Comparative evaluation of limited preemptive methods

Gang Yao, Giorgio Buttazzo and Marko Bertogna

Scuola Superiore Sant'Anna, Pisa, Italy, {g.yao, g.buttazzo, m.bertogna}@sssup.it

Abstract

Schedulability analysis of real-time systems requires the knowledge of the worst-case execution time (WCET) of each computational activity. A precise estimation of such a task parameter is quite difficult to achieve, because execution times depend on many factors, including the task structure, the system architecture details, operating system features and so on.

While some of these features are not under our control, selecting a proper scheduling algorithm can reduce the runtime overhead and make the WCETs smaller and more predictable. In particular, since task execution times can be significantly affected by preemptions, a number of scheduling methods have been proposed in the real-time literature to limit preemption during task execution. In this paper, we provide a comprehensive overview of the possible scheduling approaches that can be used to contain preemptions and present a comparative study aimed at evaluating their impact on task execution times.

I. Introduction

Preemption is a key factor in real-time scheduling algorithms, since it allows the operating system to immediately allocate the processor to incoming tasks that have higher priority to complete. Under dynamic priority scheduling like Earliest Deadline First (EDF), higher priority task corresponds to higher urgency due to its earlier deadline, while under fixed priority, a higher priority task may preempt a more urgent but lower priority task. In fully preemptive systems, the running task can be interrupted at any time by another task with higher priority, and be resumed to continue later. In other systems, preemption may be disabled for certain intervals of time during the execution of critical operations (e.g., interrupt service routines, critical sections, etc.). There are also some situations in which preemption is completely forbidden to avoid unpredictable interference among tasks and consequently, a higher degree of predictability is achieved.

This work has been partially supported by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement no. 216008.

Non-preemptive scheduling can achieve a more predictable execution behavior since the task instance will continue until completion once gets started, however, non-preemptive regions of code introduce additional blocking delays that may degrade the effective system utilization. Indeed, when the preemption cost is ignored in the analysis, as the assumption commonly adopted in many research works, fully preemptive scheduling is more efficient compared to the non-preemptive case in terms of processor utilization. In practice, however, arbitrary preemptions cause significant runtime overhead and high fluctuations in task execution times.

Whenever a preemption takes place, different sources of overhead must be taken into account. First of all, the current task is suspended and inserted in the ready queue, then the scheduler makes a context switch and the new task is dispatched. The time needed for these operations is referred to as *context switch cost* and is denoted by σ. This cost can easily be taken into account in the schedulability analysis by increasing the WCET of the preempting task by an amount equal to σ.

When the preempted task resumes its execution, there are other indirect costs to be considered, related to cache misses, pipeline refills, bus contentions and so on. In fact, an extra overhead is necessary to refill the pipeline of the pre-fetch mechanism, since preemption typically destroys program locality of memory references. An additional overhead is charged to the resumed task to reload the cache lines evicted by the preempting task. Such a reloading also generates additional accesses to the RAM, therefore possibly increasing the number of bus conflicts caused by other peripherals. The cumulative execution overhead due to the combination of these effects is referred to as *architecture related cost* and is denoted by γ. Unfortunately, this cost is with high variance and depends on the specific point in the task code when preemption takes place [1], [10], [15]. Figure 1 illustrates a simple schedule result where the different sources of overhead explained above are highlighted.

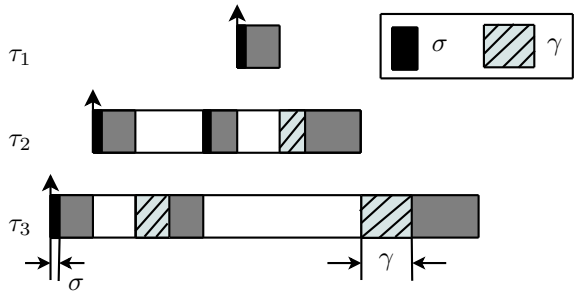


Fig. 1. A sample preemptive schedule with different sources of overhead.

When preemption is allowed at any time during task execution (as in fully preemptive systems), γ is typically evaluated in the worst-case scenario, which provides a safe, but pessimistic, bound on the preemption cost. As showed by Bui et al. [6], the task execution time increment due to cache interference can be as large as 33% for PowerPC MPC7410 with 2 MByte two-way associative L2 cache.

Finally, the total increase of the WCET of task τ_i is also a function of the total number of preemptions experienced by τ_i, which in turn depends on the task set parameters, on the activation pattern of higher priority tasks, and on the specific scheduling algorithm. Such a circular dependency of WCET and number of preemptions makes the problem not easy to be solved. Some methods for estimating the number of preemptions have been proposed in [9], [23], but they are restricted to the fully preemptive case.

