Improving Multiprocessor Average-Case Schedulability using A Modified Global Dual Priority Algorithm

Josep M. Banús *{jbanus@etse.urv.es}*, Alex Arenas and Jesús Labarta §

Departament d'Enginyeria Informàtica i Matemàtiques, Universitat Rovira i Virgili
§ Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya

Abstract

In this paper we present a modification of the Dual Priority Scheduling Algorithm to work on shared memory multiprocessor systems improving the average-case schedulability. The proposal deals with global fixed-priority preemptive scheduling of periodic tasks on identical processors. The algorithm allows to schedule hard real-time periodic tasks using task migration between different processors. Using this approach we are able to schedule task sets that cannot be scheduled via traditional partitioning methods. Extensive simulations show that the proposed algorithm gives higher success ratios than previous global scheduling schemes and traditional partitioning methods.

Keywords: real-time, multiprocessors, schedulability, global scheduling, periodic tasks

1. Introduction

Multiprocessor systems have evolved rapidly in the last years. At the same time, the use of these powerful computing resources in real-time systems has opened several problems concerning scheduling strategies. The problem of determining when and where a given task must execute without missing its deadline or compromising other task deadlines in multiprocessor systems often becomes intractable. Besides, when the scheduling is possible, algorithms that are optimal for uniprocessor systems are not necessarily optimal when the number of processors increases [1] (it is well known that optimal scheduling for multiprocessors systems is a NP-Hard problem). Nevertheless, the great availability of these systems has made them interesting for the real-time community and the research in this area has been reactivated in the last years. Usually, two alternatives are proposed to schedule tasks in these systems: (i) local scheduling or partitioning method; this methodology first allocates statically periodic tasks to processors and, after that, an optimal uniprocessor scheduling algorithm is used individually on each processor. And (ii) global scheduling or non-partitioning method; in this case there is a global scheduler that dynamically binds periodic tasks to processors, obtaining dynamic load balancing, fault tolerance, etc.

Traditionally the first method is used, mainly because it takes advantage of well-known uniprocessor strategies and because often the average-case performance is higher than the average-case of the global scheduling.

However, recently the second method is receiving more attention from the research community that has evaluated the differences between both alternatives. More of them have dealt with utilization upper bounds for the Global Rate Monotonic Scheduling (GRMS) using these bounds as a necessary schedulability condition and to perform new tasks admission control. Unfortunately these upper bounds are too pessimistic and produce low processor utilizations. Hence, to find schedulability in heavy loaded systems the straight solution consists in their simulation. In particular, in [2] it is showed that the partitioning method is not necessarily the best approach.

Nevertheless, the global scheduling method has some important drawbacks: no efficient schedulability tests exist, no optimal priority-assignment is known, multiprocessor anomalies appear and computational complexity increases. Some of these drawbacks can be avoided using heuristics.

In this paper we have modified the Dual Priority algorithm [3-5], that we will call *Modified Global Dual Priority* (MGDP), to use the spare capacity of processor to serve some selected periodic tasks instead of serving aperiodic tasks. With this algorithm every task execution might run at two different priority levels and on different processors. This gives the scheduler flexibility to fit tasks into processor capacities increasing the utilization.

2. Framework and Assumptions

We consider a real-time multiprocessor system with *m* symmetrical processors and shared memory. All tasks are considered independent and can be preempted at any time. Each task τ_i has a worst-case computation requirement C_i, a period T_i and a deadline D_i, assumed to satisfy $D_i \leq T_i$. The worst-case analysis is assumed to include the initial cache misses (i.e., every execution of a task starts in an empty state). Therefore, the first execution of every instance will be considered neither a preemption nor a migration. Each periodic task has an utilization factor defined as $u_i = C_i/T_i$. The utilization of the task set is $U=\sum u_i$ and is assumed to be lower or equal than the capacity of the system ($U \leq m$). Finally, we assume all overheads for context switching, task scheduling, task pre-emption and migration to be zero. Note that we also assume that every

task executes its worst-case execution time. Nevertheless, in some experiments we will consider also the number of preemptions and migrations in order to compare different algorithms.

