Dual Priority Algorithm to Schedule Real-Time Tasks in a Shared Memory Multiprocessor

Josep M. Banús, Alex Arenas and Jesús Labarta §

Departament d'Enginyeria Informàtica i Matemàtiques, Universitat Rovira i Virgili
§ Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya
{jbanus@etse.urv.es} *{aarenas@etse.urv.es}*

Abstract

In this paper we present an adaptation of the Dual Priority Scheduling Algorithm to schedule both hard real-time periodic tasks and soft-aperiodic tasks in shared memory multiprocessor systems. The goal is to achieve low mean aperiodic response times while meeting all periodic task deadlines. Our proposal allows periodic and aperiodic tasks to migrate to other processors to improve aperiodic mean response time. We show via extensive simulations that our approach gives better results than local Slack Stealing schedulers.

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 [1,2]. 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 [3] (it is well known that optimal scheduling for multiprocessors systems is a NP-Hard problem [4]). 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; this methodology first allocates periodic tasks to processors and, after that, an optimal uniprocessor scheduling algorithm is used individually on each processor [1]. And (ii) global scheduling or dynamic binding; in this case there is a global scheduler that dynamically binds periodic tasks to processors. Recent works have evaluated the differences between both alternatives [5], comparing the number of schedulable task sets.

Joint scheduling of real-time periodic, sporadic and soft-aperiodic task sets (from now on we will use the term aperiodic task referring to soft-aperiodic tasks) has been extensively studied for uniprocessor systems. For example the Deferrable Server [6], the Sporadic Server [7] and the Slack Stealing Algorithm [8,9] (that offers an optimal scheduling strategy) solve this problem. Unfortunately, the applicability of these scheduling algorithms to the joint multiprocessor system is not straightforward.

A common feasible method in multiprocessors systems consist in to partition periodic tasks among processors statically and after to use a well-known uniprocessor scheduling algorithm as a local scheduler. Besides, it is also allowed migration of aperiodic tasks to any processor [10,11,12]. In this scenario, the periodic task deadlines are guaranteed and aperiodic tasks achieve good response time because the migration algorithm is intended to allocate the task to the most adequate processor. This method is specially interesting for processors with unbalanced loads because migration favors a more efficient use of the whole system. However, it is also useful for balanced loads because while a processor must execute periodic tasks in order to meet their deadlines other processors may be idle or may have enough laxity to execute aperiodic tasks. In a previous work [13] we have tested this scenario with Slack Stealing local schedulers for periodic tasks, allocating aperiodic tasks to processors with a global scheduler. The better results were achieved using a Next-Fit distribution strategy for the global aperiodic scheduler. However, we noticed that the results could be improved mainly because the periodic load is unbalanced: with local schedulers a processor may be in a very busy period while other processors may be nearly idle.

In this paper we propose to use a global scheduler for both periodic and soft-aperiodic tasks. The goal is to obtain better mean aperiodic response times by allowing

periodic tasks to migrate among processors. This scheme balances both periodic and aperiodic load. Therefore, with a global scheduler periodic tasks are able to advance work in any processor, increasing the near future readiness to serve aperiodic requests. The existing multiprocessor global schedulers (RM [23], EDF [22] and AdaptativeTKC [5]) do not deal with aperiodic tasks. Furthermore, there are not schedulability tests for these global schedulers. The usual schedulability test is performed via simulation, because the deterministic nature of periodic tasks. The consideration of aperiodic tasks introduces stochastic factors and invalidates the simulation tests. For this reason, it is necessary to adapt or design a global scheduler able to guarantee periodic task deadlines and able to serve aperiodic requests.

Within the uniprocessor domain, the *Slack Stealing Algorithm* (SS) [8,9] is optimal in the sense that it minimizes the response time of every aperiodic task among all the uniprocessor scheduling algorithms that meet all periodic task deadlines. Due to optimality, SS is a good candidate to be adapted to multiprocessor scheduling. The algorithm relies on the exact schedulability conditions given in [24]. It computes offline idle processor periods throughout the whole hyperperiod and stores these values in a table. At runtime, the scheduler uses this table to determine the availability to execute aperiodic tasks. Unfortunately, it is not suitable as a global scheduler because it is not possible to compute a global table. Furthermore, in a generic multiprocessor framework, there will be quite a lot of periodic tasks in the whole system, with their respective different periods, and this would generate a very huge table (its size is proportional to the least common multiple of the periods). Then SS unpractical as a global scheduler but it is still a good reference when measuring aperiodic response times.