It is worth pointing out that all the presented causes of overhead are closely related and should be considered in an systematic manner. Due to page limits, however, this paper focuses on scheduling methods for controlling the number and the position of preemptions. We analyze different limited preemptive scheduling methods, and investigate how the preemption-related problems can be dealt with from the scheduling point of view.

The rest of the paper is organized as follows: Section II introduces some terminologies and presents the considered scheduling methods; Section III assesses them and discusses their pros and cons; Section IV illustrates some simulation results, finally, Section V states our conclusions and future work.

II. Limited preemption methods

We consider a set of n periodic and sporadic real-time tasks to be scheduled on a single processor by a fixed priority algorithm. Each task τ_i is characterized by a worst-case execution time (WCET) C_i, a relative deadline D_i, and a period (or minimum inter-arrival time) T_i. A constrained deadline model is adopted here, so D_i is assumed to be less than or equal to T_i. For scheduling purposes, each task is assigned a fixed priority P_i, used to select the running task among those tasks ready to execute. A higher value of P_i corresponds to higher priority.

Notice that task activation times are not known a priori and the actual execution time of a task can be less than or equal to C_i. Tasks are indexed by decreasing priority, i.e., $\forall i \mid 1 \leq i < n : P_i > P_{i+1}$. For convenience, $hp(i)$ is used to represent the subset of tasks with priority higher than P_i.

Preemption is considered a pre-requisite to meet timing requirement in real-time system design, however, in most cases a fully preemptive system will produce many unnecessary preemptions. This phenomenon has been observed by Jeffay [12], where he conjectured that *"if preemption is required for feasibility, it will be limited to a few tasks"*. Following this reasoning, several limited preemptive scheduling algorithms have been proposed and studied. Clearly, limited preemptive scheduling becomes a key problem when designing reliable, dependable, and predictable real-time embedded systems, and it follows the research trends in cyber physical systems that integrate both cyber and physical components. The following scheduling algorithms, which are variations of the classical Fixed Priority Scheduling algorithm [16], are considered in this paper and the performance of different schedulers are evaluated in a comparative way.

- Fully Preemptive Scheduling (FPS). It corresponds to the classical fixed priority algorithm where each task can be preempted anytime and anywhere inside its code.
- Non-Preemptive Scheduling (NPS). It corresponds to the fixed priority algorithm where preemption is disabled. In this way, each task is selected based on its priority, and it runs until completion once gets started.
- Preemption Threshold Scheduling (PTS) [20]. Each task τ_i is assigned a nominal priority level P_i and a preemption threshold π_i, which is higher than or equal to P_i. Thus, τ_i can be preempted by τ_k only if $P_k > \pi_i$.
- Floating Non-Preemptive Regions (f-NPR). In this case, each task τ_i can disable preemption for a time interval of at most Q_i units of time. When a higher priority task arrives, the running task can switch to non-preemption mode for Q_i units of time, before the preemption is triggered. Since the running task can switch to non-preemptive mode at any time depending on the arrival of high priority tasks, the non-preemptive regions are assumed to be *floating* inside the task code.
- Fixed Preemption Points (FPP) [7]. Preemption can only take place in pre-defined positions, which are called preemption points. Thus, each task is divided into a set of fixed non-preemptive regions, among which the longest and the last ones play a crucial

role in the schedulability analysis. We denote these two lengths as q^{max} and q^{last}, respectively.

The above limited preemptive algorithms also differ in the way they can be implemented. Floating Non-Preemptive Regions can be realized by setting a timer to enforce the maximum interval in which preemptions are disabled. On the other hand, in FPP, preemption points are statically inserted inside the application code as function calls to the scheduler. This simplifies the scheduler implementation, since no timer is needed, but requires a user intervention to explicitly specify where to insert each preemption point.

Techniques to estimate the cache-related preemption delays have been proposed in [11], [14]. However, most research results considered only a single task in the analysis. A method to incorporate the effect of instruction cache on response time analysis has been proposed in [13]. Only recently, some more general frameworks [17], [19] have been proposed to deal with multi-task real-time systems. Finally, a partial preemptive model [18] has been proposed to consider the preemption cost and limited preemption. However, each task is limited to have only one non-preemptive region.

