:

$$AUR = \frac{\sum_{i=1}^{MAX_TASKS} Utill(i)}{\sum_{i=1}^{MAX_TASKS} MaxAU(i)} \quad (1)$$

Results

Based upon the results acquired from the simulation, the interpretations of the results are performed. The numbers of processors considered in the system are two, four and eight (Garyali *et al*., 2010). The scheduling algorithm that is proposed in the multiprocessor scheduling environment is known GPUAS. The NG-GUA and G-GUA algorithms are used to compare the performance of GPUAS algorithm. The plots from all the results cover an average load in the range of [1-10] in the multiprocessor environment with two, four and eight-core platform (Garyali *et al*., 2010) for step and arbitrary TUF task model.

Figure 5 depicts the AUR result under an increasing load for step TUF. From the overall results, as the number of processors increase, a higher utility is recorded for all scheduling algorithms. Overall, the nature of the curves indicates that the proposed GPUAS algorithm has achieved better performance by producing a higher accrued utility as compared to the existing NG-GUA and G-GUA algorithms.

Figure 6 depicts the AUR results for execution of the arbitrary TUF tasks in the system. Overall, the patterns of the curves from the results in the arbitrary TUF tasks set are similar to the step TUF tasks set. In the case of arbitrary TUF, a task may not be able to accrue its maximal possible utility even though the execution is completed before its termination time. Although these algorithms guarantee that the highest PUD task to be selected, it does not necessarily represent that the maximum possible utility gained by the executed tasks.

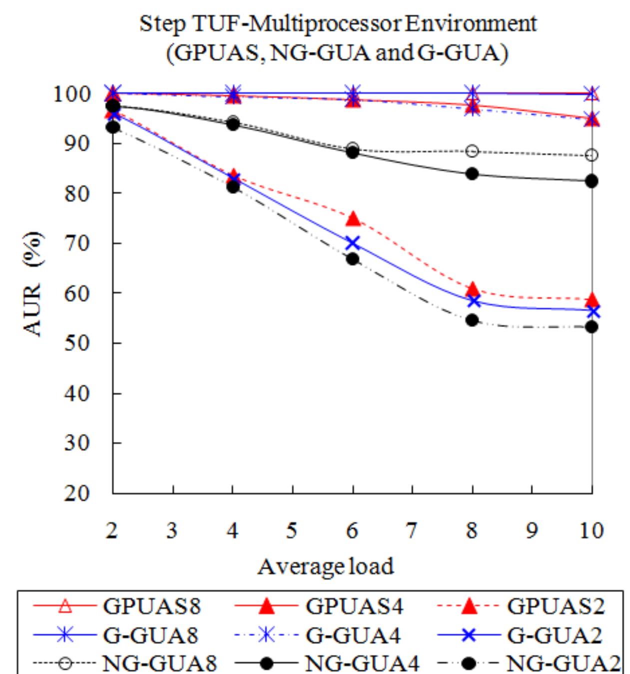

Fig. 5. Results for step TUF

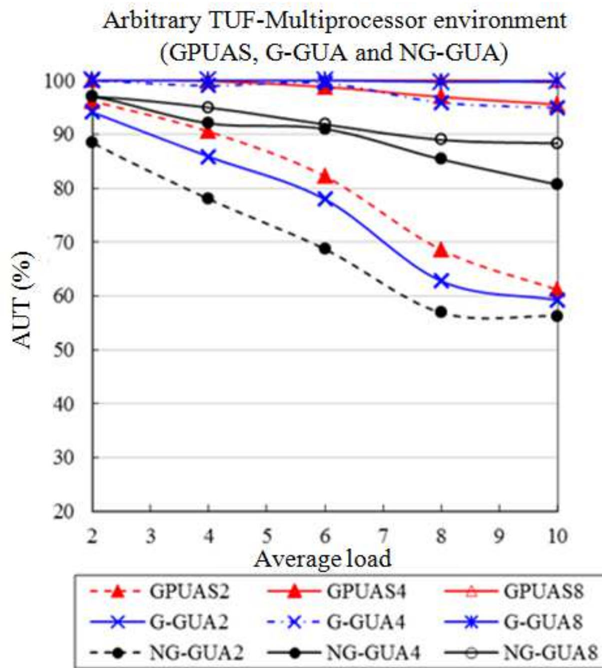

Fig. 6. Results for arbitrary TUF

From the overall results, as the number of processors increase, a higher utility is recorded for all scheduling algorithms. From the overall results, GPUAS shows significant improvement compared to NG-GUA algorithm in all multicore platforms. GPUAS also shows a significant improvement compared to the G-GUA algorithm in the dual core platform. From the results, it is observed that the GPUAS8 and G-GUA8 algorithms that are executed in the eight core platform have produced the highest utility to the system as compared to the dual and quad core platforms.