In contrast, the *Dual Priority Algorithm* (DP) [14,15] does not need any pre-calculated table and therefore scales with the number of different periodic tasks better than Slack Stealing. On the other hand, the mean aperiodic response times obtained in previous uniprocessor works were worst than those obtained by the optimal Slack Stealing algorithm but quite close. In addition to its low memory usage, it has very low computational overheads, because it is based on an offline computation: the worst case response time possible for every periodic task due to higher priority periodic tasks interference. This value is used to promote a periodic task from a low priority level to a higher priority level band. 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, and in particular to the purpose of serving dynamic requests. Furthermore, other benefits may be obtained. The original uniprocessor DP 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. Within the multiprocessor domain it is possible to increase the number of schedulable task sets even more because conflictive periodic tasks may use other processors spare capacity. For an illustrative example see Figure 1.

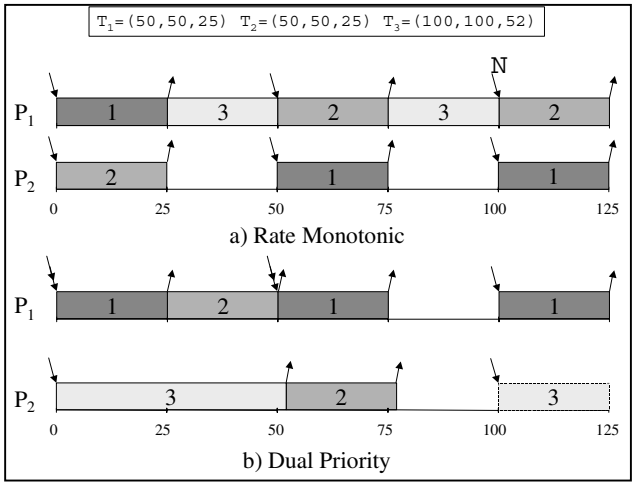

Figure 1: Dual Priority can increase the number of schedulable task sets. This figure represents the execution of the task set detailed in the legend using (a) RM global scheduling and (b) DP global scheduling using two processors. Downwards arrows mark task activation and upward arrows mark task termination. Although the task set only has a 152% total load it is not schedulable by a global scheduler with a fixed Rate Monotonic priority (see Figure 1(a), where T_3 misses a deadline at time t=100). Likewise this task set is not schedulable by an Earliest Deadline First global scheduler, illustrating that EDF is not optimal for multiprocessors [3]. In contrast, using a DP global scheduler, if a promotion time equal to zero is used for task T_3 then the whole system is schedulable (see Figure 1(b)).

Additionally, there are extensions of the DP model to deal with jitters releases [19], shared resources and arbitrary deadlines [20]. These extensions will be incorporated to the multiprocessor DP in a future work. Our goal in this paper is to adapt DP to multiprocessors systems and compare the aperiodic response times with SS results.

The rest of the paper is organized as follows. In Section 2 we set the framework and assumptions valid through this paper. In Section 3 we detail the adaptation of the Dual Priority algorithm to be used as a global

scheduler in shared memory multiprocessors. In Section 4 we study the results found in extensive simulations measuring the mean aperiodic response times and in Section 5 we summarise the conclusions of this work.

2. Framework and Assumptions

Consider a real-time multiprocessor system with N symmetrical processors and shared memory. Every processor *p* has allocated a set of *n* periodic tasks $TS_p=\{\tau_{p1}, ...,\tau_{pn}\}$. This allocation is performed at design time, for example using any of the techniques described in [17]. These tasks are independent and can be preempted at any time. Each task τ_{pi} has a worst-case execution time C_{pi}, a period T_{pi} and a deadline D_{pi}, assumed to satisfy $D_{pi} \leq T_{pi}$. Every instance k of a task must meet its absolute deadline, i.e. the k-th instance of the task τ_{pi} , say τ_{pi}^k, must be completed by time $D_{pi}^k= (k-1)T_{pi} + D_{pi}$. The *hyperperiod* of the task set is the least common multiple of all periods. We express all time measures (i.e. periods, deadlines, computations, etc.) as integral multiples of the processor clock tick. Each periodic task has a utilization factor defined as $U_{pi}= C_{pi}/T_{pi}$ and the maximum utilization factor in a processor *p* is $Umax_p= \max (U_{pi})$, $1\leq i \leq n$.

Each aperiodic task, J_k, has an associated arrival time α_k, a priori unknown, and a processing requirement C_{ap}^k (note that they do not have deadlines). These tasks are queued in FIFO order.

All periodic workloads considered in this paper have a total processor utilization (U_p) lower than the theoretical upper bounds (i.e. $U_p = \sum_{\tau pi \in TSp} U_{pi} \leq n(2^{1/n}-1)$). Hence any pre-runtime bin-packing algorithm can be used without problems to allocate the periodic tasks to processors. Recall that the remaining processor capacity is used to execute aperiodic tasks.