III. Assessment of the approaches

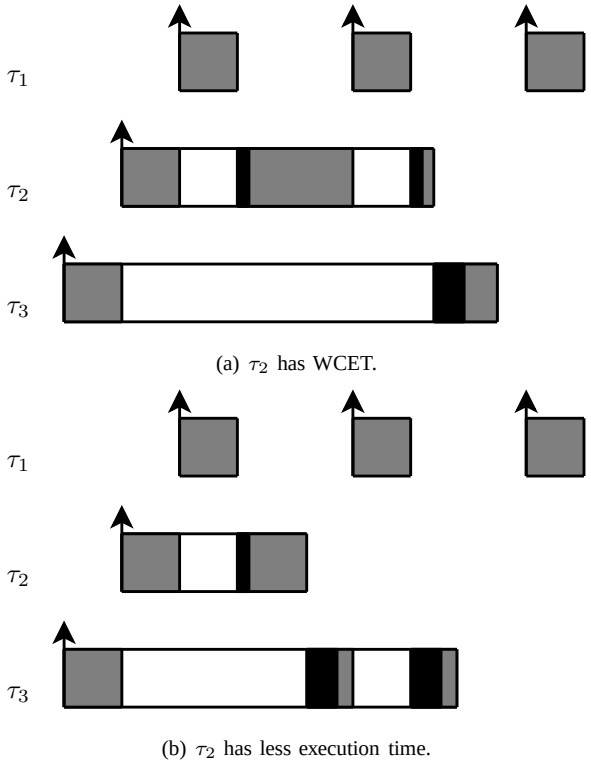
Although several different methods for reducing the number of preemptions have been proposed and analyzed, a comparative evaluation of these approaches is still missing. In this section, we discuss advantages and disadvantages of these various solutions and assess them in terms of effectiveness, implementation complexity, and impact on schedulability.

A. Non-preemptive scheduling (NPS)

The simplest way to limit the overhead due to preemption is to execute each task in a non-preemptive fashion. In this way, each task, once started, will continue until completion without being preempted. As a consequence, the total number of preemptions in the system will be zero. One major drawback of NPS is that the utilization loss is relatively high and high priority tasks experience large blocking delays from lower priority tasks.

Indeed, in non-preemptive systems it is easy to see that there is no least upper bound on the processor utilization below which the schedulability of any task set can be guaranteed. This can easily be shown by considering a set of two periodic tasks, τ_1 and τ_2, with priorities $P_1 > P_2$ and utilization $U_i = \epsilon$, arbitrarily small. If $C_2 > T_1$, $C_1 = \epsilon T_1$, and $T_2 = C_2/\epsilon$, the task set is unschedulable, although having an arbitrarily small utilization.

A fair assessment of the non-preemptive approach in comparison with the fully preemptive scheduling (FPS) can only be done when the preemption overhead is taken into account. However, a precise estimation of the preemption cost under FPS is not trivial. In fact, since high priority tasks may arrive at any time, the preemption cost must be estimated in the worst-case condition, leading to pessimistic results.



(a) τ_2 has WCET.

(b) τ_2 has less execution time.

Fig. 2. Number of preemptions occur by τ_3.

The total number of preemptions ν_i experienced by a task τ_i is also difficult to compute off line. One proposed method is to consider all the activations of higher priority tasks occurring within the response time R_i of task τ_i, that is:

$$\nu_i = \sum_{\tau_k \in hp(i)} \left\lceil \frac{R_i}{T_k} \right\rceil .$$

However, this approach leads to very pessimistic results. Consider, for example, the task set depicted in Figure 2, where gray areas represent task execution and black ones represent preemption cost. As shown in Figure 2(a), the first two instances of τ_1 do not preempt τ_3. However, Figure 2(b) illustrates that, if τ_2 completes earlier than its WCET, τ_3 suffers one more preemption.

Another factor which makes estimating the number of preemptions difficult is that each preemption introduces an extra execution time (γ) on the preempted task, as discussed in Section I. The accumulation of such overheads leads to an increment of the task remaining execution time, so possibly increasing the number of potential preemptions. Such a circular dependency between WCET and

number of preemptions makes the overhead very difficult to be accounted in the analysis of fully preemptive systems.

B. Preemption threshold scheduling (PTS)

Preemption threshold scheduling (PTS) can be considered as a trade-off between FPS and NPS. Indeed, if each threshold priority is set equal to the task nominal priority ($\forall i \mid 1 \leq i \leq n : \pi_i = P_i$), the scheduler behaves the same as FPS; whereas, if all thresholds are set to the maximum priority, the scheduler behaves the same as NPS. In addition, Wang and Saksena [20] showed that, by appropriately setting the thresholds, PTS can achieve a lower utilization loss compared to both FPS and NPS, and consequently, a higher utilization efficiency.

When counting the potential preemptions for task τ_i, only tasks with priority higher than π_i must be taken into account. However, when considering the exact number of preemptions for one specific task, the situation illustrated in Figure 2 may still occur under PTS. The only difference between PTS and FPS is that, under PTS, there are less tasks that can preempt the current task τ_i, i.e., the potential preempting task subset changes from $hp(i)$ to $\{\tau_k \mid P_k > \pi_i\}$. However, the total number of preemptions for a specific task is still difficult to estimate.

