Overhead control in real-time global scheduling

M.Naeem Shehzad, A.M Déplanche, Yvon Trinquet, Richard Urunuela
IRCCyN, site de l'Ecole Centrale de Nantes
Nantes, France.
{Naeem.Shehzad,Anne-Marie.Deplanche,Yvon.Trinquet,Richard.Urunuela}@irccyn.ec-nantes.fr

Abstract

A number of optimal algorithms are known for scheduling of periodic tasksets with implicit deadlines on real-time multiprocessor systems. These algorithms belong to the branch of global scheduling which allows the migration of tasks between the processors. Though these are theoretically optimal, questions are raised about their practical implementation because optimality is achieved at cost of excessive scheduling points, migrations and preemptions. Controlling the overhead to a possible minimum level is one of the critical areas of research. This paper is specifically concerned with particular global scheduling algorithms that combine fluid scheduling and deadline partitioning, while guaranteeing optimality. It proposes the utilization of some heuristics to improve their performance by reducing the number of migrations and preemptions. Our simulation results validate our approach and show a significantly reduced number of migrations and preemptions when compared to a previous basic version of such a scheduling algorithm[1].

1. Introduction

The utility of real-time systems are getting more and more common in the human life. From basic consumer products like mobile phones, microwaves up to automobiles and space sciences, they have got exceptional importance everywhere. This revolution requires increased processing speed. However, the speed of a single processor may not be increased beyond certain limits mainly because of high power consumption and too much heat dissipation [18]. It led the researchers to find some alternatives. The multiprocessing is one of the solutions to this problem where two or more processors are engaged to enhance the effective speed to carry out the workload in more comprehensive and efficient way. Currently multiprocessor architectures are easily available for embedded systems because the major manufacturers of processors are producing low priced multiprocessor architectures. The advent of multicore architecture where multiple processor cores are placed on a single chip, is quite significant in this regard. Real-time operating system or *RTOS* which manages such hardware architectures is drawing attention of the researchers.

The design of real-time systems is generally based on a set of concurrent periodic tasks with some hard timing requirements. The scheduler is an important part of the *RTOS* which is responsible for managing the available processing resources for the given taskset. In a multiprocessor system, it has to decide which task will execute at which time and on which processor, such that all tasks meet their timing constraints. The two main types of real-time multiprocessor scheduling strategies are: partitioned scheduling and global scheduling.

Partitioned scheduling is the most studied and relatively matured scheduling technique. In this type of scheduling, the taskset is partitioned and each task subset is allocated to a processor. After allocation of tasks to their processor, migration to a different processor is not allowed [16, 17]. Hence, after allocation, it becomes a simple problem of uniprocessor scheduling. However, the allocation of tasks to the processors is same as bin-packing problem which is known to be NP hard in strong sense [11]. The advantages of partitioning include low scheduling overhead, improved average cache performance and reutilization of uniprocessor algorithms. Main disadvantage of the partitioned scheduling is that it is inherently sub-optimal and therefore does not guarantee the complete utilization of the resources. There are some tasksets which are schedulable only when they are not partitioned.

In global scheduling, there is a single queue of ready tasks for all the processors. The scheduler is responsible for scheduling all the tasks. The migration of tasks between the processors is allowed. A task can execute on one processor and, after a preemption, can be resumed on a different processor. A task may be migrated after completing one complete invocation, called a job, or even during a job. This increases the schedulability which allows better utilization of resources. Moreover, global scheduling is better for dynamic systems where there is frequent arrival and leaving of tasks. An important advantage is that a number of global scheduling algorithms are found to be optimal for periodic tasks with implicit deadlines. An

[1]This work has been supported by the French Agence Nationale de la Recherche through the RESPECTED project (Contract ANR-2010-SEGI-002). See http://anr-respected.laas.fr

optimal algorithm for a task model is the one which can schedule all the tasksets of that model that can be scheduled by any other algorithm [6].

