An Efficient Scheme to Allocate Soft-Aperiodic Tasks in Multiprocessor Hard Real-Time Systems

Josep M. Banús, Alex Arenas, 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

We propose a scheme to allocate individual soft-aperiodic tasks in a shared memory symmetrical multiprocessor for hard-real time systems applications. The aim of this scheme is to minimize the aperiodic mean response time by running a local uniprocessor scheduler to serve periodic tasks guaranteeing their deadlines, and serving aperiodic tasks with a global scheduler that uses different allocation strategies. We show that performance achieved depends highly on the allocation decisions made by the global scheduler when several processors are available to execute aperiodic tasks. We compare four classical allocation strategies and the best results are obtained using a next-fit allocation strategy. The mean aperiodic response time with this strategy is less than 2 times the computation time required by the aperiodic tasks, up to total loads of 97%.

1. Introduction

Multiprocessor systems have evolved spectacularly in the last years. These systems are available from personal computers to large-scale computational machines. In particular, 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, the algorithms that are optimal for uniprocessor systems are not optimal when the number of processors is increased [3] (it is well known that optimal scheduling for multiprocessors systems is NP-Hard [4]). Nevertheless, as a first approach it is usual to allocate periodic processes to processors and, after that, to use an optimal uniprocessor scheme on each processor individually [1].

The common framework to deal with in real-time systems usually involves periodic, and/or aperiodic, and/or sporadic tasks. In this situation the schedulability of these systems has been investigated following two main approaches: i) to treat every processor as a single uniprocessor executing all kind of local tasks or ii) to consider only one kind of tasks to be executed at any processor i.e. periodic or sporadic. Using the first approach all well-known techniques for uniprocessors are used, e.g. [5-9]. This implies a large number of advantages, in particular avoiding the complexity of multiprocessor systems scheduling, but the fact of using a multiprocessor is not exploited: load balancing, redundancy, fault tolerance, etc. are not considered. The second approach is suitable to implement only those systems composed by one kind of tasks. This may be originated from the initial problem formulation or by a transformation of different types of tasks into a single one). Transforming aperiodic tasks or sporadic tasks into periodic tasks may under-use the capacity of the processors, because the minimum inter-arrival time is used as the new task period. Previous works that only deal with periodic tasks consider the problem of static allocation (e.g. [10-14]) or dynamic allocation (e.g. [15]). The performance measure used is the number of processors needed or the number of task sets found schedulable. On the other hand, modeling all tasks as aperiodic tasks ([16-20]) or sporadic tasks ([21,22]) gets rid of important a-priori knowledge of the tasks, the periodicity of some tasks, transforming the application in a completely dynamic system. With these on-line scheduling policies only instantaneous feasibility is taken into account and nothing can be ensured for the near future. Consequently, these approaches are truly useful for real dynamic systems, not for many critical hard real-time systems.

In the past, joint scheduling real-time periodic, sporadic and soft-aperiodic (from now on we will use the term aperiodic task referring to soft-aperiodic tasks) task sets has been extensively studied for uniprocessor systems, for example the Deferrable Server [8], the Sporadic Server [23] and the Slack Stealing Algorithm [6,7] (that offers an optimal scheduling strategy) solve this problem. Unfortunately, the applicability of these scheduling algorithms to the joint multiprocessor system is not straightforward.

However, it is possible to find a feasible solution: to execute periodic tasks fixed in every processor using a well known uniprocessor scheduling algorithm while allowing aperiodic tasks to migrate to any processor [24,25]. 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 is so more for unbalanced loads, but it is also useful for balanced loads: 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 the aperiodic tasks. The differences would arise from the allocation strategy of these aperiodic tasks.

In this paper we want to exploit the benefits of the static allocation of periodic tasks (guaranteeing their deadlines), and the benefits of using the spare capacity or spare laxity to execute aperiodic tasks in any processor on a symmetric multiprocessor system. With this goal in mind, we propose a scheme to allocate aperiodic tasks in multiprocessor hard-real-time systems. Similar approaches can be found in the context of distributed systems [25,26], but allocating groups of aperiodic tasks and using message passing. These approaches allocate hard aperiodic tasks to nodes where its deadlines will be meet, but they do not specify what to do whenever there are several nodes that can serve the aperiodic tasks. We will focus our study in shared memory multiprocessor systems, where it is not necessary to send any message to allocate tasks because a global scheduler can use the information located in the shared memory of the current state of the local schedulers. Furthermore, tasks migration is direct and less expensive, because tasks codes and data do not move from the shared memory. Although these multiprocessor characteristics simplify periodic tasks scheduling at the same time makes a clear difference with distributed systems, the problem of finding the best processor to execute aperiodic tasks is not trivial, because the scheduling decisions on-line can modify substantially the behavior of future task executions. Here we study whether aperiodic task allocations influence their response times.