Discussion

From the overall results shown in Fig. 5, GPUAS shows significant improvement compared to NG-GUA algorithm in all multicore platforms for the entire load range. However, GPUAS shows a significant improvement compared to the G-GUA algorithm only in the dual core platform. In dual core platform, the average load of 2 is estimated as the starting point of an overloaded situation in the system. At this load, GPUAS2 achieved 96.58% and G-GUA2 algorithm achieved 96.05% and NG-GUA2 with 93.14% of the accumulated utilities. At this load, GPUAS2 has improved the NG-GUA2 algorithm for 3.44% and improved G-GUA2 for 0.53% of the accumulated utilities.

However, as the load increases, more significance of GPUAS2 is observed in the system as compared to the G-GUA and NG-GUA algorithms. When load is equal to 6, in the dual core platform, GPUAS2 accrued 75.03%, G-GUA2 accrued 70.05% and NG-GUA2 accrued 66.76% of utilities. GPUAS2 algorithm has improved G-GUA2 for 4.98% and outperformed NG-GUA2 algorithm for 8.27% of the accumulated utilities. At the highest load (i.e., *load* = 10), GPUAS2 has achieved 58.89%, G-GUA2 with 56.61% and NG-GUA2 gained 53.24% of the accumulated utilities. Thus, GPUAS2 has improved G-GUA2 for 2.28% and outperformed NG-GUA2 algorithm for 5.65%. GPUAS2 outperformed the G-GUA2 algorithm because GPUAS2 ensure that the newly request is inserted into the *utlist* queue that has the highest possibility to be executed as soon as the owner task releases the respective resource. Therefore, the requesting tasks may produce positive utility to the system. On the other hand, G-GUA2 locates the requesting task into the *utlist* that has the least remaining execution cost. Therefore, G-GUA2 does not guarantee that the requesting task will be scheduled to use the respective resource as soon as the task being scheduled in GPUAS. Therefore, more requesting tasks are ending up waiting in the *utlist* without being scheduled in G-GUA2. This reflects the lower utility accrued in G-GUA2 as compared to GPUAS2 in dual core platform. The improvement of GPUAS2 is small i.e., at most only 4.98% because in the dual core platform, although more number of tasks inserted into the selected queues in GPUAS2 but more tasks are overdue and therefore ending up being aborted. This is why the task placement in GPUAS2 is less over the G-GUA2.

It is observed that GPUAS2 outperformed NG-GUA2 for the entire load range. The excellent performance of GPUAS2 is also observed over the NG-GUA2 for the entire load range. This is because GPUAS2 is a greedy scheduling algorithm and uses PUD as a metric to achieve the highest accrued utility at any instance while NG-GUA2 uses the deadline metric during under load and PUD during overloaded conditions. On the other hand, during overloads NG-GUA aborts any requesting task that produced lowest PUD to overcome the overloaded situation. The abortions reduced the value of utility accrued to the system in NG-GUA2. GPUAS omits the abortion and inserts the lowest PUD task into a queue.

From Fig. 5, it is observed that a sharper degradation as the load increases in dual core platform. Although algorithms allows tasks to migrate to the available resources or move to the resources with the least PUD in the *utlist* queue, in dual core platform the number of available resources is limited. Due to the limited resources, many backlogged tasks are ending up not been migrated anywhere although these algorithms allows them to do so. Due to the limited resources, more tasks are overdue and therefore ending up being aborted. More aborted tasks are produced as the load increases and consequently produced more zero utility tasks to the

system. This is why a sharper degradation is observed as the load increases in the dual core platform.

Referring to Fig. 5, in four core platform, approximately the system is considered to be overloaded when the average load is equal to 4. At this load, GPUAS4 has successfully gained 99.63% of utility and G-GUA4 moderately accrued 99.33% while NG-GUA4 accrued 93.84% of the utilities. Thus, the GPUAS4 algorithm outperforms G-GUA4 for 0.30% and NG-GUA for 5.79% at this load. It is observed that G-GUA and GPUAS accrued the same utility in the quad and eight core platforms. At the highest load (i.e., *load* = 10), GPUAS4 has achieved 94.96%, G-GUA4 with 94.71% and NG-GUA4 gained 82.42% of the accumulated utilities. Thus, GPUAS4 has improved G-GUA4 for 0.25% and outperformed NG-GUA4 algorithm for 12.54%.