Proportionate fair, or simply PFair, was presented by Baruah et al. in 1996 [22]. It is a global scheduling algorithm which guarantees a 100% system utilization for periodic tasksets with implicit deadlines. It is based on the concept of fairness which means that each task gets a processor share proportional to its utilization factor at each instant. Thus at any time t, time allocated to a task T_i with utilization factor $T_i.u$ either is $\lfloor t * T_i.u \rfloor$ or $\lceil t * T_i.u \rceil$. In PFair, the time can be viewed as divided into small intervals of equal lengths called quanta. All the tasks are scheduled at the start of each quantum. It results that PFair achieves the optimality at a very huge runtime overhead [23] due to frequent scheduling points, migrations and preemptions. PF, PD and PD^2 are three PFair algorithms which are proven to be optimal [22]. ERFair [2], which is a work-conserving technique, and the PFair staggered model [14], which uses non-synchronized quanta for scheduling, are extended forms of PFair.

A number of recent papers have combined the notion of fluid scheduling with the one of deadline partitioning to still guarantee optimality while improving performance of PFair. In [19], such approaches are qualified as DP-Fair.

Boundary fair or BFair, is a scheduling technique proposed by Zhu et al. [27] that belongs to this category and that is optimal for periodic tasksets with implicit deadlines. The scheduling is done only at the boundaries which are the deadline times of tasks, and fairness is met only at those boundaries. It reduces the number of scheduling points as compared to that of PFair. BFair shows better performance than PFair when the tasks in the tasksets are fewer. When there are 100 tasks in the tasksets, Zhu showed [27] that BFair has only 48 % scheduling points when compared with PD, a well known PFair algorithm.

$LLREF$ [12] is also a DP-Fair technique based on an interesting abstraction of time and local remaining execution time. It is optimal for periodic tasksets with implicit deadlines. The scheduling is done at the boundaries similar to that of BFair. As it will be explained later, the differences come where the execution time units for the tasks are computed and allocated. $LRE-TL$ [10] which uses an improved dispatching technique and $E-TNPA$ [9] which deals with work-conserving scheduling are extensions of $LLREF$.

The study of effects of migration and preemption on different aspects of processor scheduling is quite common [5, 25, 26]. It is even more important in case of global scheduling which allows free migration between processors. The migrations and preemptions increase the schedulability because there are tasksets which are scheduled only when the tasks are allowed to migrate or preempt. But on the other hand, the main objection on global scheduling is the overhead due to the frequent scheduling points, migrations and preemptions. Migration causes cache miss and excessive load on the data buses between the processors while preemptions lead towards context switches. Theoretically the cost of preemption and migration is considered to be negligible but practically when these occur frequently their effect in the system cannot be neglected. On some modern multicore architectures, the cost of migration is much lower than in past but still a non-zero value. In short, frequent migrations and preemptions in the system lead towards an increase in worst case execution times of the tasks which results in the missing of deadlines.

Some researchers have worked in the domain of overhead control due to preemption and migration in global scheduling. [13] used the technique of delayed preemption for preemption control in non-optimal global scheduling. Aoun et al. [3] used the processor affinity technique to reduce the number of migration in PFair scheduling. Megel et al. [21] proposed a linear programming formulation and a local scheduler technique to reduce number of preemption and migration for optimal global scheduling algorithms.

The motivation behind this article is to explore and utilize simple heuristics to reduce overhead due to migration and preemption upto minimum possible level while keeping the optimality. The first heuristic controls the migrations while the second reduces the number of preemptions in the system. We have used these heuristics with a standard non-work conserving DP-Fair algorithm explained hereafter with the help of the time and nodal remaining execution time plane, or simply $TN-Plane$ abstraction model of Funaoka et al. [8]. The heuristics are described in accordance with this $TN-Plane$ model.

System Model. We consider that T is a set of N synchronous periodic tasks T_i where $i = 1, 2...N$ to be scheduled on M identical processors. Each task T_i has a period $T_i.p$ equal to its relative deadline $T_i.d$ (implicit deadline), an execution time $T_i.e$ and utilization factor $T_i.u$ (which is $T_i.e/T_i.p$) in range (0, 1]. U is sum of utilization factors of all the tasks in the taskset. Each task in such a system is released repeatedly in accordance with its period $T_i.p$. Each such invocation is called a job of the task. We assume that all the tasks are independent, i.e. they do not share any common resource and do not have any precedence with each other. The costs of migration, preemption and context switch are assumed to be already added in execution times. A processor can not execute more than one task at any given time and a single task cannot execute on more than one processor at any given time.

The rest of the paper is organized as follows. Section 2 gives an overview of DP-Fair by using the abstraction of $TN-Plane$. Section 3 discusses dispatching process inside a $TN-plane$. Section 4 presents the simple heuristics we propose and finally experimental results are given in section 5.