The paper is structured as follows: in the next section we present the framework of the system we deal with. After that, in section 3 we propose the scheme to allocate the aperiodic tasks to processors and we analyze several strategies of choosing the processor to execute. In section 4 we show the results of simulation studies to compare different allocations. Finally, in section 5 we present the conclusions.

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 periodic tasks, $TS_p=\{\tau_{p1}, ..., \tau_{pn}\}$. This allocation is performed in some way at design time, for example using any of the techniques described in [11], but this will not be a subject of discussion in the present paper. These tasks are independent and can be preempted at any time. Each task τ_{pi} has a worst-case computation requirement C_{pi}, a period T_{pi}, 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 $D_{pi}^{k}= (k-1)T_{pi} + D_{pi}$. We express all time measures (i.e. periods, deadlines, computations, etc.) as integral multiples of the processor clock tick.

Every processor p will execute these tasks using a local Slack Stealing scheduler (LS_p) [6,7].

The scheduler needs every periodic task to have a fixed priority, P_{pi}, usually determined using Deadline Monotonic [5]. This scheduler guarantees that every instance of every task will meet its deadline. Every local scheduler determines the slack available for aperiodic processing by computing

$$S_{pk}(t) = \min_{\{0 \leq i \leq n\}}(A_{pi}(\gamma_{pi}(t)+1) - \mathcal{A}_{pi}(t) - \mathcal{I}_{pi}(t))$$

where $S_{pk}(t)$ is the available slack for aperiodic processing in processor p with priority k at time t, $\gamma_{pi}(t)$ is the number of instances of τ_{pi} completed by time t, $\mathcal{I}_{pi}(t)$ is the level i inactivity during [0,t], $\mathcal{A}_{pi}(t)$ is the cumulative aperiodic processing consumed during [0,t] at level i or higher and $A_{pi}(\gamma_{pi}(t)+1)$ is the largest amount of time available for aperiodic processing between 0 and the competition of the next τ_{pi} periodic task. The table A_{pi} is computed pre-runtime using the algorithm originally specified in [6] and redefined in [7].

We assume that there is enough memory to store all these tables. Also, we assume that the hyperperiod of every TS_p, i.e. the least common multiple of all task periods in processor p, is short enough to keep A_{pi} a reasonable size.

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 treated in a FIFO order by a global scheduler (GS) that allocates them to processors using the state information of the local schedulers. We use the recommendations from SSA authors [6], using the highest priority level (k=0) for aperiodic tasks, i.e. using $S_{p0}(t)$.

Finally, for the sake of simplicity, we assume all overheads for context swapping, task scheduling, task pre-emption and aperiodic tasks migration to be zero.

3. Dynamic Allocation and Scheduling scheme

In this section we propose a global scheme to deal with the problem of dynamically allocate aperiodic task and schedule the whole multiprocessor system of periodic and aperiodic tasks.

We use the Slack Stealing Algorithm (SSA) [7] to locally schedule the periodic tasks to every processor. This algorithm has been proven to be optimal in the sense of minimizing the response time of every aperiodic task among all scheduling algorithms that use the same priority assignment. To allocate aperiodic tasks we use a global aperiodic tasks scheduler and a global ready queue data structure. This global scheduler has knowledge of the state of all the local schedulers and distributes aperiodic tasks among processors accordingly to their available slack. The SSA can reclaim unused periodic execution time and can service hard deadline aperiodic tasks but can not cope with release jitters nor shared resources. These are not drawbacks in the scope of this study because we want to show that different aperiodic tasks allocation schemes have different performances even for the best scheduling algorithm

In our proposal, when an aperiodic task is requested to be executed it is queued in the Global Aperiodic Ready Queue (GARQ). The Global Scheduler (GS) tries to allocate every aperiodic task J_k among processors, see Figure 1. As a first approach the GS allocates every aperiodic task statically among processors, the GS selects for the allocation the processor p with more slack available. Then, the GS queues the aperiodic task to the processor Local Ready Queue (LRQ_p). If there is no processor with slack, the aperiodic task remains queued in GARQ until some slack is available at any processor. This may happen when a periodic task is completed.