3. Global Dual Priority Scheduling

The Dual Priority (DP) uniprocessor scheduling algorithm has many good properties (to mention one, it achieves very good mean aperiodic response times). However, the most important characteristic is that, although it is a fixed priority system, it almost has the flexibility of dynamic priority systems.

The *Dual Priority* Algorithm for uniprocessors is based on the offline computation of periodic tasks worst-case response time, using a recurrent formula (1), which in turn is also used as a scheduling test [6].

$$W_i^{n+1} = \sum_{j \in hp(i)} \left\lceil \frac{W_i^n}{T_j} \right\rceil C_j + C_i \qquad (1)$$

This recurrent formula is used pre-runtime to compute the worst-case response time of periodic tasks (the final W_i is the computation requirement of task τ_i and all higher priority tasks (*hp(i)*)) interference. The key element of this approach is that the periodic task τ_i can be delayed until its promotion time ($=D_i - W_i$) and still be guaranteed to meet its deadline. During runtime, a periodic task starts with a low priority level (lower than aperiodic tasks) and at promotion time its priority is raised to a higher level where it will only receive the interference of other higher priority promoted tasks, guaranteeing its deadline. This provides the scheduling algorithm with a duality in the treatment of the tasks depending on the interferences, and therefore it inherits some good characteristics from fixed priority algorithms and some from dynamic priority algorithms. This duality could be well adapted to multiprocessors systems, for example to serve aperiodic requests or to increase periodic task sets schedulability. In fact, the original uniprocessor DP [4] purpose was to increase the periodic utilization of the processor. This was done by establishing a promotion to a higher priority for periodic tasks that otherwise would miss their deadline with the rate-monotonic priority assignment. Our idea is that within the multiprocessor domain it is possible to increase the number of schedulable task sets even more because conflictive periodic tasks (or the higher priority tasks that interfere with them) may use other processors spare capacity.

Unfortunately, formula (1) is not valid for multiprocessors. Therefore, we propose to use the DP in a hybrid conformation between local and global scheduling (MGDP). At design time all periodic tasks are statically distributed among processors and their promotion times are computed. At run time it is possible to execute periodic tasks in any processor during a period of time (dynamic phase) and execute them into a predefined processor after their promotion time (static phase) (see Figure 1). Hence, there are two priority levels: the Low Priority Level (LPL) and the High Priority Level (HPL). Accordingly, every task has two priorities, one of each band, but only one is active: at start time it has its LPL and after the promotion it has its HPL. The priority assignment in the LPL can be arbitrary. Usually we establish a Global Rate Monotonic assignment. On the other hand, the HPL must use local Rate Monotonic order to guarantee task deadlines.

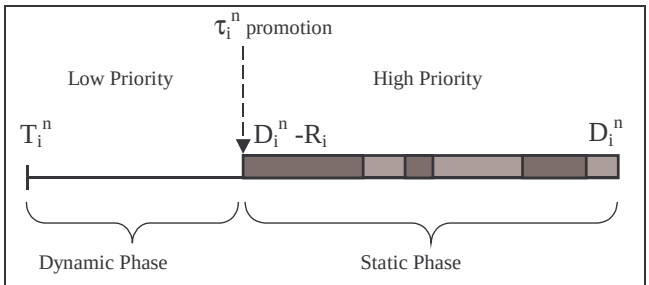

Figure 1: Periodic tasks allocation phases in MGDP

The design time distribution of periodic tasks among processors is performed by an adapted version of RM-FFDU partitioning algorithm to Dual Priority.

First, we sort the task set using rate-monotonic order and we set the LPL. After that, we use RM-FFDU [7] to pre-allocate tasks to processors. In heavy loaded systems, there will be some remaining tasks impossible to allocate. In this case, we assign the remaining tasks to processors with the lowest utilizations, overloading them. Obviously overloaded processors will not pass the schedulability test but we will tag this task as *not-guaranteed* and the processor as *overloaded*. At run-time other processor spare capacity will help these *not-guaranteed* tasks to meet their deadlines.