Moreover, under PTS, a preemption takes place immediately on the arrival time of a higher priority task, if the preemption is allowed. Hence, if arrival times are not known a priori, preemptions can occur at any position of the task code. As already pointed out, the arbitrary position of a preemption within the task code requires considering the worst-case scenario in the evaluation of the preemption cost, thus leading to pessimistic analysis.

C. Floating non-preemptive regions (f-NPR)

An alternative approach to limit the number of preemptions occurring on a task τ_i is to disable preemption for a time interval at most long Q_i. Since the running task can switch to non-preemptive mode at any time, the non-preemptive regions are assumed to be *floating* inside the task code, meaning that their start time is unknown.

The maximum length Q_i that τ_i can execute in non-preemptive mode to preserve feasibility has been computed both under EDF [2] and fixed priority [21]. After the value Q is computed off line, the run-time scheduling policy is working in the following way: when a task instance is selected to start by the global scheduler, it executes in regular mode. Suppose a new task instance with higher priority arrives, the current task will not surrender the processor immediately; instead, it will switch to non-preemptive mode and continue for extra Q_i units of time,

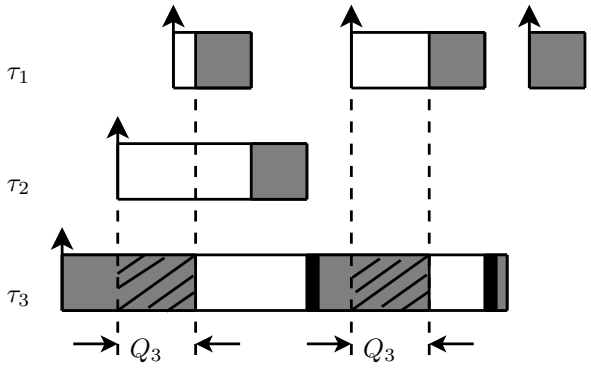


Fig. 3. A simple task set scheduled by limited preemption with floating NPR.

or the remaining execution time of the current instance, whichever is shorter. Figure 3 illustrates a sample task set scheduled under f-NPR, where the dashed regions represent the sections of code executed in non-preemptive mode. Notice that the activation of τ_2 does not preempt τ_3 immediately. Instead, τ_3 switches to non-preemptive mode and continues execution for Q_3 units of time. When the first instance of τ_1 arrives, τ_3 is still executing in non-preemptive mode, thus preemption is deferred until τ_3 finishes this non-preemptive region.

It has also been shown that, given a preemptively feasible task set, the computed length Q_i is always non-negative. Therefore, a preemptively feasible task set is still feasible under f-NPR, provided that the length of each non-preemptive region does not exceed the value Q_i. As there are some unfeasible task sets under fixed priority FPS and NPS, which can be successfully scheduled by f-NPR, we can conclude that f-NPR dominates FPS. This can be shown by considering the flowing example.

Example 1. Consider the task system composed of two sporadic tasks $\{\tau_1 = \{2, 4\}, \tau_2 = \{3, 6\}\}$, where the first number is the task WCET and the second number is the task minimal inter-arrival time and the relative deadline. Suppose tasks are assigned fix priorities and τ_1 has the higher priority. It can be easily verified that this task set is unfeasible under fixed priority FPS and NPS scheduling. However, by setting Q_2 equal to $2 - \epsilon$, where ϵ is an arbitrarily small positive value, the task set becomes feasible under f-NPR.

Notice that, once a task τ_i starts executing, it will continue for at least Q_i units of time, unless it completes earlier; hence, the preemption may take place at any place in the task code, except the first Q_i units of time. Consider the example task set in Figure 3, the preemption occurs when τ_1 (or τ_2) arrives after τ_3 has already started, otherwise τ_1 (or τ_2) will start before τ_3, hence no preemption will happen. When higher priority task arrives after τ_3 gets started, τ_3 will continue for Q_3 units of time before the preemption. For this reason, f-NPR slightly reduces the

possible preemption positions compared to PTS, where preemptions can occur anywhere in the code.

Under f-NPR, the maximum number of preemptions that τ_i can experience is upper bounded by $\lceil C_i/Q_i \rceil - 1$. It is worth pointing out that this estimation is independent of the number of tasks in the system, which might be rather large in some practical systems. As showed in [21], under fixed priority, the average value of Q_i/C_i is usually greater than 0.5 for a ten-tasks system even under high system utilization (90%),[1] thus this method provides a good solution for estimating the number of preemptions.