If the GS is allowed to allocate various tasks simultaneously to the same processor, then it would be necessary to keep track of the slack already assigned in each processor in an array, lets say V_p. Hence, the available slack for processor p would be computed as follows:

$$S_{pk}(t) = \min_{\{0 \le i \le n\}}(A_{pi}(\gamma_{pi}(t)+1) - \boldsymbol{A}_{pi}(t) - I_{pi}(t))) - V_p(t)$$

where $V_p(t)$ is the slack already assigned in processor p by time t.

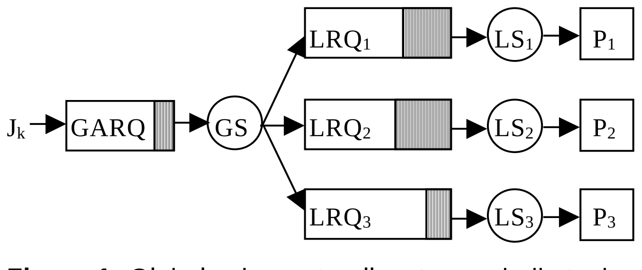

Figure 1: Global scheme to allocate aperiodic tasks. Squares represent processors, circles schedulers and rectangles queues.

Allocating simultaneously various aperiodic tasks to a processor is inefficient because it can cause the LRQ to be very long. Suppose that at time t a processor P_1 has quite a lot of slack and a second processor P_2 has not, then, during a period of time all aperiodic task arrivals are queued in LRQ_1 and, after a short time, some slack is available in P_2 but no more aperiodic tasks arrive. This slack in P_2 can not be used. Furthermore, processor P_1 may run out of slack for a long period of time, because its periodic tasks are delayed to the maximum. This would increase the mean aperiodic response time unnecessarily. Note that this happens even in the case P_2 has some slack, but less than P_1 ($S_2(t) << S_1(t)$). Hence, the GS should be designed to allocate only one aperiodic task to a processor at a time.

On the other hand, every local scheduler (LS_p) dispatches aperiodic tasks only if there is slack available. If there is not enough slack to complete the aperiodic task, a signal must be programmed to queue again the aperiodic task in the LRQ_p. Such a task will wait for new slack to become available in the same processor while probably another processor may be idle or may have enough slack to execute it. Furthermore, another processor may be processing more recent aperiodic tasks. Hence, is seems reasonable to return aperiodic tasks running out of assigned slack to the GARQ. To maintain the FIFO order in the GARQ it is needed to mark the aperiodic tasks with a time stamp and sort the GARQ with this criterion.

Keeping in mind all these observations, we conclude that if aperiodic tasks have to be waiting in a queue it is better to keep them queued in the GARQ to minimize as much as possible their mean aperiodic response time. Our Global Scheduler algorithm has the following steps:

```
1- suspend until
  -1a     an aperiodic task new arrival, $J_k$
  -1b     or an aperiodic task executing on a processor
           runs out of assigned slack, $J_k$
  -1c     or new slack is available on a processor that had
          none
2- if (1a or 1b) then queue $J_k$ to GARQ
3- pick the first aperiodic task in GARQ, $J_k$
4- select a processor with available slack, p
5- if it exist such a processor p, queue $J_k$ to $LRQ_p$
6- else queue $J_k$ to GARQ
```

Figure 2: Global Scheduler Algorithm

To implement this algorithm, it is necessary to determine the selection function (step 4) that returns which processor, among all with available slack, to select.

Let us suppose J_k is the aperiodic task to allocate with a processing requirement C_{ap}^k and that there are two processors, P_1 and P_2, with slacks $S_1(t)$ and $S_2(t)$

respectively. If $S_1(t) < C_{ap}^k < S_2(t)$ then necessarily P_2 must be chosen because is the only processor that can execute J_k completely, on the other hand, if $S_1(t) < S_2(t) < C_{ap}^k$, a priori the best option is still to choose P_2 because it can execute a larger portion of J_k than P_1. Otherwise, when each processor has enough slack to complete J_k it is not a priori clear which is the best option. The current processor with more slack is not necessarily the best choice, it could be sufficient to choose a processor with enough slack available, and several strategies can be proposed. In the present work we compare the following allocation strategies:

First-Fit-Allocation (FFA): To choose the first processor found with enough slack.
Next-Fit-Allocation (NFA): To choose the next processor found with enough slack, starting from the next processor in a predefined order.
Best-Fit-Allocation (BFA): To choose the processor with less but enough slack found.
Worst-Fit-Allocation (WFA): To choose the processor with more slack found.