2. Overview of DP-Fair scheduling and TN-Plane

Time and Nodal remaining execution time Plane ($TN-Plane$) is a visual abstraction to represent execution of tasks on multiprocessors which allows insightful and analytical understanding. It has been introduced by Funaoka et al. [8]. The figure 1 shows the basic idea behind a $TN-Plane$. In this figure, time is represented on horizontal axis and task remaining execution time on vertical axis. The core idea for fair techniques of tracking the fluid schedule leads to represent it with a constant slope of $-T_i.u$ as shown by the dotted line. It corresponds to the ideal execution of the job. The practical execution, represented by the solid line, has a slope -1 when task is executing and slope 0 while task is waiting. The job released at t0 must finish its execution $T_i.e$ just up to its deadline $t0 + T_i.d$. Its execution between these two dates can be viewed as the movement of a token. The current location of the token at time t signifies the remaining execution time of the task at that time $T_i.e(t)$. When the job starts, the token has $T_i.e$ units to consume. When the job finishes, the token has nothing to consume because it is on the zero level.

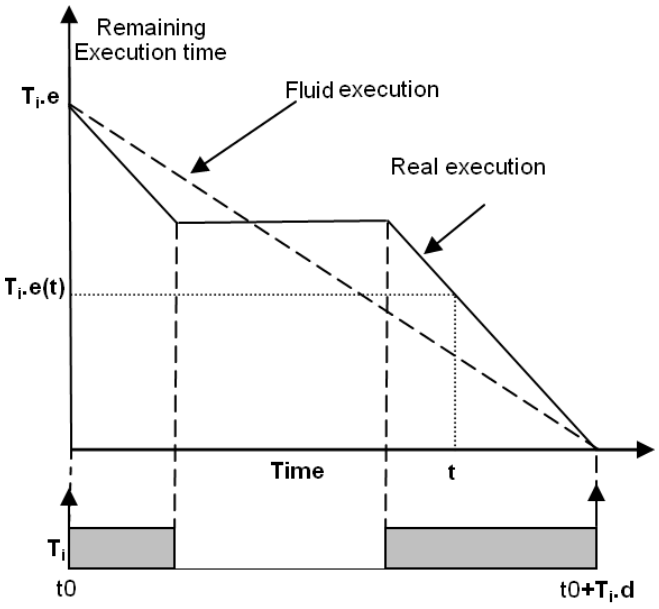

Figure 1. Representation of fluid and real executions.

When N tasks are considered in the system, their fluid schedules are made as shown in the figure 2. By recalling the result of Hong and Leung [15] establishing that no optimal on-line scheduler can exist for a set of jobs with two or more distinct deadlines on two or more processors, all DP-Fair strategies chose to subdivide time into slices where all the tasks have the same (local) deadline. The boundaries of these slices are all the deadlines (same as arrivals due to the implicit deadline assumption) of all tasks

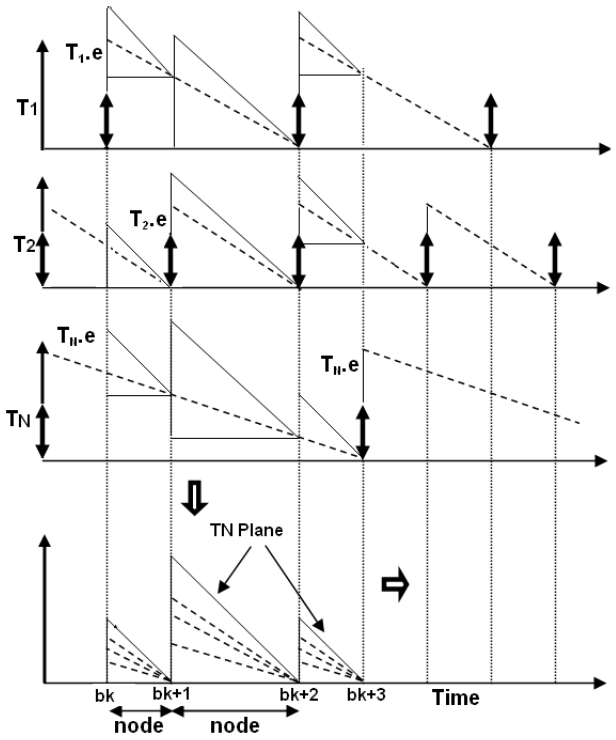

Figure 2. TN planes.

and are abbreviated as b_0,b_1...b_k etc. The distance between any two boundaries is also called a node as shown in figure 2.