Under f-NPR, the circular dependency between the WCET of τ_i and the number of preemptions ν_i experienced by τ_i can be treated using the following recurrent relation:

$$\begin{cases} \nu_i^0 = \left\lceil \frac{C_i}{Q_i} \right\rceil - 1 \\ \nu_i^s = \left\lceil \frac{C_i + \gamma \nu_i^{s-1}}{Q_i} \right\rceil - 1 \end{cases}$$

Where the iteration process converges when $\nu_i^s = \nu_i^{s-1}$ and ν_i is the worst-case estimation. Notice the value Q_i is calculated from $hp(i)$, hence, it is available if the computation is performed in decreasing priority order.

D. Limited preemption with Fixed Preemption Points (FPP)

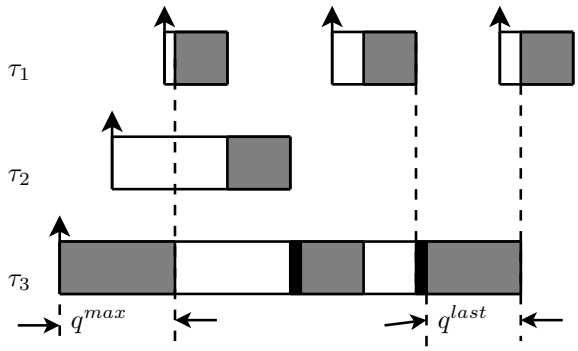


Fig. 4. A simple task set scheduled by FPP.

Enabling preemption to occur only at predefined positions in the task code allows achieving higher predictability. Indeed, the preemption points inside each task can be properly selected at design time to reduce the architecture related cost as much as possible. A sample task set scheduled by FPP is given in Figure 4. Task τ_3 is divided into three NPRs, and preemptions can occur only at the NPR boundaries. Notice that the third instance of τ_1 arrives during the execution of τ_3's final chunk, thus the preemption is avoided.

[1]In [21], all results are derived ignoring preemption cost.

It is worth observing that the f-NPR method is more general than FPP, since f-NPR allows NPRs to be anywhere in the task code. In other words, the value Q_i, which is computed under f-NPR model in Section III-C, can still be used as the maximum length of NPRs under FPP model, without violating the schedulability of the task set. However, f-NPR leads to more pessimistic analysis, because the architecture related cost must be considered in the worst-case scenario, which becomes unnecessary under FPP.

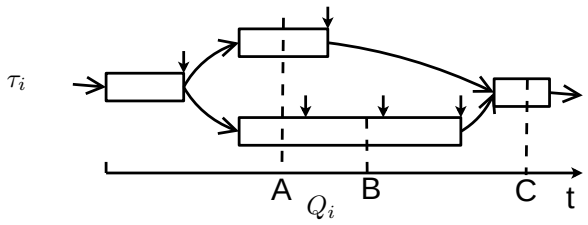
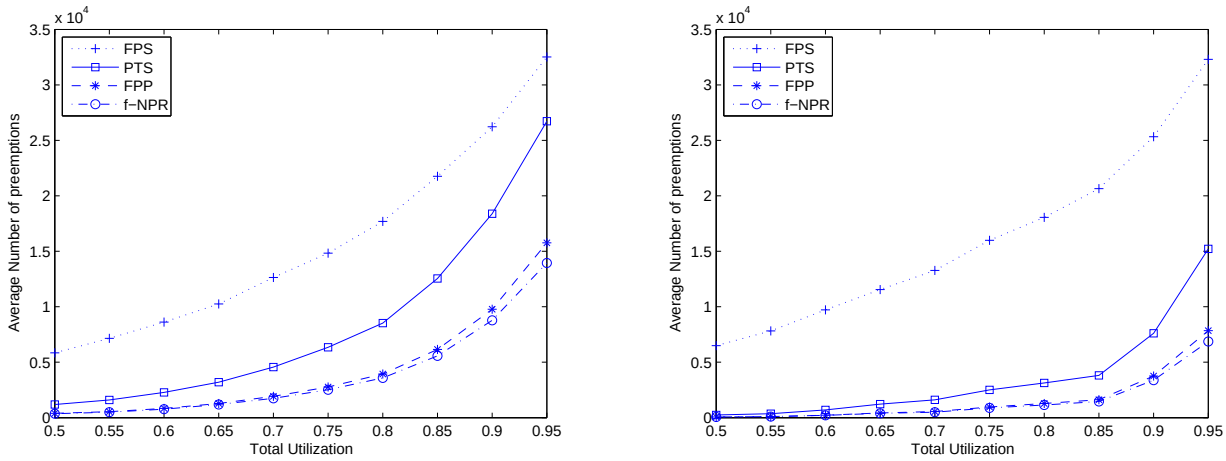


Fig. 5. A simple task program represented by control flow graph.