In all these algorithms, when failing to find a processor with enough slack, the processor with more slack available is chosen. Note that the first two strategies stop searching when they find a processor with enough slack while the last two require to search over all the processors**.** When the aperiodic processing requirements are similar or longer than the usual available slacks, every allocation scheme behaves similarly because all them have to check every processor and probably they will choose the same processor, otherwise, when more slack is available significant differences can develop.

4. Results and Discussion

In this section we show the simulation results for the four different allocation functions in the scope of the global scheme proposed in Figure 1. We study the efficiency in performance comparing the mean aperiodic response time using synthetically generated task sets. We have fixed the number of processors in the multiprocessor system to 4, after the evaluation of the performance with up to 8 processors. Four processors have shown to be enough to evidence performance differences and allow us to simulate a large number of experiments in a reasonable time. On the other hand, a larger number of processors is not common in shared memory real time systems.

Before comparing the differences in efficiency between the four allocation algorithms we will analyze an example that demonstrate that considerable performance differences may occur. In Figure 3 we represent the available slack evolution for each processor using FFA and in Figure 4 we represents the same example using a NFA strategy. In both figures, processor P_1 evolution is represented at the top and processor P_4 at the bottom (note that every processor corresponding graphic has a different available slack scale depending on the configuration).

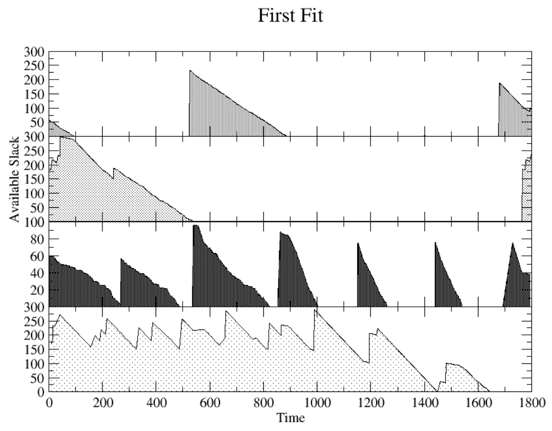

Figure 3: slack evolution example using FFA

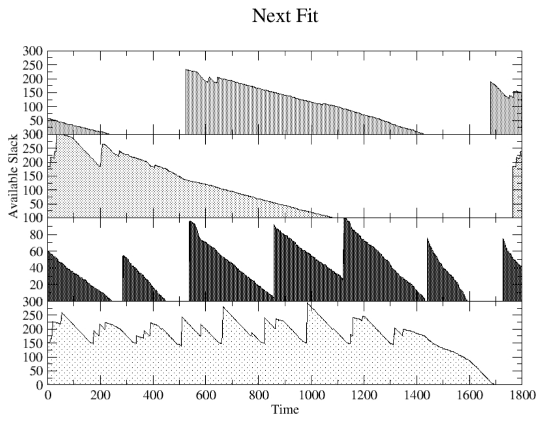

Figure 4: slack evolution example using NFA

The base width of the peaks, that represents the time the slack is available, is wider for the NFA for all processors than for the FFA. This means that the NFA is distributing in a better way the aperiodic job and hence the efficiency in front of the FFA is improved. Processor P_4 has a lot of available slack and it is periodically renovated, FFA only uses this capacity from time to time, exhausting other processors capacities, while NFA takes advantage of the periodic capacity of P_4, enlarging others processors available slack. Curiously, with NFA, at the beginning P_4 has slightly less slack than with FFA but later the slack time availability is longer. Finally, we observe that with FFA about the 30% of the time at most one processor has available slack, while with the NFA this situation only happens about the 5% of the time. The consequence should be a shorter mean response time for aperiodic tasks using the NFA scheme. To corroborate this insight we have performed a sequence of experiments in a wide variety of situations. We have generated synthetic task sets to check the performance of the multiprocessor system.