A right angled isosceles triangle can then be considered between any two consecutive boundaries for each task. The right most vertex of the triangle coincides with the fluid schedule. Since triangles for all the tasks are of the same size in a node, task execution domains for all the tasks may be represented as a single overlapped isosceles triangle that constitutes a $TN-Plane$. Figure 2 shows three $TN-Planes$ between boundaries b_k and b_{k+3}. In a $TN-Plane$, the execution times on vertical axis for the tasks are not the actual remaining execution times of their jobs but are termed as nodal remaining execution times. These times are the local ones that are to be consumed by the end of current $TN-Plane$ so as the fluid execution for tasks is ensured at boundaries. The $TN-Plane$ makes it possible to envision the entire scheduling activity over time as scheduling in repeated $TN-Planes$ of various sizes, so that feasibly scheduling on a single $TN-Plane$ results in a feasible schedule for all $TN-Planes$ across time [4, 12, 19].

There are two aspects that need to be considered for scheduling in a TN-Plane: allocating nodal remaining execution times for all tasks, and dispatching these execution time units within the node among the processors.

- For allocation nodal remaining execution times, when only non work-conserving scheduling is considered, two main possibilities exist. First, the nodal remaining execution time for each task can be com-

puted exactly proportional to the task utilization factor. It is the case for $LLREF$ [10] where a task T_i with utilization factor $T_i.u$ gets $T_i.u*L$ units of execution time where L is the length of the current node. The resulting value may be a non-integer one which causes a practical problem (due to the hardware characteristics of processors, execution time unit numbers should be integral multiples of the highest precision timer). The second option counters this problem. BFair algorithm proposed by Zhu et al [27] allocates some mandatory execution time units to each task and one optional time unit to some eligible tasks, so as to approximate as much as possible the fluid executions. Thus nodal remaining execution times are defined as integer numbers.

- The dispatching within nodes can be either static or dynamic. For example, in the BFair algorithm [27], all the dispatching decisions are computed in advance for all the nodes over an hyper period (due to the periodic assumption). At runtime, they are simply implemented. For each node dispatching, it uses the McNaughton algorithm [20] that sequentially packs tasks to processors. Inversely, the $LLREF$ algorithm makes its dispatching decisions over time: at the beginning of each node, as well as at some secondary scheduling events that may occur within a node (as it will be explained in the next section).

This division of the scheduling process into two independent parts allows to concentrate on the two parts separately. It has resulted in the finding of some hybrid algorithms with improved characteristics as compared to original ones. Thus Cho et al. [7] have used the BFair for the computation of nodal remaining execution times as proposed by Zhu et al. [27] but have used a dynamic technique for dispatching tasks to processors inspired from the $LRE-TL$ [10]. They have shown that it avoids some unnecessary migrations.

The next section explains in detail the dispatching aspect of task scheduling inside a $TN-Plane$ since this is the part which deals with a major part of preemptions and migrations.

3. Dispatching within a TN-Plane

The execution of tasks in a single $TN-Plane$ is shown in figure 3. We recall that it is the overlapped isosceles triangle of all the tasks inside which their execution status can be shown as tokens. The left side represents the nodal remaining execution time while the oblique side of the $TN-Plane$ is called No Nodal Laxity Diagonal (or in short $NNLD$). Nodal remaining execution time of task T_i is noted as $T_i.l$. The scheduling objective is to make all tokens arrive at the rightmost vertex of the $TN-Plane$ with zero nodal remaining execution time. Such an arrival is said nodally feasible.

