Accounting for overhead in fixed priority pre-emptive scheduling

Real-time Architectures
Exercise F

René Gabriëls Dirk Gerrits

June 25, 2007

2IN20 Real-time Architectures 2006/2007
Department of Mathematics & Computer Science
Technische Universiteit Eindhoven

0531594 René Gabriëls <r.gabriels@student.tue.nl>
0531798 Dirk Gerrits <dirk@dirkgerrits.com>

Contents

1 Introduction

The area of fixed priority pre-emptive scheduling (FPPS) has seen a lot theoretical literature on proving that deadlines hold. For this, several necessary and several sufficient conditions have been proposed, but sometimes an exact (necessary *and* sufficient) condition is needed to tell us if a certain task set is schedulable. For this one can calculate the *worst case response time* of the tasks. The task set is schedulable if and only if all worst case response times are less than or equal to the corresponding relative deadlines,

For both normal FPPS and hierarchical FPPS, formulas for the worst case response time have been developed. However, these usually assume that the overhead of scheduling and context-switching is 0 (or at least negligible). This is not the case in reality, so to apply the theory to actual systems, certain adaptations will have to be made to the formulas. In this paper we give an overview of the various techniques presented in the literature.

2 Non-hierarchical FPPS

2.1 Worst-case response time

In non-hierarchical FPPS we have periodic tasks τ_i, each with a computation time C_i, period T_i, relative deadline D_i, and a set of higher priority tasks $hp(i)$.

If we denote τ_i's blocking time (due to mutual exclusion on shared resources) as B_i, the (recursive) equation for its worst case response time is:

$$WR_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{WR_i}{T_j} \right\rceil C_j$$

(The B_i term is left out when blocking time is not taken into account, as in the original formula appearing in [LSD89].)

To compute the least fixed point of this equation one can use an iterative procedure, computing the value of $WR_i^{(k)}$:

$$WR_i^{(0)} = 0$$

$$WR_i^{(k)} = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{WR_i^{(k-1)}}{T_j} \right\rceil C_j \qquad \text{for } k \geq 1$$

for increasing k, until $WR_i^{(k)}$ exceeds D_i (meaning the task set is not schedulable), or until $WR_i^{(k+1)} = WR_i^{(k)}$ (meaning we have computed the worst case response time).

2.2 Overheads

In the implementation of an FPPS system one typically maintains two queues of tasks: the *delay queue* holding the tasks that have finished computing in their current period, and the

ready queue holding the tasks that haven't.[1] A timer interrupt periodically halts the execution of the then running process, allowing the system to update the queues and possibly select another process to run.

This leads to two kinds of overhead. The time taken moving newly-arrived or pre-empted tasks between the queues, and choosing a new task to be scheduled, is called the *scheduling overhead*. The time needed to pre-empt a task, save its context, load the context of another task, and resume that task, is called the *context-switching overhead* [SHAB03]. Various techniques have been developed on dealing with these.

2.3 Context-switching overhead as increased computation times

One of the most straightforward ways to deal with context-switch times in the literature is simply increasing the computation times C_i of all tasks τ_i by twice the time of doing a context-switch: $C_i := C_i + 2 \cdot C_s$. This accounts for one context-switch into the task, and one context-switch out of the task. This approach is taken in [KR90], [LVM91], [BWH93], and [BR99], and [ABD+95] reports that it's rather typical for FPPS. In [BR99] it is mentioned that the lowest priority task only needs to be charged one context-switch because it never pre-empts another task.

In [But05] a variation of this approach is used. Each task's computation time is instead increased by a number of context-switch times equal to the number of times that task will be pre-empted. This changes which context-switches are charged towards which tasks, but overall it's not more pessimistic than the above approach.

Another variation is to separate the two context-switch times into two (possibly different) numbers: the time to context switch to a task C_{s1}, and the time to context switch from a task C_{s2}. Then, for all tasks τ_i: $C_i := C_i + C_{s1} + C_{s2}$. This approach is taken in [KAS93] and [ABB96]. If C_s is taken as the average of C_{s1} and C_{s2} it is equivalent to the approach above, except possibly for the lowest priority task.