Finally, for the sake of simplicity, we assume all overheads for context swapping, task scheduling, task pre-emption and migration to be zero. In fact, some of these overheads may be taken into account in the pre-runtime worst-case execution analysis.

3. Multiprocessor Dual Priority (MPDP).

The Dual Priority (DP) scheduling algorithm has the following characteristics: (i) it guarantees periodic tasks deadlines, (ii) it achieves very good mean aperiodic response times, (iii) it has very low computation and memory requirements, (iv) it can use the full processor capacity, (v) it can use periodic tasks spare time (when they finish earlier than their worst case execution time) and (vi) it recovers quickly and in a controlled way from transient overloads. The first four characteristics are very important for the objectives of this paper. In particular, since in a multiprocessor system the number of tasks can be very high, resulting in a very long hyperperiod, which means excessive memory requirements for algorithms such as static Slack Stealing. 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 in 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 [21].

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

This recurrent formula is used pre-runtime to compute the worst-case response time of periodic tasks (the final W_i is task τ_i computation requirement and all higher priority tasks interference). Thus, the periodic task τ_i can be delayed until its promotion time = $D_i - W_i$. During runtime, a periodic task stars with a low priority level (lower than aperiodic tasks) and at promotion time its priority is updated to a higher level where it will only receive the interference of other higher priority promoted tasks, being its deadline guaranteed. However, this formula is not valid for multiprocessors.

The alternatives in multiprocessor to the scheduling test formula are two: to write an algorithm to find these values or to measure the release times through a simulation. The former alternative is also difficult and probably ineffective, because in a multiprocessor system there is an added uncertainty factor: the results depend on the decisions previously done about which task to execute where and these decisions make some tasks to interfere with others or not. For example, it is well known the multiprocessor anomaly of reducing some execution times can increase the run length [2]. As a consequence, an algorithm to compute the worst case response time for periodic tasks in a multiprocessor would consider the worst situations, giving very pessimistic values.

We have done some attempt to find the upper bounds to periodic task response times and the results obtained were far worse than those measured with simulations. We have refused this line because using these algorithms as a scheduling test result in rejecting perfectly schedulable task sets. Furthermore, when the test is passed, the computed values are so pessimistic that there is very low flexibility to execute aperiodic tasks.

The other alternative is to run at design time a simulation of the multiprocessor system with a pre-emptive global scheduler based on fixed priorities to find out the worst case response time for periodic tasks. This

method is used to determine the schedulability of task sets with global RM schedulers in [23]. If the measured release times are lower than their respective deadlines the system is schedulable and these values could be used in a *Dual Priority* global scheduler at run time. However, a new question raises: how long should the design-time simulation take?. In an uniprocessor system the simulation starts at the critical instant (all tasks are ready to be executed at the same time) and can be stopped when every periodic task first instantiation is finished. This does not hold for multiprocessors: the simulation must run all over the hyperperiod to find out the right values. The easy example depicted in Figure 2 shows this phenomenon.

In Figure 2, the first instantiation of task T_3 finishes at time t=8, with a response time of 8 time units, and the second instantiation finishes at time t=29, giving a response time of 9 units, i.e., a time unit worse than the previous instantiation. This is due to the fact that the second instantiation receives the interference of a task T_2 previous instantiation. A similar related phenomenon is detailed in [22] when using a global EDF scheduler.

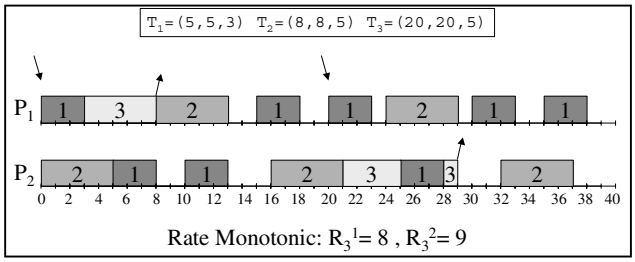

Figure 2: using a RM global scheduling the critical instant is not necessarily the initial time, when all tasks are ready to be executed at the same time. This figure represents the execution of the task set detailed in the legend using RM global scheduling with two processors. Downwards arrows mark T_3 activation and upward arrows mark T_3 termination.