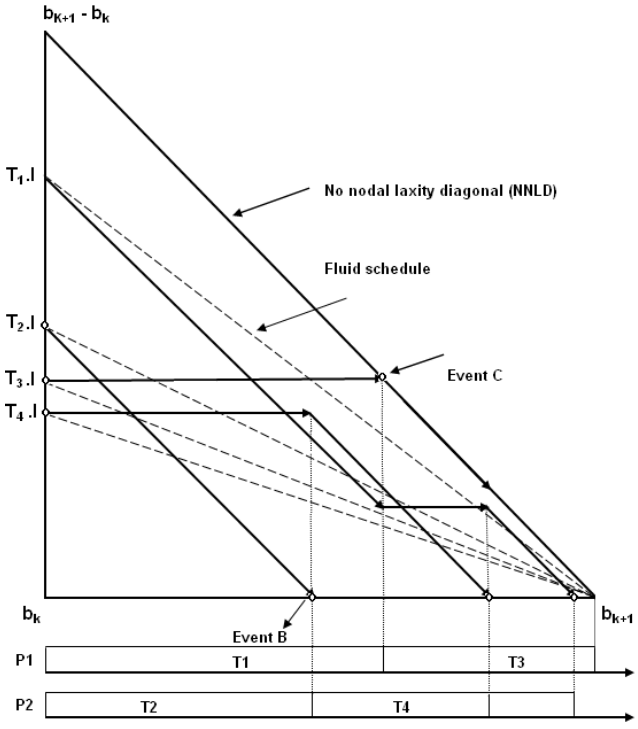

Figure 3. Taskset scheduling in a TN-Plane on two processors: P1 and P2.

At any point, at most M tasks can be executed on the M processors, i.e. at most M tokens can move simultaneously diagonally. The task switching in $TN-Plane$ comes when a token hits either the domain with the zero nodal remaining execution time (bottom hitting event or event B) or hits the $NNLD$ (ceiling hitting event or event C). These events B or C define secondary scheduling events. The following general rules have to be observed while scheduling within a $TN-Plane$ (demarcated by the boundaries b_k and b_{k+1}):

- Task with zero nodal laxity is given maximum priority. A running task (with non zero nodal laxity) has to be preempted and replaced by the previous one. Actually, when a token hits that side, it implies that the task does not have any local laxity. Thus, if it is not selected immediately, then it cannot satisfy the objective of nodal feasibility.

- Stop the task when its nodal remaining execution time is completely consumed (in case of non work-conserving context). The task is preempted and is replaced by another possibly waiting task.

- Do not let more than $M-\sum_{i=1}^{N}(T_i.u)(b_{k+1}-b_k)$ units of time idle between b_k and b_{k+1}.

While respecting these rules, the scheduling of a taskset in the $TN-Plane$ is nodally feasible if and only if the sum of nodal remaining utilization factors of all the tasks is less than or equal to the capacity of the processors [7].

$$\sum_{i=1}^{N}(\frac{T_i.l(b_k)}{b_{k+1}-b_k}) \leq M \quad (1)$$

If this condition is not satisfied then more than M tokens strike against $NNLD$ simultaneously out of which only M tokens are selected to execute and the rest move out of the $TN-Plane$. This point of arrival of more than M tasks on $NNLD$ is also termed as critical point. The taskset is not nodally feasible if there is any critical point in the $TN-Plane$.

4. Overhead control

Once the nodal execution times for the taskset are computed and the simple scheduling rules are established, one can see that there is still room to design complementary dispatching strategies so as to reduce the number of task preemptions and migrations. Our intention is to explore this space and to evaluate the resulting improvement in term of overhead. Our proposal has been guided by two observations and gives rise to two related heuristics. These heuristics are used with an optimal global scheduling algorithm called DP-Fair. The DP-Fair uses BFair algorithm [27] for computation of nodal remaining execution time units while it uses the dynamic dispatching technique discussed in section 3 for task allocation. The first heuristic results from the standard assumption of instantaneous preemptions and migrations. Then in theory it does not matter which processor is hosting a given task, but only which tasks are running at a given time. That is why most algorithms give no explicit prescription about how to assign tasks to processors. Thus, heuristic 1 deals with the task to processor assignment criterion. The second one comes from the fact where it has been shown that the order in which the M tasks are selected for execution is not important, provided that they have non-zero nodal remaining execution times [10]. Thus, heuristic 2 is concerned with the running task selection criterion. The heuristics are described with the same notations as used by Funk et al. [10].

4.1. Heuristic 1

Usually, running tasks are assigned to the available processors without considering their previous histories. According to heuristic 1, a task keeps the record of the processor on which it was executed last time and then an affinity relation exists between task and processor. Heuristic 1 takes into account this relation and if possible, tries to assign a newly running task to the same processor on which it was scheduled the last time. This heuristic is applied at main scheduling points i.e. those that coincide with time boundaries (or start of a $TN-Plane$) as well as at secondary scheduling events (event B or event C occurrences).