In [BTW94] it is noted that it's not needed for the entire task to finish before its deadline, only everything up to the last observable event. In particular, the context-switch away from the task at the end of its computation may occur after the task's deadline. For this $C_i^T (= C_i)$ is split into the time C_i^D up to the last observable event produced by the task, and the rest $(C_i^T - C_i^D)$. The formula for WR_i is updated to:

$$WR_i = B_i + C_i^D + \sum_{j \in hp(i)} \left\lceil \frac{WR_i}{T_j} \right\rceil C_j^T$$

(I.e., the outer C_i is replaced by C_i^D, but the C_j in the sum stays the same ($C_j = C_j^T$).)

2.4 Task scheduling overhead as increased computation times

In [BR99], this same approach for incorporating context-switching overhead in the computation times is applied to task scheduling overheads as well. The grand total of all overheads

[1] Other terms for the delay queue are the *wait queue* and *start queue*. Other terms for the ready queue include *run queue* and *dispatch queue*.

in the system is measured, and an average of this over all tasks is added to the computation times: $C_i := C_i + O_v$. The program used for overhead measurement is described in [BR99]'s appendix A.

2.5 Task scheduling overhead as additional tasks

Modeling the timer interrupt more directly can be done by introducing a new task with highest priority, a period T_{CLK} equal to that of the timer, and a computation time C_{CLK} that accounts for the updates of the task queues. This approach is taken in [LVM91], [KAS93], [Bur95], [ABB96] and [But05].

In [Bur95] it is observed that the cost of placing a task on the delay queue depends on the potential size of said queue (i.e., on the number of periodic tasks in the application). They instead make *two* additions to account for this:

- Add a timer interrupt task with highest priority as above, with computation time C_{CLK} being the overhead occurring on each interrupt assuming that there are tasks on the delay queue but none are removed from it.
- Add a set of *fictitious* tasks fpt, one for each of the original periodic tasks. The fictitious task corresponding to each periodic task has the same period, but computation time C_{PER} equal to the cost of moving one task from the delay queue to the run queue.

The updated formula for WR_i is:

$$WR_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{WR_i}{T_j} \right\rceil C_j + \left\lceil \frac{WR_i}{T_{CLK}} \right\rceil C_{CLK} + \sum_{f \in fpt} \left\lceil \frac{WR_i}{T_f} \right\rceil C_{PER}$$

This approach is also mentioned in [ABB96], but not worked out in detail.

In [BWH93] three approaches are mentioned: the two above, and a third, even more refined alternative. This third alternative is based on the observation that the overhead of moving the first task between queues is greater than that of moving any additional tasks in the same tick. The formula above is therefore still too pessimistic.

To model this the cost of moving n tasks is changed from $n \cdot C_{PER}$ to $ct_s + (n-1) \cdot ct_m$, where $ct_s (= C_{PER})$ is the cost of moving the first task, and ct_m the cost of moving any additional task.

The last term in the equation above is then refined as follows:

$$WR_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{WR_i}{T_j} \right\rceil C_j + \left\lceil \frac{WR_i}{T_{CLK}} \right\rceil C_{CLK} + I^F$$

$$K = \left\lceil \frac{WR_i}{T_{CLK}} \right\rceil$$

$$V = \sum_{f \in fpt} \left\lceil \frac{WR_i}{T_f} \right\rceil$$

$$I^F = \begin{cases} V \cdot ct_s & \text{if } K \geq V \\ K \cdot ct_s + (V - K) \cdot ct_m & \text{if } K < V \end{cases}$$

2.6 Evaluation

The approach in subsection 2.4 of measuring the total system overhead and averaging it out over all the task computation times is a very crude one. It was, however, successfully used in practice.

Subsections 2.3 and 2.5 describe complementary techniques that cover the context-switching overhead and task scheduling overhead, respectively. Each subsection describes several approaches, each producing tighter, less pessimistic bound than the previous one. The last approach of each subsection is therefore in some sense the "best": the least CPU time is wasted, and fewer task sets that would always meet their deadlines are falsely rejected.

These tighter bounds, however, have a larger number of more complex variables. It may not always be possible to accurately determine their values. And if such an accurate approach *can* be used, the results might not differ significantly from a simpler approach, which means the extra calculations are wasted.