The periodic task sets have been originally generated with balanced loads of 70% per processor. This periodic load is low enough to generate schedulable task sets and high enough to experiment some difficulties in serving aperiodic requests. The periodic load has been sequentially increased in the experiments up to 73% by increasing proportionally all worst-case execution times of all periodic tasks. The number of periodic tasks in each processor has been fixed to 15 and the maximum utilization factor defined as (C_i/T_i) has been fixed to 20%. 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 the SSA is able to use eventual spare periodic time. All generated task sets have a breakdown utilization [27] greater than 75% and all the sets were schedulable. Unless explicitly stated, the aperiodic load is fixed to 25% thus, the total maximum load per processor analyzed is 98%. 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 30 time units (minimum period * (maximum breakdown utilization - periodic load) =100*(100%-70%)). Therefore, we have used computation time requirements for aperiodic tasks in the range [1:25], achieving high demanding aperiodic workloads.

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. Every point we represent in the figures is obtained as the average mean aperiodic response times of 100 different task sets. For every task set 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.

In the experiments, we have used periodic non-harmonic task sets and harmonic task sets. Harmonic tasks sets are representative of some real time systems, they usually give higher processor utilization [5] and they have been commonly used in the literature benchmarks [28]. We have also compared task sets with different period ratio (i.e. the ratio between the largest period and the shortest period) from task sets within 100:1000 period ratio to 100:3000 period ratio. Finally, a comparison between task sets with high breakdown utilization [27] and lower breakdown utilization has been performed.

The first experiment has been designed to locate the load range where the differences between the allocation strategies are more relevant. In Figure 5 the aperiodic load has been fixed to approximately 15%, and the total load raises up to 98%. As can be seen, the differences between allocation strategies arise when the system is heavily loaded, mainly in the last 5 points. We have obtained similar results with higher aperiodic loads (i.e. 33%). For this reason, we have designed all the following experiments to observe these differences when the total load varies from 95% to 98%.

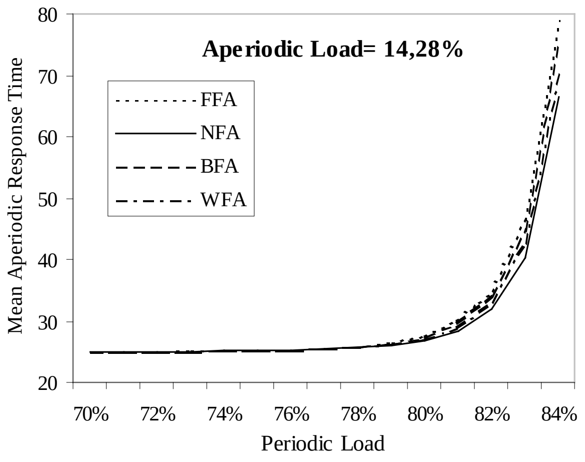

Figure 5: T=100:1000, C_{ap}=25

The results of the second experiment are depicted in Figure 6. The periods of periodic tasks range from 100 to 1000 and the hyperperiod is 378000. This parameters lead to non-harmonic periods. In Figure 6 we plot the mean aperiodic response time versus the periodic load ranging from 70% to 73% when the aperiodic load is 25 %. The calculation time of aperiodic tasks has been fixed to 1 time tick, i.e. we have a lot of small aperiodic tasks. The results show that the best allocation scheme is in this case the NFA, closely followed by the WFA while the worst are BFA and FFA in this order. The difference of performance between NFA and FFA is approximately 63% in the most difficult situation when there is a 98% of total load in every processor. The variance of the reported mean average response time fluctuates within 20%-25% in the different points.

The third experiment show the behavior when the periods have a wider range than in the second experiment, the new range periods is from 100 to 3000. The results are represented in Figure 7 and Figure 8. The results are similar than those in the second experiment but the mean aperiodic response time increases as the maximum period, i.e. nearly three times the values of the second experiment. It is remarkable that the difference between NFA and FFA represents an improvement of the 80% in the mean aperiodic response time, even with the closest strategy to the NFA, the WFA, the difference is approximately of the 40%. Furthermore, if we compare this figure with Figure 6 we detect that the more robust strategy to the extension on the period ranges is the NFA.