The key difference between f-NPR and FPP model lies in the selection of preemption points: under f-NPR, the points are dynamically decided at run time, according to the activation of higher priority tasks, while under FPP, the possible preemption points are statically inserted in the task code at the system design stage. Figure 5 illustrates a simple task program, which is represented as a control flow graph. The edges represent the possible control flows, and each box represents a basic block (sequences of instructions) with the length meaning the estimated execution time of this block. Under FPP model, since the preemption points are already specified (as the down arrows above the blocks), preemptions will be postponed to the next point in the current program path. In the figure, if high priority task arrives at position A, under FPP model the preemption will happen at the next down arrow position, while under f-NPR model, it will happen after the timer of scheduler counts Q_i units of time, that is, at position B if the current task follows the lower branch, or position C otherwise.

Another difference between f-NPR and FPP is given by the length of the final execution chunk. As showed in Figure 3, under f-NPR the final preemption on τ_3 can be very close to the task end, depending on the arrival time of high priority tasks. On the other hand, under FPP, the final chunk has a fixed length q^{last} (as in Figure 4). A long final chunk decreases the interference from higher priority tasks, possibly reducing the task response time. As a consequence, the task can tolerate more blocking from lower priority tasks. Therefore, we conjecture that under FPP tasks can have longer NPRs than under f-NPR.

When preemption points are defined for each task, the



(a) Number of tasks: n=6

(b) Number of tasks: n=12

Fig. 6. Average number of preemptions with different number of tasks.

task set schedulability under FPP can be verified through response time analysis [5]. However, this algorithm needs to check multiple instances within a certain time interval and thus has a high computational complexity. Burns and Wellings also investigated this problem in their latest book [8], showing that the feasibility test may be reduced to the first job of each task if the worst-case response time of each task is no larger than its period. We [22] independently investigated this topic and formally proved that under which conditions the schedulability test can be simplified to the first job of each task.

The FPP method can provide better timing predictability in the system design. Suppose the preemption cost at each program point is known from some timing analysis methods, the designers can select the possible preemption points by taking into account both the system feasibility and predictability. One preliminary result is presented in [3] to select the least number of points, under the assumption of fixed preemption cost.

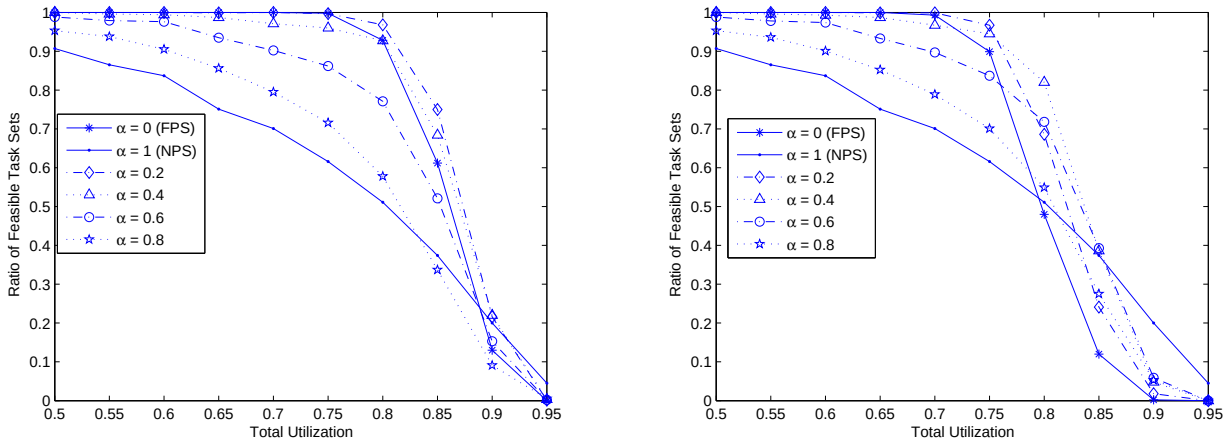
Under FPP the maximum number of preemptions is easy to compute and is equal to the number of preemption points in the code. Also, since preemptions can only take place at some pre-defined positions, the architecture related cost can be better estimated. Hence, a feasible task set under f-NPR is also feasible under FPP, meaning that FPP can achieve higher utilization level.