3 Two level hierarchical FPPS

In two level hierarchical scheduling, multiple task sets are run on one processor. Each task set is usually called an *application*. The applications are scheduled on a global level, typically called *budget scheduling*, and then the tasks are scheduled on a local level (within the application). This scheme provides temporal isolation between tasks in different applications, i.e. a task in one application that overruns its deadline will not affect any task in another application. The scheduling policy for both levels (i.e. global and local) can be chosen freely. We will only consider the case where FPPS is used on both levels.

Applications are implemented as *servers*, that have a period T, capacity C and priority P. The capacity is the maximum computation time that the server's tasks may use during one server period. Each server is activated once during each period, and is suspended as soon as its capacity has been exhausted. There are 4 kinds of servers, characterized by how they deal with the capacity when no task is ready to run upon activation, and the time at which the capacity is replenished.

- Polling server: if no task is ready upon activation of the server, the capacity is lost immediately, until it is replenished at its next activation.
- Periodic server: if no task is ready to run upon activation, the capacity of the server is idled away as if a task was fully utilizing it. The capacity is replenishment at the next activation.
- Deferrable server: if no task is ready to run upon activation, the capacity is kept for tasks to consume until the end of the server's period. The capacity is replenished upon the next activation.
- Sporadic server: if no task is ready to run upon activation, the capacity is kept until there is a task ready to run. The capacity is replenished one period after a task starts using some of its capacity. The amount that is replenished, is the amount that has been consumed by a task until the server becomes idle again or runs out of capacity.

For each of these 4 servers, we can compute the worst case response time for a task belonging to some application. The same approach is used as for the non-hierarchical (classical) case as described in section 2. Davis and Burns [DB05] give a nice summary of the results that have been derived.

An important result is that rate monotonic priority assignment is not optimal when on the global level of two level hierarchical FPPS. Rather, the optimum server parameters have to be found using an exhaustive search algorithm.

3.1 Worst case response time

Saewong et al were the first to derive response time formulas for a set of servers under two level hierarchical FPPS [SRLK02]. Davis and Burns later tightened these formulas [DB05], on which the discussion in this subsection will be based.

The critical instant for a task τ_i under a polling, periodic, deferrable or periodic server S happens when it is released simultaneously with all higher priority tasks within the server (defined by the set $hp(i)$), just after lower priority tasks have used all the server capacity at the beginning of the server's period, while there is maximum interference from higher priority servers (defined by set $hp(S)$) during the last period the task is executed. The total response time is then equal to the sum of the following 3 terms:

1. The time it needs to execute the task and all higher priority tasks within the same server that have been released simultaneously (similar to the non-hierarchical case):

$$L_i(w) = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w}{T_i} \right\rceil C_j \tag{1}$$

2. The total length of the gaps in periods where the server's capacity is fully consumed (all but the last period), which equals the number of such periods times the gap length:

$$\left\lceil \frac{L_i(w)}{C_s} \right\rceil (T_s - C_s) \tag{2}$$

3. The maximum interference during the last period in which the server is executing the task under consideration:

$$I(w) = \sum_{x \in hp(S)} \left\lceil \frac{w - (\left\lceil \frac{L_i(w)}{C_s} \right\rceil T_s - C_s) + J_x}{T_x} \right\rceil C_x \tag{3}$$

where J_x is the release jitter of server x. For a deferrable server, $J_x = T_x - C_x$, while for the other servers this is 0.

The smallest fixed point can be found by a iterative procedure, although the formula needs to be modified a bit, because the third term is not monotonically non-decreasing. However,

all three terms together are monotonically non-decreasing.

$$w_i^0 = C_i + \left\lceil \frac{C_i}{C_s} \right\rceil (T_s - C_s) \tag{4}$$

$$w_i^{n+1} = L_i(w_i^n) + \left\lceil \frac{L_i(w_i^n)}{C_s} \right\rceil (T_s - C_s) + \sum_{x \in hp(S)} \left\lceil \frac{0 \uparrow (w_i^n - (\left\lceil \frac{L_i(w_i^n)}{C_s} \right\rceil T_s - C_s)) + J_x}{T_x} \right\rceil C_x \tag{5}$$

The procedure stops for task τ_i at step n if $w_i^n > D_i$ in which case the task is not schedulable; or when $w_i^n = w_i^{n-1}$, which equals the worst case response time of task τ_i.