From the results in Fig. 5, it is observed that the GPUAS8 and G-GUA8 algorithms that are executed in the eight core platform have produced the highest utility to the system as compared to the dual and quad core platforms. In eight core platform, approximately the system is considered to be overloaded when the average load is equal to 8. At this load, GPUAS8 and G-GUA8 have successfully gained 100% of utility and NG-GUA moderately accrued 88.45% of the utilities. Thus, the GPUAS8 and G-GUA8 algorithms outperformed NG-GUA8 for 11.55% at this load. From the figure, at the highest load (i.e., *load* = 10), GPUAS8 and NG-GUA8 have achieved 100% and NG-GUA8 with 87.56% of the accumulated utilities. Thus, GPUAS8 and G-GUA8 have improved NG-GUA8 for 12.44%. As the number of processors increase, the higher utility accrued to the system by the GPUAS algorithm as compared to the existing G-GUA and NG-GUA algorithms. The enhancement of GPUAS has tremendously improved the utility accrued to the system in multiprocessor environment.

Overall, the improvement of GPUAS over the G-GUA algorithm is observed only in the dual core platform. In the quad and eight core platforms, the performances of these algorithms are similar. Note that, GPUAS ensures that a task is inserted into a selected queue that has the highest possibility to be executed. In dual core platform, due to limited resources more number of tasks inserted into the selected queues. In dual core platform, only two queues are available for the insertion of tasks in the queues. Therefore, the insertion always occurs between these queues and more number of tasks inserted into the queues. Hence, the PUD metric used in GPUAS for selection of queues for task insertion has an impact to the system. On the other hand, in the quad and eight core platforms, more resources are available. Therefore, less number of tasks inserted into the queues. Therefore, the metric used for task insertion in GPUAS has a minor impact to the system.

Conclusion

This paper has discussed the design and evaluation of GPUAS in the multiprocessor environment. The proposed GPUAS algorithm is compared with the existing NG-GUA and G-GUA algorithms. Simulation results revealed that GPUAS2 has improved for less than 4.98% on G-GUA in dual core platform and remain the same performances in quad and eight core platforms. Since GPUAS has improved G-GUA in the metric used for insertion of task in a queue, GPUAS only has impact on the system with less number of queue i.e., in the dual core platform with only two queues available. In the quad and eight core platforms, more resources are available and therefore less number of tasks is inserted into a queue. Therefore insertion procedure is less significant in this environment. Simulation results also revealed that GPUAS has improved the NG-GUA at most 12.44% for the entire load range in all platforms. This is because GPUAS omits the unnecessary abortions that occur in NG-GUA. Overall, the GPUAS algorithm outperforms the existing algorithms by accruing the highest utility to the system due to the highest resource consumption by exploiting the migration and task insertion attributes of the executed tasks. This chapter also has confirmed the advantage of GPUAS as compared to the existing NG-GUA algorithm in all platforms for at most 12.44% and has improved the G-GUA algorithm at most 4.98% in the dual core platform. The contribution of GPUAS algorithm in the dual, quad and eight core platforms that achieved the highest accrued utility and success ratio making it suitable and efficient scheduling algorithm for real time application.

Acknowledgement

The authors wish to thank anonymous reviewers for their valuable, detailed comments that improve both the content and representation of this study.

Funding Information

This research was funded by the Ministry of Higher Education Malaysia and Universiti Putra Malaysia under Fundamental Research Grant FRGS 08-01-15-1722FR

Author's Contributions

All authors equally contributed in this work.

Ethics

The corresponding author confirms that the other authors have read and approved the manuscript and there is no ethical issue involved. This paper is original and contains unpublished material.

References

Garyali, P., M. Dellinger and B. Ravindran, 2010. On best-effort utility accrual real-time scheduling on multiprocessors. Proceedings of the 14th International Conference on Principles of Distributed Systems, Dec. 14-17, Springer-Verlag Berlin, Heidelberg, pp: 270-285. DOI: 10.1007/978-3-642-17653-1_21

Idawaty, A., S. Shamala, M. Othman and Z. Zuriati, 2012. Performance of partition utility accrual real time scheduling algorithm. J. Comput. Sci., 8: 1225-1234.

Jensen, E.D., C.D. Locke and H. Tokuda, 1985. A time-driven scheduling model for real-time operating systems. Proceeding of the IEEE Symposium on Real-Time System, (SRS' 85), IEEE Computer Society, pp: 112-122.

Li, P., H. Wu, B. Ravindran and E.D. Jensen, 2006. A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints. IEEE Trans. Computer., 55: 454-469. DOI: 10.1109/TC.2006.47

Wu, H., B. Ravindran, E.D. Jensen and P. Li, 2004. CPU scheduling for statistically-assured real-time performance and improved energy efficiency. Proceeding of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, Sept. 8-10, IEEE Xplore Press, USA, pp: 110-115. DOI: 10.1109/CODESS.2004.240827