Proceedings of the 6th Real-Time Scheduling Open Problems Seminar (RTSOPS)

held in conjunction with the 27th Euromicro Conference on Real-Time Systems (ECRTS)

Edited by Marko Bertogna[1] and Arvind Easwaran[2]
[1]*University of Modena, Italy*
[2]*Nanyang Technological University, Singapore*

Lund, Sweden
July 7, 2015

Message from the Chairs

It gives us great pleasure to introduce this volume that includes the Proceedings of the 6th Real-Time Scheduling Open Problems Seminar (RTSOPS 2015). This volume represents the continued openness in the Real-Time Systems research community to share and discuss unsolved problems concerning real-time scheduling theory.

This 6th edition of the seminar series is co-located with the 27th Euromicro Conference on Real-Time Systems (ECRTS 2015) that is being held in Lund, Sweden. The daylong seminar features 12 open problems distributed over four sessions. Each session includes three papers and collaboration time. All participants are invited to interact in groups and tackle the presented problems. The hope is that we make headway in solving the problems and that more complete problem definitions and solutions will emerge as a result of the discussions initiated during RTSOPS 2015.

We would like to thank the generosity of the Program Committee for their time and attention to detail that helped us assemble the program for the day. We are also grateful to the RTSOPS Steering Committee for their feedback and advice. This program would not have been possible without the efforts and support of the ECRTS 2015 organizing committee and the General Chair, Karl-Erik Årzén.

We invite all of you to join us in taking advantage of this excellent opportunity to learn and interact with our fellow colleagues. We hope you enjoy RTSOPS 2015.

Marko Bertogna[1] and Arvind Easwaran[2]
[1]University of Modena, Italy
[2]Nanyang Technological University, Singapore
RTSOPS 2015 Program Chairs

RTSOPS 2015 Technical Program Committee

RTSOPS Steering Committee

Technical Program

A Note on Some Open Problems in Mixed-Criticality Scheduling

Pontus Ekberg and Wang Yi
Uppsala University, Sweden
Email: {pontus.ekberg | yi}@it.uu.se

In this note we list a few open problems that, despite being foundational for mixed-criticality (MC) scheduling theory, have received little or no attention in the existing literature. We also present a couple of (unpublished) claims related to the listed problems. The latter should, of course, be viewed with a healthy dose of skepticism.

We mainly consider the MC sporadic task model that is commonly used in the literature (sometimes called the "Vestal model"), see, e.g., [3] for a concise description of its semantics. To the best of our knowledge, all the listed problems remain open for all common variations of the system model. For example, they are open regardless of whether we assume (i) implicit, constrained or arbitrary deadlines, (ii) K criticality levels for any constant or non-constant $K \geqslant 2$, (iii) a uniprocessor or multiprocessor platform, (iv) preemptive or non-preemptive scheduling, or (v) any common subclass such as task sets with harmonic periods.

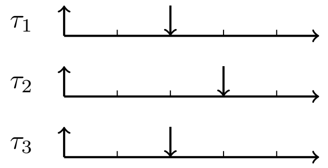
In the remainder of this note we assume a system with constrained deadlines, two criticality levels (called LO and HI) and a preemptive uniprocessor. It is also assumed that all schedulers are online (i.e., non-clairvoyant), like any real scheduler must be. In particular, a scheduler cannot know the actual execution time of a job except by executing it until completion, and it cannot know the exact release patterns of the sporadic tasks beforehand (it only knows about the *minimum* release separations).

Let a *job sequence* be a static sequence of unbounded length that specifies all the release times of jobs in one execution of the system. While the release times are fixed in a given job sequence, execution times are still unknown. For non-MC sporadic task systems on preemptive uniprocessors, it is known [1] that a particular job sequence generated by that task system—the synchronous arrival sequence (SAS)—is a guaranteed worst case, in the sense that if the SAS is schedulable, then so is every other sequence of jobs that the system can generate. This fact greatly simplifies feasibility analysis of such systems. Unfortunately, this property does not hold for MC sporadic tasks, as is demonstrated by the task set in Figure 1. It is easily verified that this task system has a correct scheduling policy for the SAS (execute τ_1 first, then τ_2 or τ_3 depending on τ_1's finishing time). However, it can also be verified that the job sequence where a job of τ_1 is released 1 time unit after the jobs from the other tasks does not have a correct online scheduling policy.