Algorithm
Suppose

- H_B- List of running tasks. Maximum size of H_B is M
- $getLastProc()$- returns the processor on which task was executed last time
- P- An object that represents a processor

1. for(each newly task T inserted into H_B)
2. $P = T.getLastProc()$;
3. if (P is idle)
4. execute T on P;
5. else
6. execute T on any idle processor;
7. end for

The computational complexity of heuristic 1 is O(M).

4.2. Heuristic 2

At a scheduling point in a $TN-Plane$ all the ready tasks have equal priority and atmost M of them can be chosen arbitrarily for execution. Heuristic 2 attempts to control the preemptions at these scheduling points. According to this technique, the tasks executing on a processor just before the scheduling are given priority to reexecute provided they still are ready. By continuing such executions, some unnecessary preemptions are avoided. This heuristic is applied at main scheduling points.

Algorithm
Suppose

- H_B- List of running tasks. Maximum size of H_B is M
- $ReadyList$- is the list containing unsorted ready tasks

Inputs H_B, $ReadyList$
Outputs H_B

1. for (i= 1 $\rightarrow H_B$.size)
2. T=H_B.getTask(i);
3. if($T \epsilon ReadyList$)
4. $ReadyList$.remove(T);
5. else
6. T.preempt();
7. H_B.remove(T);
8. if ($ReadyList$.size() !=0)
9. T = $ReadyList$.getfirst();

10. $H_B.add(T)$;

11. end for

12. if(H_B.size < M)

13. As usual take tasks from $ReadyList$ if available and complete the H_B

The computational complexity of heuristic 2 is $O(M)$. The for loop checks each task in the H_B that contains tasks that were previously running at b_k . If it contains a task which is still ready in the next node (line 3), it will keep this task, otherwise it is removed from the running task list(line 7).

5. Experimentation

We performed a series of simulation based experiments to find the effect of using the above mentioned heuristics with DP-Fair. We used STORM [1, 24] as simulation tool. STORM is a freeware software tool developed in our research team. It is able to simulate the behavior of predefined or user defined real-time multiprocessor scheduler and to evaluate their performance by computing specified metrices on the schedules they construct. Tasksets with uniformly distributed periods and execution times with constant value of taskset utilization factor were obtained using a MATLAB program. STORM takes these tasksets as input and schedules each of them according to a given scheduling algorithm. We observed the scheduling of each taskset upto the LCM of the periods of the tasks. The number of tasks in each taskset was in the range of 2.1 and 2.5 times the number of processors. The interval for the task periods is (3, 20]. A point on a graph is drawn by taking an average of results of experiments on 30 tasksets. The vertical axis of each graph represents the specified parameter of an algorithm as an average of the same parameter of DP-Fair as a function of number of processors on the horizontal axis.

Each graph represents, for the studied parameter, the average ratio (in percentage) of its value given by the DP-Fair algorithm applying the concerned heuristic over the one given by the original DP-Fair algorithm, as a function of number of processors on the horizontal axis.

For the first set of experiments, a multiprocessor system with 2, 4, 6, 8 and 10 processors was considered. It compares the effects of using different heuristics with DP-Fair algorithm. The sum of utilization factor of taskset was equal to the number of processors. We first used heuristic 1, then heuristic 2 and finally both the heuristics with the DP-Fair to find their effect on the migrations and preemptions. When we used both the heuristics in sequence, we called the resulting algorithm as hybrid algorithm. The order of the heuristics does not make any difference. The results of this set of experiments are given in figure 4 and figure 5.

The results show that heuristic 1 reduces more than 50 % of the migrations of original algorithm. It does not affect the number of preemptions. On the other hand, heuristic 2 not only reduces the number of preemptions by 15 % on average but also reduces the migration of tasks to some extent. The reduction in migration is due to the fact that tasks which continue their executions due to heuristic 2 may have been migrated to different processors in case of preemption. Hybrid algorithm keeps the advantages of both heuristic 1 and heuristic 2.