IV. Simulation results

A set of simulations with randomly generated task sets have been performed to better evaluate the effectiveness of the considered algorithms. Unless otherwise stated, each simulation run was performed on a set of n tasks with total utilization U varying from 0.5 to 0.95 with step 0.05. Individual utilizations U_i was uniformly distributed in [0,1], using the UUniFast algorithm [4]. Each computation time C_i was generated as a random integer uniformly distributed in [10, 50], and then T_i was computed as $T_i = C_i/U_i$. The relative deadline D_i was generated as a random integer in the range $[C_i + 0.8 \cdot (T_i - C_i),\ T_i]$. The total simulation time was set to 1 million units of time. For each point in the graph, the result was computed by taking the average over 1000 runs.

In the first simulation the number of preemptions produced by each approach was monitored and reported in Figure 6. The number of tasks n was set to 6 and 12, respectively. We restricted to preemptive feasible task sets and ignored the preemption cost as in [20], [21]. Under PTS, the algorithm proposed by Wang and Saksena [20] to find the maximum priority threshold was used to minimize the number of preemptions. Under f-NPR and FPP, the longest non-preemptive region was computed according to the method presented in [21], and for FPP case, the preemption points were inserted from the end of task code to the beginning.

As expected, FPS generates the largest number of preemptions, while f-NPR and FPP are both able to achieve a higher reduction. PTS has an intermediate behavior. Notice f-NPR can reduce slightly more preemptions than FPP since on average, each preemption is deferred more time than FPP (please refer to Figure 5), however, we have to consider that FPP can achieve a lower and more predictable preemption cost, since preemption points can be suitably decided off line with this purpose. As showed in the figure, FPS produces similar number of preemptions when the number of tasks increases, while all other three methods can reduce the preemptions number to a even higher degree. This is because each task will have smaller individual U_i, thus can suffer more blocking from lower priority tasks.



(a) Preemption cost: $\gamma = 3$

(b) Preemption cost: $\gamma = 6$

Fig. 7. Ratio of feasible task sets under different preemption cost.

	Preemption Position	Number of Preemptions	Schedulability Level
FPS	any position	maximum	medium
NPS	-	zero	low
PTS	any position	medium	high
f-NPR	any except first Q_i	low	high
FPP	pre-defined	low	high

TABLE I. Comparison of different scheduling policies

In a second simulation, we evaluated how the schedulability of the task sets changes as a function of the length of non-preemptive regions, by monitoring the percentage of feasible tasks sets at a given utilization level U. The schedulability is verified by simulation, since it is highly intractable to simulate all possible arrival patterns, the results are only indicative to illustrate how the system schedulability is affected by each model. Under each scheduling algorithm, a preemption cost (γ) was added to the remaining execution time of the preempted task, after each preemption. We considered two cases when γ was equal to 3 and 6, respectively. The context switch cost (σ) was assumed to be accounted in the task WCET. In this simulation, all NPRs were assumed to be floating in the task code and a parameter α was used to control the length of NPRs: $\forall i \mid 1 < i \leq n : Q_i = \alpha * C_i$.

Let N_U denote the number of feasible task sets under utilization U, and $N_{tot} = 1000$ denote the total number of task sets. Figure 7 illustrates how N_U/N_{tot} varies as a function of U. Notice that FPS and NPS can be considered as two special cases, with $\alpha = 1$ and $\alpha = 0$, respectively. In the figure they were represented by two solid lines. An interesting result found in this experiment is that, under high workloads, limited preemption method with properly selected value of α can improve the system schedulability with respect to FPS. This became more evident when the preemption cost increases.

Since at different U level the feasible ratio may increase or decrease with α, in the third simulation, the system utilization U was varied from 0.05 to 0.95 with step 0.05 and the ratio of all feasible task sets was monitored, which is defined as the sum, among all utilization levels, of the total number of feasible task sets, divided by the number of considered task set, that is:

$$\frac{\sum_{U=0.05}^{U=0.95} N_U}{\sum_{U=0.05}^{U=0.95} N_{tot}}$$

Results were illustrated in Figure 8, which plots the ratio of all feasible sets against different NPR lengths (as value of α). The simulation considered different context switch costs (γ): 2, 4 and 6. When α becomes larger, all three algorithms performances become closer to NPS method, as expected. The feasible ratios become lower as γ increases, due to the negative impact of preemption cost. Another interesting result is that limited preemptive scheduling with some values of α out-performs FPS in terms of overall schedulability, and this phenomenon becomes more evident with higher preemption costs. Actually, the schedulability problem becomes a trade-off between preemption cost and extra blocking. In the simulation, α was defined uniformly for all the tasks, hence, the schedulability can be further improved if a different α value is properly selected for each task.