The methodology described in the previous paragraph allows us to execute periodic and soft-aperiodic tasks with a global DP scheduler, but still has some drawbacks. The introduction of aperiodic tasks implies the introduction of indeterminism and produces periodic task shifts. As a consequence of these shifts periodic task execution patterns that have been not tested in the pre-runtime simulation may appear. Therefore, it would be possible to miss some deadlines. That means that via simulations it is not possible to find the worst response times. The task set in Figure 3 allows us to observe this phenomenon in a clear way. Figure 3-a plots a global RM execution of this example. The worst response time measured for task T_3 is 12 time units. Because its deadline is 21 the resulting promotion time is 21-12 = 9. Therefore, a priori this task can be delayed 9 time units.

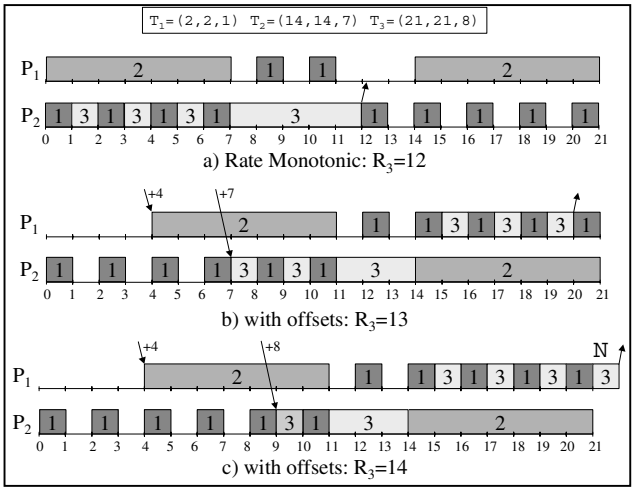

Figure 3: the worst case response time obtained via simulation with RM global scheduling can lead to miss some deadlines with a global Dual Priority scheduler

Figure 3-b depicts the execution of the same example but with an initial offset for the first instantiation of tasks T_2 and T_3. This could be equivalent to the execution delays due to aperiodic work. Now the response time of T_3 is 13, one unit more than previously. If this task was delayed one time tick plus (see Figure 3-c) it would miss its deadline. Note that a priory this task could be delayed two extra time units. This simple example illustrates the fact that the Dual Priority for multiprocessors is not so straightforward. The global RM execution does not try every offset. When executing aperiodic tasks, it is not sufficient to simulate a hyperperiod as a schedulability test as it is usually done in systems with pure periodic workloads (for example, see [5,23]).

In uniprocessor systems the *Dual Priority Algorithm* accomplishes the following premises:

1) No aperiodic task is executed while any periodic task is promoted.
2) Delaying the execution of a task to its promotion time is not a problem because this change in the offsets never will be worse than in the critical instant.
3) The computed worst case response time for any task includes the interference produced by all the higher priority tasks.

In multiprocessor systems these premises are not valid. The first is not valid because different processors can execute simultaneously aperiodic tasks and promoted periodic tasks. The second is not valid because the number of possible offsets is unbounded and then it is possible to have a situation worst than the critical instant. The third is not valid because when changing the periodic task offsets it is possible for some task to receive interference from higher priority periodic tasks not considered in the pre-runtime analysis. These anomalies

are similar to those described in previous works, for example [2,3]. Probably this could be avoided if it was forbidden to execute aperiodic tasks during the promotion time of any periodic task, but this would lead to long periods of time without being able to execute aperiodic tasks and the aperiodic tasks response times would be very bad.

Due to all these problems we propose to use the DP in a hybrid conformation between local and global scheduling. 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).

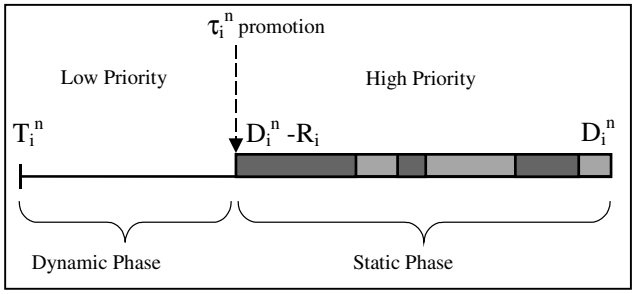

Figure 4: Periodic tasks allocation phases in MPDP

In our proposal, all periodic tasks are statically distributed among the processors, using any partitioning algorithm. Then the uniprocessor formula is used pre-runtime to compute the worst-case response time of periodic tasks locally. These values are used as an extra task parameter. During runtime, when a task arrives it is queued in a Global Ready Queue (GRQ). In this queue, aperiodic tasks have higher priority than periodic tasks and are queued in FIFO order. On the other hand, periodic tasks are sorted according their fixed low priority (for example RM priority order). The global scheduler (GS) selects the first N tasks from this queue to execute on the N processors. Additionally, there are N High Priority Local Ready Queues ($HPLRQ_i$ with i in [1..N]) used to queue promoted periodic tasks. When a periodic task τ_{pi} is promoted (i.e. it remains as much time as the pre-computed worst-case response time to its deadline) 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 imply a change in priority and can cause a pre-emption. Note that at promotion time, a periodic task must migrate to its original processor, where it will only receive the interference of other higher priority promoted tasks, being its deadline guaranteed.