After that, we set a rate-monotonic priority assignment in HPL for local tasks. The following step consists in computing the promotion times using the uniprocessor formula (1) and setting it to zero for not-guaranteed tasks, i.e. immediate promotion. At the same time, to meet a *not-guaranteed* task deadline we give its higher priority tasks preference to be executed in other processors. Therefore, we assign them the higher priorities in the LPL. By doing this we reduce the higher priority tasks interference and therefore *not-guaranteed* tasks may possibly finish on time. We tag these high priority tasks as *selected-tasks*.

The runtime process is illustrated on Figure 2. When a task arrives it is queued in a Global Ready Queue (GRQ). In this queue, *selected-tasks* have higher priority than the other periodic tasks. They are queued according to a global rate-monotonic priority assignment relative to all *selected-tasks*. The remaining periodic tasks are sorted according to their fixed low priority. The global scheduler (GS) selects the first *m* tasks from this queue to execute on the *m* processors. Additionally, there are *m* High Priority Local Ready Queues ($HPLRQ_i$ with *i* in [1..*m*]) used to queue promoted periodic tasks. When a periodic task τ_{pi} is

promoted it is moved from the GRQ to its corresponding processor $HPLRQ_p$. Processors with promoted periodic tasks are not allowed to execute tasks from the GRQ. Note that a promotion implies a change in priority and can cause a pre-emption. At promotion time, a periodic task must execute in its originally designated processor, where it will only receive the interference of the higher priority promoted tasks, guaranteeing its deadline. Note that sometimes this condition will cause a promoted task to migrate from a processor to its designated processor.

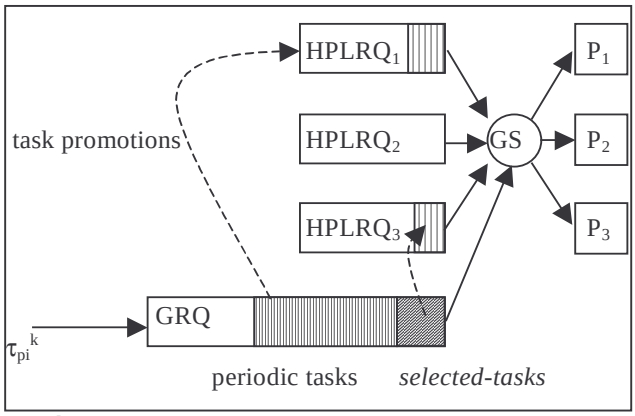

Figure 2: DP Global Scheduler. Squares represent processors, circles schedulers and rectangles queues. For any processor P_i , if $HPLRQ_i$ is not empty P_i executes the first promoted task. Otherwise P_i executes the first task from GRQ.

Within this scheme, while a periodic task is not promoted, it can be executed in any processor. This reduces the number of periodic task waiting for a specific processor, taking advantage of idle processors, advancing periodic work and making the system ready for future 'overloads'.

4. Results and Discussion

In this section we perform an average-case performance evaluation comparing the global scheduling detailed in the previous section (MGDP) versus both global and local schedulers from the literature [2]. The evaluation methodology we have used is based on the simulation of extensive randomly generated synthetic task sets. We test the average-case performance and robustness of the scheduling algorithms under an extent range of situations.

We use the *Success Ratio* (the fraction of all generated task sets that are successfully scheduled with respect to an algorithm) as the performance measure for average-case performance. Unless otherwise stated, we have used the same experimental setup used in [2] (m=4, n=uniform(4,12), $T_i \in \{100,200,300,...,1600\}$, $E[u_i]$=0.5, $stddev[u_i]$=0.4). The only difference is that our synthetic task generator discards the unfeasible cases (where the total load generated is greater than the available capacity) because it would introduce a distortion of the real algorithm's performance. For simulations in figures 3 to 6 the success ratio is computed as the average of 2,000,000 task sets. Hence, with a 95% confidence, we obtain an error of the success ratio that is less than 0.1%.