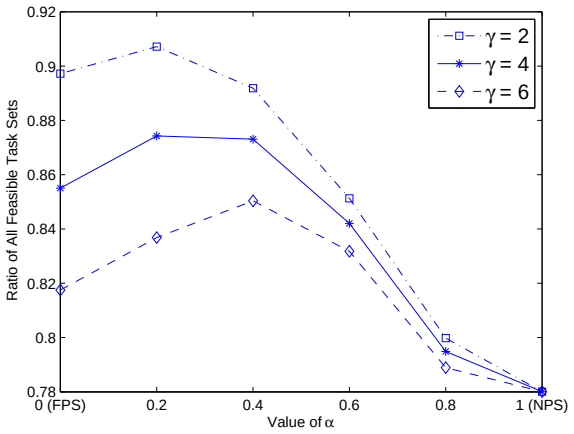


Fig. 8. Ratio of all feasible task sets with different preemption cost.

V. Conclusions

The results presented in this paper can be summarized in Table I, which compares the possible positions where preemption can occur, the number of preemptions, and the impact on schedulability. As showed in the table, under NPS the number of preemptions is zero, but the schedulability penalty is quite high due to the large blocking delays. PTS can reduce the overall number of preemptions with low schedule overhead, however, it cannot give a good bound on the number of preemptions for each task. f-NPR has good results on the number of preemptions and utilization level, however, it cannot provide enough knowledge on the preemption position as all the former policies. FPP is the most promising approach amongst all listed algorithms regarding the preemptions related issues.

Even though FPP shows its superiority from the preemption point of view, since there is a large space to be exploited in the system design, e.g., power/memory constraints, cache behavior, timing predictability and so on, it is still premature to conclude which one is the most promising method. How to apply the suitable method to optimize the system will be a future research topic.

References

[1] S. Altmeyer and G. Gebhard. Wcet analysis for preemptive scheduling. In *8th Int. Workshop on Worst-Case Execution Time Analysis*, pages 105–112, Prague, Czech, July 2008.

[2] S. Baruah. The limited-preemption uniprocessor scheduling of sporadic systems. In *ECRTS '05: Proc. of Euromicro Conf. on Real-Time Systems*, pages 137–144, July 2005.

[3] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo. Preemption points placement for sporadic task sets. In *ECRTS '10: Proc. of Euromicro Conf. on Real-Time Systems*, July 2010.

[4] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[5] R. Bril, J. Lukkien, and W. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Systems*, 42(1-3):63–119, 2009.

[6] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *Proc. of the Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 101–110, 2008.

[7] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. *S. Son, editor, Advances in Real-Time Systems*, pages 225–248, 1994.

[8] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX (Fourth Edition)*. Addison Wesley Longmain, 2009.

[9] J. Echague, I. Ripoll, and A. Crespo. Hard real-time preemptively scheduling with high context switch cost. In *Proc. of 7th Euromicro Workshop on Real-Time Systems*, pages 184–190, 1995.

[10] G. Gebhard and S. Altmeyer. Optimal task placement to improve cache performance. In *Proc. of the ACM-IEEE Int. Conf. on Embedded Software*, pages 259–268, Salzburg, Austria, 2007.

[11] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Trans. on Programming Languages and Systems*, 21(4):703–746, 1998.

[12] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *RTSS '92: Proc.s of Real-Time Systems Symposium*, pages 89–99, Dec 1992.

[13] J.V.Busquets-Matraix and et al. Adding instruction cache effect to an exact schedulability analysis of preemptive real-time system. In *Proc. of Euromicro Workshop on Real-Time Systems (EUROMICRO-RTS'96)*, pages 204 – 212, 1996.

[14] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. on Computers*, 47(6):700–713, 1998.

[15] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proc. of Workshop on Experimental Computer Science*, San Diego, California, 2007.

[16] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal ACM*, 20(1):46–61, 1973.

[17] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemption points. In *RTSS '06. Proc. of 27th Real-Time Systems Symposium*, pages 212–222, Dec. 2006.

[18] H. Ramaprasad and F. Mueller. Bounding worst-case response time for tasks with non-preemptive regions. In *RTAS '08: Proc. of Real-Time and Embedded Technology and Applications Symposium*, pages 58–67, April 2008.

[19] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proc. of Euromicro Conf. on Real-Time Systems*, pages 41–48, 2005.

[20] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proc. of Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 328–335, 1999.

[21] G. Yao, G. Buttazzo, and M. Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *Proc. of Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 351–360, China, 2009.

[22] G. Yao, G. Buttazzo, and M. Bertogna. Feasibility analysis under fixed priority scheduling with fixed preemption points. In *Proc. of Conf. on Embedded and Real-Time Computing Systems and Applications*, Macau, China, Aug. 2010.

[23] P. M. Yomsi and Y. Sorel. Extending rate monotonic analysis with exact cost of preemptions for hard real-time systems. In *Proc. of 19th EuroMicro Conf. on Real-Time Systems*, pages 280–290, 2007.