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

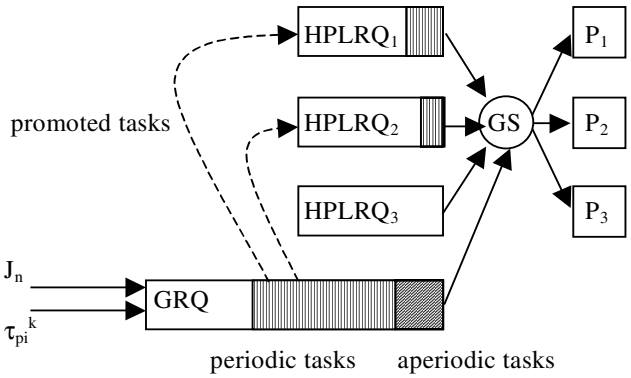

Figure 5: DP Global Scheduler. Squares represent processors, circles schedulers and rectangles queues. For any processor P_i , if $HPLRQ_i$ is not empty execute the first promoted task in P_i. Otherwise P_i execute the first task in GRQ.

periodic work and making the system ready for future aperiodic demands.

The time involved in the task migration and context-switching costs has been considered negligible, although it will be a subject of further study. At a first attempt, we have tried to minimize the impact of these costs by saving some migrations using a processor renaming strategy. Hence, when a periodic task is promoted and it is the only one promoted of its original processor, it has not to migrate back to it. In these situations a logical name swap between its current processor and its original processor is performed, saving the migration.

4. Results and Discussion

In this section we show the simulation results comparing the global scheduling detailed in the previous section (MPDP), versus the local scheduling Slack Stealing + global aperiodic scheduling using the next fit dynamic binding (SS+NF, [13]). We study the efficiency in performance comparing the mean aperiodic response time using synthetically generated task sets. As far as we know, no global scheduling has been studied to execute both periodic and soft-aperiodic tasks. In consequence we have compared our global scheduling proposal to the best local scheduling: the Slack Stealing algorithm [8,9].

We have varied the number of processors in the multiprocessor system from 2 to 8, which is quite common in shared memory real time systems.

The periodic task sets have been originally generated with balanced loads of 65% per processor. This periodic load is low enough to generate schedulable task sets ($65\% < n(2^{1/n}-1)$ [18], where n is the number of tasks) and high enough to experiment some difficulties in serving aperiodic requests. In the experiments we have increased proportionally the periodic load up to 72%, keeping fixed

the number of tasks and their corresponding periods. The periods of periodic tasks range from 100 to 3000 and the hyperperiod is 378000. This parameters lead to non-harmonic periods but with certain multiplicity degree (i.e., some periodic tasks may be ready to execute simultaneously, but not always the same tasks nor a great number of tasks). The number of periodic tasks in each processor has been fixed to 15. The weight load defined as (D_i/T_i) has been fixed to 1. In all the simulations every periodic task execute its worst-case computation time, although both algorithms are able to use eventual spare periodic time. All generated task sets have a breakdown utilization [24] greater than 75% and all the sets were schedulable (i.e., all periodic tasks computation requirements can be increased by a certain factor to raise the total load to 75% maintaining the task set schedulable). The aperiodic load is fixed to 25%, thus the total maximum load per processor analyzed is 97%. If a Sporadic Server with a period equal to the smallest period of the periodic task set is used, the maximum server size would be 35 time units (minimum period * (maximum breakdown utilization - periodic load) =100*(100%-70%)). In fact, a 100% breakdown utilization is not frequent. Its more realistic a 90% and this would give a maximum server size of 25 time units. Hence, we have used computation time requirements for aperiodic tasks in the range [1:25] time, achieving high demanding aperiodic workloads. We illustrate the trends reporting the results obtained for the extreme values.

We measured the mean response time of the aperiodic requests as a function of the periodic loads. The arrivals of the aperiodic requests were assumed to be Poisson distributed. For every point we represent in the figures there are involved 200 simulations, one hundred different task sets tested against both scheduling algorithms. The results depicted are obtained as the division of the mean aperiodic response obtained with SS+NF by those obtained with MDPD. For every task set and scheduling algorithm we repeated the simulation until we have reached a 95% confidence that the measured value was within a 5% interval around the true value. The only parameter we varied in these simulations was the initial seed for the aperiodic tasks arrival distribution generation.