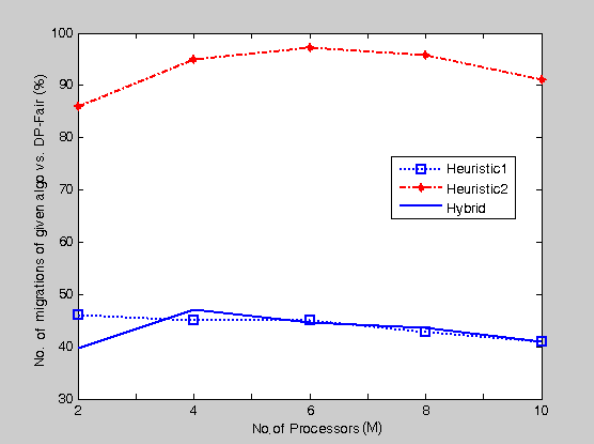

Figure 4. Migration control

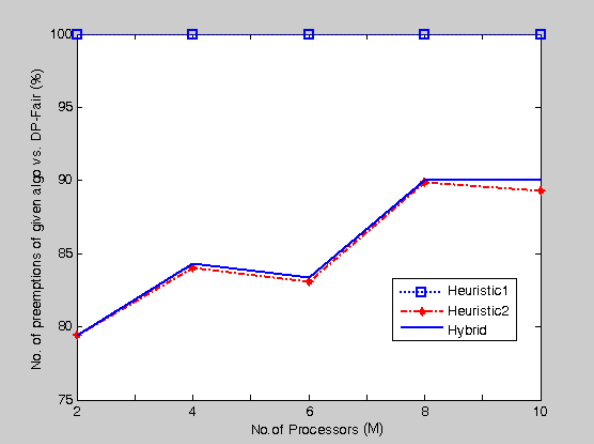

Figure 5. Preemption control

In the second set of experiments, we tested the performance of hybrid algorithm varying the sum of utilization factors U. A multiprocessor system with 2, 4, 6 and 8 processors was considered. The graphs in the figure 6 and figure 7 show the results. The hybrid algorithm improves the migration control with a reduction in utilization factor. Smaller the utilization factor, higher the probability for a processor to get free and better are the chances for the tasks to re-execute on the same processor.

With relatively lower utilization factor, the number of tasks which have consumed their nodal execution time before the main scheduling points (due to the non work conserving behaviour of the scheduling) increases letting idle units before these points. The algorithm gets disable in this situation and finally preemption is not controlled.

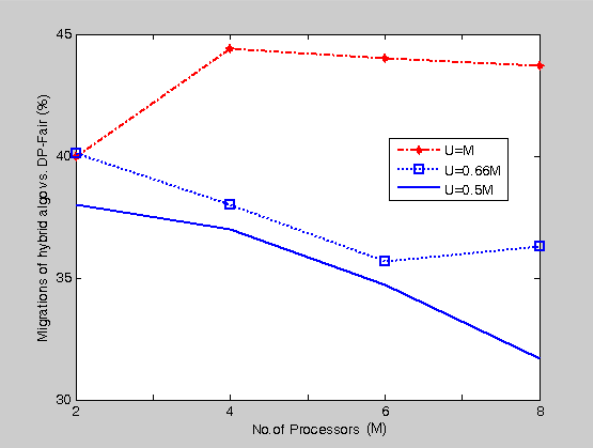

Figure 6. Migration control of hybrid algorithm with variable utilization factor

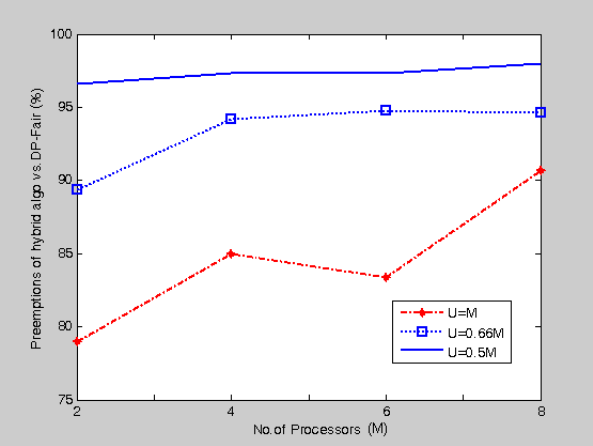

Figure 7. Preemption control of hybrid algorithm with variable utilization factor

6. Conclusion

In this article, we have proposed the utilization of a couple of heuristics to improve the performance of optimal global scheduling by reducing the number of migrations and preemptions. Our simulation results have validated our approach and showed a significant reduction in the number of migrations and preemptions when compared to a previous basic version of such a scheduling algorithm. We tried various heuristics and finally we found the combination of above mentioned heuristics to be quite efficient. There may exist some more methods for the improvement. Experimentation with addition of some more techniques may be interesting to further enhance the efficiency in terms of overhead reduction. Utilization of such overhead controlling techniques may increase the acceptance of global scheduling because overhead is the main objection on these algorithms.