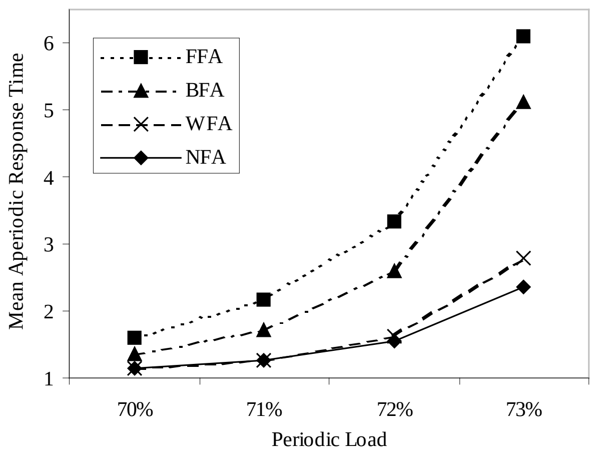

Figure 6: T=100:1000, C_{ap}=1

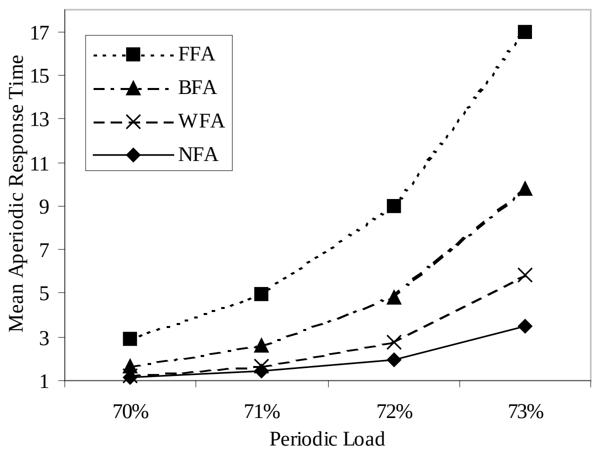

Figure 7: T=100:3000, C_{ap}=1

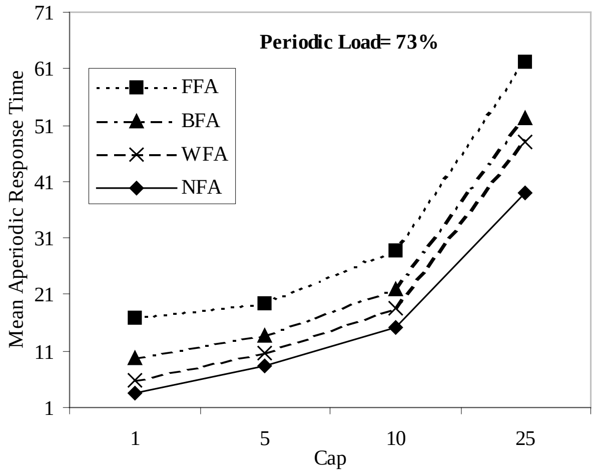

Figure 8: T=100:3000

In Figure 8 we analyze the specific case of maximum load (73% of periodic load) when the calculation time of the aperiodic tasks increases from 1 to 25 time ticks, i.e. the work of the aperiodic tasks is enlarged. In this situation the differences between the four allocation strategies are maintained. Although the mean aperiodic response time increases with the total load, the relative waiting times decrease to a 10%-20%. In fact, for an aperiodic requirement greater than or equal to 5 time ticks the relative waiting times converge in a narrow scale for the four strategies. This means that the waiting times are similar for the four strategies as the aperiodic demands increase.

In the fourth experiment, we want to study the effect of the multiplicity of periods in the performance of the system. The hyperperiod is 400000 and the tasks periods, in the range [128:3125], are more harmonic (multiples of powers of 2 and 5) than in the third experiment. Although this difference is important, still we have comparable period ranges and hyperperiod between experiments third and fourth. The results obtained are similar to those achieved in previous experiments, but the mean aperiodic response times are better than in the third experiment (they range from 1 to 13 when C_{ap}=1). We have compared the results obtained in the last two experiments versus the second experiment, which uses a shorter range of periods. We have observed that the relative difference between WFA and NFA is getting larger (WFA is becoming less efficient) while the relative difference between BFA and NFA is getting shorter (BFA is becoming more efficient).