For the first experiment, the synthetic periodic tasks have been generated to be small. The maximum utilization factor (Umax) has been fixed to 5%. Because the number of tasks (15) and the total load generated [65%, 72%], all tasks have a similar utilization factor. The results for this experiment are depicted in Figures 6 and Figure 7.

In Figure 6 we can see the results obtained for aperiodic request of one-tick computations. The first observation is that differences arise from total loads above 92% (67% periodic load +25% aperiodic load). The smaller the number of processors, the greater the differences. For example, with 2 processors and 70% periodic load, SS+NF is 75% worse than MPDP. For the same load, with 8 processors the disadvantage is reduced to 39%. This is because SS+NF has a dynamic part, the aperiodic task allocation, and it is easier to find a processor available for an aperiodic request when the number of processors is greater. When the number of processors is small, MPDP takes advantage of its dynamical aspect (i.e., the two priority levels and the periodic task migration capacity). In Figure 7 the periodic task sets are the same but the aperiodic requests are less frequent and have a computation request of 25 ticks. In this case, the MPDP advantage has been reduced. In the example points analyzed above (70% periodic load with two and eight processors) SS+NF is 17% and 3% worse than MPDP respectively. This is because both scheduling algorithms have more difficulties to allocate this amount of aperiodic computation request and, therefore, the results tend to equalize. MPDP is more flexible than SS+NF because of its capability to migrate periodic tasks and this allows MPDP scheduler to find small periods of time to serve aperiodic requests. When these requests are large enough both algorithms have the same problems, because the total available time to serve aperiodic tasks depends on the periodic load.

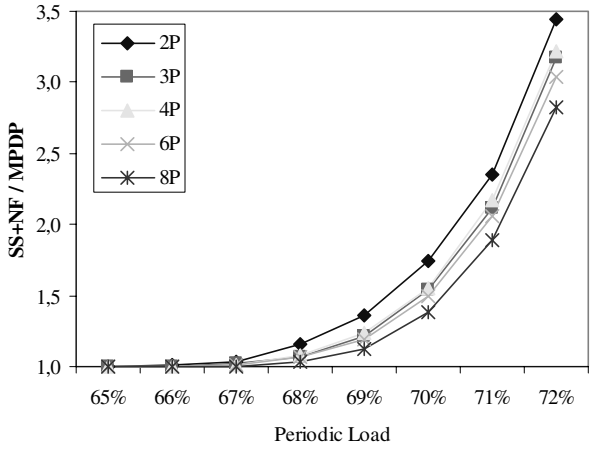

Figure 6: Umax= 5%, Cap= 1

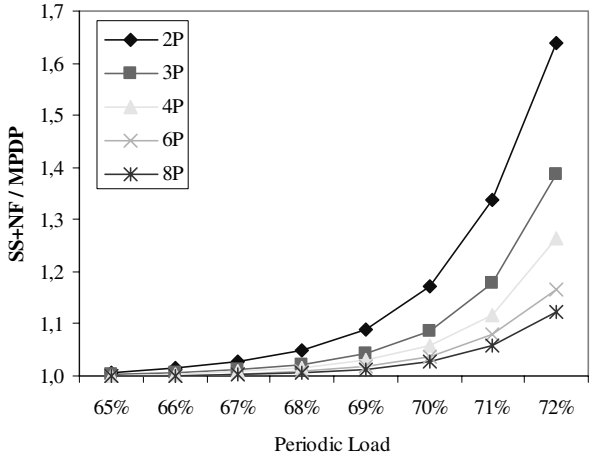

Figure 7: Umax= 5%, Cap= 25

For the second experiment the synthetic periodic tasks have been generated to have a heavy and infrequent task (large U_{pi} and long T_{pi}). The maximum utilization factor (Umax) has been fixed to 35%. The task with such a load is one of the three lower priority tasks. Because the number of tasks (15) and the total load generated [65%, 72%], the rest of tasks have a low utilization factor, which is about 2%. We have designed this experiment because with this kind of workloads the Dual Priority algorithm achieves worse mean aperiodic response times. Note that DP is based on a pessimistic value: the worst case response time of periodic tasks, which is achieved in the critical instant. When the heaviest task has the highest priority, the low priority part is a 65% of time and the high priority part is 35%. When the heaviest task has the lowest priority it receives the interference of all the other tasks, giving a high priority part of almost 100%. This means that the scheduler is not able to execute aperiodic tasks for long periods of time. In Figure 8 we can see the results obtained for aperiodic request of one-tick computations in the second experiment. Here the results are better for SS+NF, as we expected.