We have used two versions of a partitioning algorithm (RM-FFDU) as a reference for the non-partitioning algorithms. One version uses the sufficient schedulability test from Liu and Layland [8] (with polynomial time complexity) and the other (RM-FFDU+respan uses the necessary and sufficient schedulability test from Joseph and Pandya [6] (with pseudo-polynomial time complexity). Obviously, the later needs more offline computational time but achieves higher success ratio, being a good reference. This partitioning method was found in [2] to achieve the highest success ratio for the proposed experimental setup. We have also simulated three global schedulers: GRMS, AdaptativeTkC and MGDP. All of them use the 'preemption-aware' characteristic described in [2], i.e., all the dispatchers take into account the previous state to minimize the number of preemptions.

In Figure 3 we can see that RM-FFDU+respan outperforms all global schedulers but MGDP. This is so because the success ratio for MGDP includes all cases scheduled by RM-FFDU and a portion of the remaining. When the number of processors increases there are more resources available and therefore all algorithms increase their success ratio. We observe a crossover from two to three processors. Hence, we have analyzed the synthetic task sets generated for two processors. We have concluded that the generator has problems with this parameter set and it tends to generate easy to schedule task sets (i.e., with low loads) and this increases the success ratio. We have also conducted simulations fixing the number of processors to four and varying the rest of parameters, but their figures are not included for the sake of space. Increasing the number of tasks or their utilization is equivalent to reducing the number resources (because the number of processors remains the same). In these situations MGDP behaves better than the rest because it has two priority levels and breaks periodic task execution in two phases and they are easier to fit into a resource (i.e. a processor). For example, for E[U]= 90%, MGDP success ratio is 70% and with AdaptativeTkC is 35%. For E[n]= 12, MGDP success ratio is 75% and with AdaptativeTkC 39%. In Figure 4 we have focused on task sets with heavy loads because they are the most difficult to schedule. We have used the same synthetic task set generator and parameter set but we have classified the resulting tasks sets according to the total load generated. First of all, we observe that the range 85%-90% is the average-case upper bound as it happens in the uniprocessor domain. Second, the partitioning algorithm has a sudden drop at 85% because the discretization of the bin and task sizes. It usually reaches a point when tasks have sizes that do not fit into any processor. On the other hand, MGDP is able to fit tasks to processors exceeding its total capacity (i.e. can overflow the bins). This allows to maintain high success

ratios until a point (95% approximately) where the total load is so high that idle processors cannot help overloaded processors.

To be more realistic, the preemption and migration costs should be considered. For the experiments in Figure 5 we have simulated 5,000 task sets for each point and we have measured the *preemption and migration densities* (the number of preemptions or migrations divided by the length of the simulation respectively). To be fair we have only considered the task sets that were schedulable by all the algorithms. For low periodic loads the Dhall's effect is observed: partitioning algorithms have more preemptions than non-partitioning algorithms. On the other hand, MGDP has more preemptions than AdaptativeTkC. This is because tasks can be preempted in both phases, the dynamic and again in the static phase. These two phases give MGDP flexibility and therefore a higher success ratio. However, if preemptions and migration costs were significant it would reduce the success ratio achieved. In a future work we will study the way of reducing the number of preemptions in MGDP.

Acknowledgements

We thank Professor Björn Andersson for useful discussions. This research is supported by MCYT project number TIC2001-0995-C02-01.

References

[1] Dertouzos, M.L., Mok, A.K., "Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks", IEEE Transactions on Software Engineering, v.15, n.12, pp. 1497-1506, 1989

[2] Anderson, B., Jonsson, J., "Fixed-Priority Pre-emptive Multiprocessor Scheduling: To Partition or not to Partition". Real-Time Computing Systems and Applications, pp. 337-346, 2000

[3] Davis, R., Wellings, A., "Dual Priority Scheduling", Real-Time Systems Symposium, pp. 100-109, 1995

[4] Burns, A., Wellings, A.J., "Dual Priority Assignment: A Practical Method for Increasing Processor Utilization", Proceedings of the Fifth Euromicro Workshop on Real-time Systems, pp. 48-53, 1993