Finally, in the fifth experiment we investigate the effect of the breakdown utilization per processor in the difference between the four strategies. The parameters of this experiment are the same than those used in experiment one, but we have selected only task sets with the same breakdown utilization per processor. This constraint reduces the possible number of experiments drastically, e.g. we have generated 10 task sets for every case in the experiment. Two cases have been simulated, all processors having a high breakdown (95%) and all processors having a lower breakdown (85%).

When the breakdown utilization is 85% per processor, the mean aperiodic response time is a 30% worst for the lowest periodic load than in Figure 6 and grows up as the periodic load increases. When the breakdown utilization is higher, 95% per processor, the mean aperiodic response time improves with respect to previous case. This result matches with the experiences with uniprocessor systems. In this scenario, the FFA allocation scheme is not capable of taking advantage of the higher breakdown utilization, on the contrary the NFA obtain dramatic performance gains, reacting to aperiodic demands almost immediately even for a system close to 100% load.

Conclusions

We have presented a general scheme to allocate aperiodic tasks that uses a global scheduler for the whole multiprocessor system and local schedulers for every individual processor interacting via queues in shared memory. The scheme efficiency is mainly determined by the allocation strategy adopted. We have studied four different allocation strategies FFA, NFA, BFA and WFA.

We have shown under a wide range of circumstances that the Next Fit Allocation scheme is the best option to minimize the mean aperiodic response time. In general, this mean response time is lower than twice the aperiodic processing demand, which can be suitable for a large number of aperiodic applications. These results contrast with the common approach in distributed systems where the allocation strategy mostly used is to choose the processor with the highest surplus [25].

5. 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] 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.

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

[7] 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

[8] 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

[9] Spuri, M., Butazzo, G.C., "Scheduling Aperiodic Tasks in Dynamic Priority Systems", Real-Time Systems Journal, vol. 10, pp. 179-210, 1996

[10] Ramamritham, K., "Allocation and scheduling of precedence-related periodic tasks", IEEE Trans. on Parallel and Distributed Systems, v. 6, n. 4, pp. 412-420,1995

[11] 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

[12] Khemka A., Shyamasundar R. K., "An Optimal Multiprocessor Real-Time Scheduling Algorithm", Journal of Parallel and Distributed Computing 43, 37-45, 1997

[13] Sáez S., Vila J., Crespo A., "Using Exact Feasibility Tests for Allocating Real-Time Tasks in Multiprocessor Systems. Euromicro Workshop on Real-Time Systems, 1998

[14] Baruah S., "Scheduling Periodic Tasks on Uniform Multiprocessors", Information Processing Letters 80, 97-104, 2001

[15] Davari S., Dhall S.K., "An On Line Algorithm for Real-Time Tasks Allocation", Real-Time Systems Symposium, 1986, 194-200.

[16] Ramamritham K., Stankovic J. A., Shiah P-F., "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems", IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 2, pp. 184-194, April 1990.

[17] Wang F., Ramamritham K., Stankovic J. A., "Bounds on the Performance of Heuristic Algorithms for Multiprocessor Scheduling of Hard Real-Time Tasks", Real-Time Systems Symposium, pp. 136-145, 1992

[18] Koren G., Shasha D., Huang S.C., "MOCA: Multiprocessor On-Line Competitive Algorithm for Real-Time System Scheduling". Real-Time Systems Symposium, pp. 172-181, 1993

[19] Hamidzadeh B., Atif Y., "Dynamic Scheduling of Real-Time Aperiodic Tasks on Multiprocessor Architectures", Proceedings of the 29th Annual Hawai International Conference on System Sciences, pp. 469-478, 1996

[20] Manimaran G., Siva Ram Murthy C., "An Efficient Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems", IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 3, pp. 312-319, 1998.

[21] Hongyi Zhou, Schwan K., Akyildiz I.F., "Performance Effects of Information Sharing in a Distributed Multiprocessor Real-Time Scheduler", Real-Time Systems Symposium, pp. 46-55, 1992

[22] Dominic M., Bijendra N.J., "Conditions for On-Line Schedulig of Hard Real-Time Tasks on Multiprocessors", J. of Parallel and Distributed Computing, 55,121-137, 1998

[23] 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

[24] 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

[25] 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

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

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

[28] Kamenoff N.I., Weiderman, N.H., "Hartstone Distributed Benchmark: Requirements and Definitions", Real-Time Systems Symposium, pp. 199-208, 1991