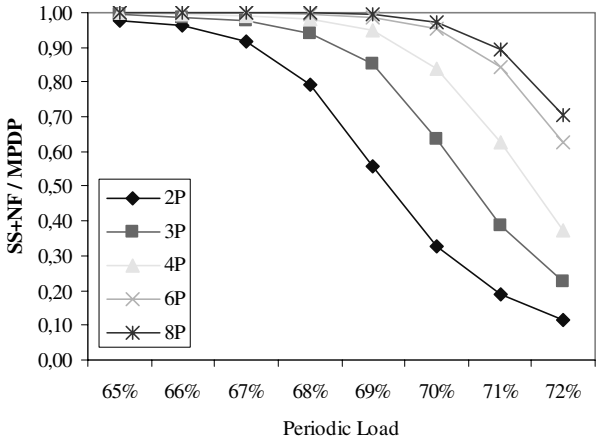

Figure 8: Umax= 35%, Cap= 1

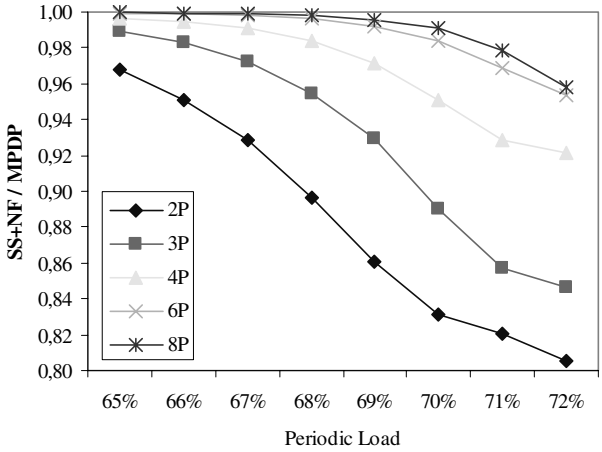

Figure 9: Umax= 35%, Cap= 25

The differences arise a little bit earlier than in the previous experiment, but still heavy total loads are needed to make some differences in the performance of both scheduling algorithms. Again, the smaller the number of processors, the greater the differences are. For example, with 2 processors and 70% periodic load, MPDP is 300% worse than SS+NF. This is because with two processors it is very likely to have them both in high priority level. For the same load, with 8 processors the MPDP disadvantage is reduced to 2,8%. This is due to the higher probability to find a processor running in low priority level, ready to serve an aperiodic request. When the aperiodic requests are of 25 ticks (see Figure 9) again the differences have been reduced substantially. For 70% of periodic load, with 2 and 8 processors, MPDP is 20% and 0,9% worse than SS+NF. For four processors or more, the differences are not statistically significant.

Finally, we have designed a third experiment, which is similar to the previous one, but the heavy task with the maximum utilization factor is not restricted to be one of the three lower priority tasks. It can be any of the tasks. In Figure 10 are depicted the results for this experiment when the aperiodic tasks have a computation request of 25 ticks. These results show that MPDP performs better than in the previous experiment. The performances are similar for both algorithms when the number of processors is high. However, MPDP is quite better than NF+SS when the number of processors is smaller. From this experiment we can conclude that MPDP only have some disadvantage in very special and identified cases which should be considered by the designer.

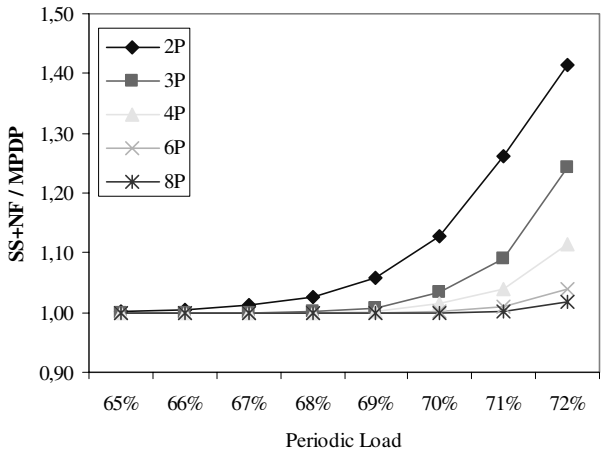

Figure 10: Umax= 35%, Cap= 25

Although traditionally the partitioning method has been used, recently the non-partitioning method is gaining interest in the research community [5,22,23]. Recent research has compared partitioning the periodic tasks among processors or not partitioning [5] with the performance metric being the number of schedulable task sets. They have concluded that with a small number of

processors it is better the non-partitioning method. We have found the same behavior when the performance metric is the aperiodic response time. Hence, the global scheduling in shared memory multiprocessors with small number of processors could be a right choice for the near future real-time operating systems.