If a task release always coincide to the release of its server (such a task is called a *bound* task), the results can be tightened a little, because we do not have to wait during the first period. The optimal priority assignment depends on whether the task set is bound or not, and whether the deadlines of the tasks are smaller than their periods. In bound task set, either rate or deadline monotonic is optimal. For an unbound task set, optimal priority assignments are more complicated.

3.2 Accounting for context switching overhead

Context switching overhead on the task level (local level) can be modelled by any of the methods described in 2. Context switching overhead on the server level (global level) can be modelled by charging the context switch time to the server's capacity, effectively lowering the server's capacity. This means when a server becomes active, a small portion of its capacity is consumed by context switching, before it can execute its activated tasks. This approach is taken in [DB05], but no formal changes are made to the formulas for the worst case response time. This is very similar to the non-hierarchical case, where the context switch time is added to to the task's computation time.

3.3 Accounting for scheduling overhead

Again, on the local level, any technique for the non-hierarchical case can be applied. On the global level, we haven't been able to find techniques which take the application scheduling overhead into account.

3.4 Evaluation

There are hardly any papers written about accounting for overhead in hierarchical fixed priority pre-emptive scheduling. And the one technique that we have been able to find, is only stated informally. This leads us to believe that this is still very much an open problem.

4 Conclusion

For non-hierarchical FPPS a lot of methods have been proposed to account for overheads in scheduling and context-switching. These approaches vary in complexity and tightness of the bounds. It depends on the situation which approach is preferable.

For hierarchical FPPS not a whole lot is known about accounting for overheads. On the local level, the non-hierarchical FPPS approaches can be used, but on the global level more research is required.

5 References

References

[ABB96] N.C. Audsley, I. J. Bate, and A. Burns. Putting Fixed Priority Scheduling Theory into Engineering Practice for Safety Critical Applications. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pages 2–10. IEEE Computer Society, 1996.

[ABD+95] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings. Fixed Priority Pre-emptive Scheduling. *Real-Time Systems*, 8(2-3):173–198, 1995.

[BR99] L. P. Briand and D. M. Roy. *Meeting Deadlines in Hard Real-Time Systems*. IEEE Computer Society, 1999.

[BTW94] A. Burns, K. Tindell, and A.J. Wellings. Fixed Priority Scheduling with Deadlines Prior to Completion. In *Proceedings of the 6th Euromicro Workshop on Real-Time Systems*, pages 138–142. IEEE Computer Society, 1994.

[Bur95] A. Burns. Preemptive Priority Based Scheduling. In *Advances in real-time systems*, pages 225–248. Prentice-Hall, 1995.

[But05] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Springer, 2nd edition, 2005.

[BWH93] A. Burns, A. J. Wellings, and A. D. Hutcheon. The Impact of an Ada Run-Time System's Performance Characteristics on Scheduling Models. In *Ada-Europe*, pages 240–248. Springer, 1993.

[DB05] R. I. Davis and A. Burns. Hierarchical Fixed Priority Pre-Emptive Scheduling. In *Proceedings of the 26th IEEE Real Time Systems Symposium*, pages 389–398. IEEE Computer Society, 2005.

[KAS93] D. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and Analysis of Fixed Priority Schedulers. *IEEE Transactions on Software Engineering*, 19(9):920–934, 1993.

[KR90] M. H. Klein and T. Ralya. An Analysis of Input/Output Paradigms for Real-Time Systems. Technical Report CMU/SEI-90-TR-19, Carnegie-Mellon University, Software Engineering Institute, Pittsburgh, Pa., 1990.

[LSD89] J. P. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 166–171. IEEE Computer Society, 1989.

[LVM91] C. D. Locke, D. R. Vogel, and T. J. Mesler. Building a Predictable Avionics Platform In Ada: A Case Study. In *Proceedings of the 12th IEEE Real Time Systems Symposium*, pages 181–189. IEEE Computer Society, 1991.

[SHAB03] A. Srinivasan, P. Holman, J. H. Anderson, and S. K. Baruah. The Case for Fair Multiprocessor Scheduling. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, pages 22–26. IEEE Computer Society, 2003.

[SRLK02] S. Saewong, R. Rajkumar, J.P. Lehoczky, and M.H. Klein. Analysis of hierarchical fixed-priority scheduling. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 173–181. IEEE Computer Society, 2002.