References

[1] http://storm.rts-software.org.

[2] J. Anderson and A. Srinivasan. Early-release fair scheduling. *In the Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 35–43, 2000.

[3] D. Aoun, A-M Déplanche, and Y. Trinquet. Pfair scheduling improvement to reduce interprocessor migrations. *In the Proceedings of 16th International Conference on Real-Time and Network Systems*, 2008.

[4] K. Bletsas B. Andersson and S. K. Baruah. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. *In the Proceedings of the Real-Time Systems Symposium*, pages 385–394, 2008.

[5] A. Block and J. Anderson. Accuracy versus migration overhead in real-time multiprocessor reweighting algorithms. *In the Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 355–364, 2006.

[6] G. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, 2004.

[7] H. Cho, Binoy Ravindran, and E. Douglas Jensen. T-l plane-based real-time scheduling for homogeneous multiprocessors. *Journal of Parallel and Distributed Computing*, pages 225 – 236, 2010.

[8] Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. Energy-efficient optimal real-time scheduling on multiprocessors. *In the Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 23–30, 2008.

[9] Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. *In the Proceedings of the 20th Euromicro Conference on Real-Time Systems ECRTS*, pages 13–22, 2008.

[10] S. Funk and Vijaykant Nadadur. Lre-tl an optimal multiprocessing scheduling algorithm for sporadic task sets. *In Proceedings of the 17th International Conferenece of Real-Time and Network systems Paris*, pages 26–27, 2009.

[11] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[12] B. Ravindran H. Cho and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. *In the Proceedings of the IEEE International Real-Time Systems Symposium*, pages 101–110, 2006.

[13] Chiahsun Ho and Shelby H. Funk. A hybrid priority multiprocessor scheduling algorithm, work in progress. *In the Proceedings of the 31st Real-Time Systems Symposium (RTSS)*, 2010.

[14] P. Holman and J. Anderson. The staggered model: Improving the practicality of pfair scheduling. *In the Proceedings of the 24th IEEE International Real-Time Systems symposium(RTSS'03)*, 2003.

[15] K.S. Hong and J.Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41:1326–1331, 1992.

[16] L. George J. Goossens I. Lupu, P. Courbin. Multi-criteria evaluation of partitioning schemes for real-time systems. *In the Proceedings of the 15th IEEE International Conference on Emerging Techonologies and Factory Automation*, 2010.

[17] J.L. Diaz D.F. Garcia J.M. Lopez, M. Garcia. Utilization bounds for multiprocessor rate-monotonic scheduling. *Real-Time Systems*, 24:5–28, 2003.

[18] D. Lammers. Intel cancels tejas, moves to dual-core designs. *EE Times*, May 7th 2004.

[19] G. Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. Dp-fair: A simple model for understanding optimal multiprocessor scheduling. *In the Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 3–13, 2010.

[20] R. McNaughton. Scheduling with deadlines and loss functions. *Management Sciences*, 6:1–12, 1959.

[21] Thomas Megel, Renaud Sirdey, and Vincent David. Minimizing task preemptions and migrations in multiprocessor optimal real-time schedules. *Proceedings of the31st IEEE Real-Time Systems Symposium*, pages 37–46, 2010.

[22] N. C.G.Plaxton S.Baruah and D.Varvel. Proportionate progress:a notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[23] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *In Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 189–198, 2001.

[24] R. Urunuela, A-M. Déplanche, and Y. Trinquet. Storm - a simulation tool for real-time multiprocessor scheduling evaluation. *In the Proceedings of IEEE International Conference on Emerging Technology and Factory Automation, ETFA, Bilbao*, 2010.

[25] C. Y. Yang, Jian-Jia Chen, and Tei-Wei Kuo. Preemption control for energy-efficient task scheduling in systems with a dvs processor and non-dvs devices. *In the Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 293–300, 2007.

[26] G. Yao, Giorgio Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with fixed preemption points. *In the Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 71–80, 2010.

[27] D. Zhu, Daniel Mosse, and Rami Melhem. Multiple-resource periodic scheduling problem: How much fairness is necessary? *In the Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 142–151, 2003.