5. Conclusions

In this paper we have detailed how to use the Dual Priority algorithm as a global scheduler in a multiprocessor system and the problems found to adapt it to these platforms. The solution proposed is an hybrid model with to phases for every task: a dynamic phase where periodic tasks can execute on any processor and a static phase, when the periodic task has to execute on a particular processor to meet its deadline. With this scheme, all periodic task deadlines remain guaranteed. The periodic processor utilization upper bounds are the same than in uniprocessor systems, but the remaining processing capacity is available to aperiodic load. With extensive simulations we have shown that this method achieves very good mean aperiodic response time. Furthermore, when the system is heavily loaded it can achieve better performance than an optimal local scheduler as the Slack Stealing with aperiodic tasks migration implemented. This performance gain is greater when the number of processors is small.

We have also identified the characteristics of particular periodic task sets that perform badly with Dual Priority. We have show that this effect is less important when the number of processor increases. Nevertheless, a further research could be done to cope with these situations.

6. References

[1] Burns, A, "Scheduling Hard Real-Time Systems: a Review" Software Engineering Journal, 6 (3), pp. 116-128, 1991

[2] Stankovic, J.A., Spuri, M., Di Natale, M., Butazzo, G.C., "Implications of Classical Scheduling Results for Real-Time Systems", IEEE Computer, v. 28, n.6, pp.15-25, 1995

[3] 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

[4] Garey M.R., Johnson D. S., "Complexity Results for Multiprocessor Scheduling under Resource Constraints". SIAM Journal on Computing, 4(4): 397-411, 1975

[5] 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

[6] Strosnider J.K., Lehoczky J.P., Sha L., "The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments", IEEE Transactions on Computers, v. 44, n. 1, pp 73-91, 1995

[7] Sprunt B., Sha L, Lehoczky J.P., "Aperiodic Task Scheduling for Hard Real-Time Systems", Real-Time Systems Journal, vol. 1, pp. 27-60, 1989

[8] Lehoczky J.P., Ramos-Thuel S., "An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed Priority Preemptive Systems". RealTime Systems Symposium 1992

[9] Lehoczky J.P., Ramos-Thuel S., "Chapter 8: Scheduling Periodic and Aperiodic Tasks using the Slack Stealing Algorithm", pp. 175-197, Principles of Real-Time Systems, Prentice Hall, 1994

[10] Sáez S., Vila J., Crespo A., "Soft Aperiodic Task Scheduling on Real-Time Multiprocessor Systems", Sixth International Conference on Real-Time Computing Systems and Applications, pp. 424-427, 1999

[11] Ramamritham K., Stankovic J.A., Zhao W., "Distributed Scheduling of Tasks with Deadlines and Resource Requirements", IEEE Transactions on Computers, C-38, (8), pp. 1110-1123, 1989

[12] Fohler G., "Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems", Real-Time Systems Symposium, 152-161, 1995

[13] Banús, J.M., Arenas, A., Labarta, J., "An Efficient Scheme to Allocate Soft-Aperiodic Tasks in Multiprocessor Hard Real-Time Systems", Parallel and Distributed Processing Techniques and Applications, v.II, pp.809-815, 2002

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

[15] Moncusí, M.A., Banús, J.M, Labarta, J., Llamosí, A., "The Last Call Scheduling Algorithm for Periodic and Soft Aperiodic Tasks in Real-Time Systems", V Jornadas de Concurrencia, 1997

[16] Burns A. Wellings, A., "Real Time Systems and Programming Languages, 2n Ed.", Addison Wesley, 1997

[17] Burchard A., Liebeherr J., Yingfeng Oh, Sang H. Son, "New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems", IEEE Transactions on Computers, vol. 44, no. 12, December 1995

[18] 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

[19] Audsley, N., Burns, A. Richardson, M., Tindell, K., Welling, A.J., "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling", Software Engineering Journal, 8(5), pp. 284-292, 1993

[20] Tindell , K., Burns, A., Welling, A.J., "An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks", Real-Time Systems, 6(2), pp. 133-151, 1994

[21] 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

[22] Gossens, J., Funk, S., Baruah, S., "EDF scheduling on Multiprocessor platforms: some (perhaps) counterintuitive observations", Real-Time Computing Systems and Applications, pp. 321-330, 2002

[23] Anderson, B., Baruah, S., Jonsson, J., "Static-Priority Scheduling on Multiprocessors, Real-Time Systems Symposium, 2001

[24] Lehoczky, J.P., Sha, L., Ding, Y., "The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior", Proceedings of Real-Time Systems Symposium, pp. 166-171, 1989