[5] Banús, J.M, Arenas, A., Labarta, J., "Dual Priority Algorithm to Schedule Real-Time Tasks in a Shared Memory Multiprocessor", Workshop on Parallel and Distributed Real-Time Systems, 2003

[6] Joseph, M., Pandya, P., "Finding Response Times in a Real-Time System", British Computer Society Computer Journal, 29(5): 390-395, Cambridge University Press, 1986

[7] Davari, S., Dhall, S.K., "An On Line Algorithm for Real-Time Task Allocation", Real-Time Systems Symposium, pp. 194-199, 1986

[8] Liu, C.L., Layland, J.W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", Journal of the Association for Computing Machinery, vol. 20(1), pp. 46-61, 1973

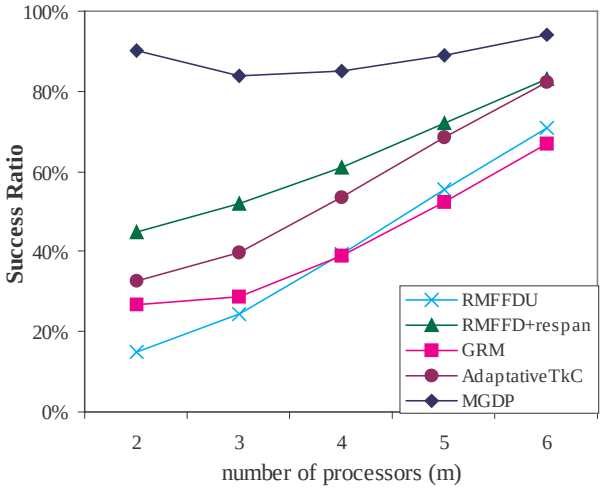

Figure 3: E[n]=8 E[u]=0.5 stddev[u]=0.4

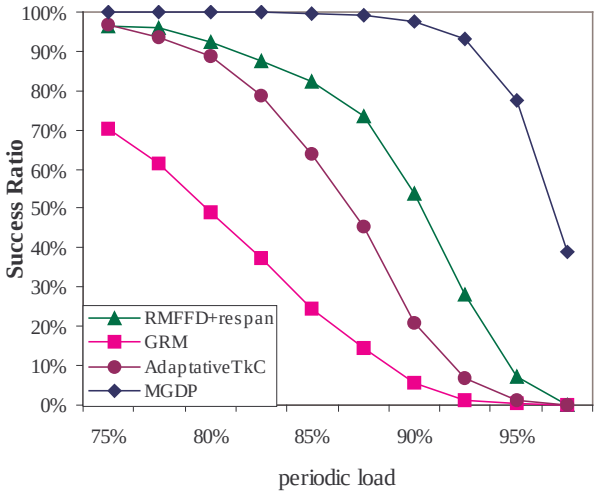

Figure 4: m=4 E[n]=8 E[u]=0.5 stddev[u]=0.4

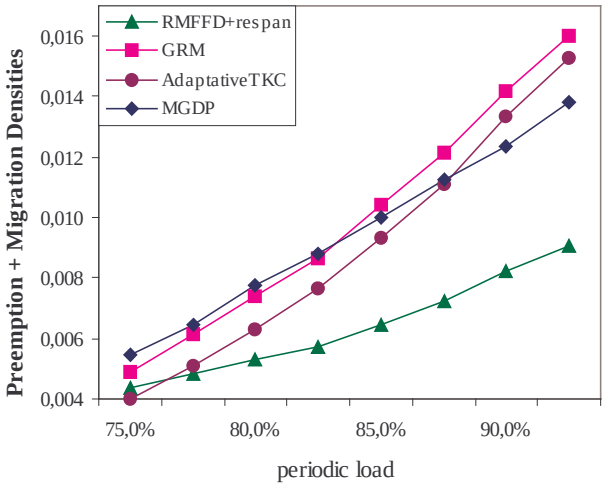

Figure 5: m=4 E[n]=8 E[u]=0.5 stddev[u]=0.4