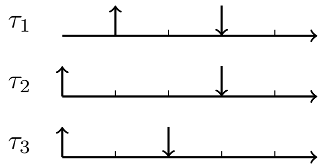
Claim 1: *The synchronous arrival sequence is not guaranteed to be the worst case for MC sporadic tasks.*

	$C(\text{LO})$	$C(\text{HI})$	D	P	Level
τ_1	1	2	2	10	HI
τ_2	1	1	3	10	HI
τ_3	1	–	2	10	LO

(a) The task system



(b) The SAS is schedulable



(c) An unschedulable sequence

Figure 1: A task system demonstrating that the SAS is not always a worst case.

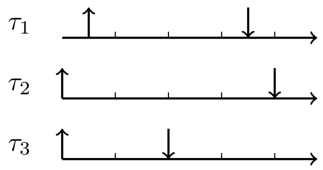
The situation is even worse if we allow non-integer release times of jobs. Figure 2 shows a task set for which all job sequences with integer release times are schedulable, but for which a job sequence with rational release times is unschedulable. This is in contrast to non-MC tasks, for which the SAS remains the worst-case even if non-integer release times are allowed.

Claim 2: *It is not enough to consider integer release times when analyzing sporadic MC task systems.*

The lack of a concrete job sequence to focus on severely hampers any effort to perform schedulability analysis of MC sporadic task systems. Can we identify a suitable replacement for the SAS?

	$C(\text{LO})$	$C(\text{HI})$	D	P	Level
τ_1	1	2	3	10	HI
τ_2	1	2	4	10	HI
τ_3	1	–	2	10	LO

(a) The task system



(b) An unschedulable sequence

Figure 2: A task set demonstrating that the worst case may have have non-integer release times.

Question 1: *For a given MC sporadic task system, can we efficiently identify some smallish set of job sequences, such that the task system is online schedulable if and only if all job sequences in that set are online schedulable?*

It is possible that the answer to Question 1 is "no", not only because it could be hard to identify such as set, but because such a set might not even exist. With current knowledge it is conceivable that an MC sporadic task system can generate two job sequences that share a common prefix, such that both job sequences have correct online schedules, but require different scheduling decisions during that prefix. Since an online scheduler for an MC sporadic task system presented with that prefix can not know which of the job sequences that eventually will be generated, it must make a scheduling decision during the prefix that is incorrect for at least one potential future. This is the motivation for the next question. If the answer to Question 2 is "no", then job sequences do not seem to be a good abstraction for analyzing MC sporadic task systems.

Question 2: *If all job sequences that can be generated by an MC sporadic task system have correct online schedules, does it follow that the task system has a correct online scheduler?*

Another task model to consider is the (strictly) periodic task model, in which each task system can generate only a single job sequence. If we are unable to find satisfactory answers to Questions 1 and 2, it seems as if the added flexibility that the sporadic task model offers over the periodic one comes at a high price in the MC setting. This is in contrast to the non-MC case where sporadic tasks are as easy to analyze as (synchronous) periodic ones on uniprocessors. Perhaps, then, research into the scheduling of MC periodic task systems is also a worthwhile effort? We are not aware of any results targeted at MC periodic tasks specifically.

Question 3: *Is scheduling or analysis for MC periodic tasks significantly easier than for MC sporadic tasks?*

Not only do we have a limited understanding of the difficult cases that may arise, but also poor understanding of how to best schedule MC systems. No optimal scheduling policy, of any complexity, is yet known for MC sporadic or periodic task systems.

Question 4: *What online scheduling policies are optimal for MC sporadic or periodic task systems?*

A remaining question concerns the computational complexity of the feasibility decision problem: does a given MC task system have a correct scheduling policy? Baruah et al. [2] have showed that the corresponding decision problem for (synchronous) fixed MC job sequences is strongly NP-complete for any constant K number of criticality levels ($K \geqslant 2$). It is also known that the feasibility problem for non-MC sporadic or periodic task systems is strongly coNP-complete. It is trivial to reduce the non-MC case to the MC case, and it is also trivial to reduce the feasibility problem for synchronous MC job sequences to the feasibility problem for MC periodic task systems (either synchronous or asynchronous). If follows that the feasibility problem for MC periodic task systems is both NP-hard and coNP-hard, and is therefore unlikely to be in either of NP or coNP. It is not clear that feasibility for MC sporadic task systems is also NP-hard, but it seems reasonable to suspect that it is at least as hard as feasibility for MC periodic tasks.

Question 5: *What is the complexity of the (online) feasibility problem for MC sporadic or periodic tasks?*

References

[1] S. Baruah and J. Goossens. Scheduling real-time tasks: Algorithms and complexity. *Handbook of scheduling: Algorithms, models, and performance analysis*, 3, 2004.

[2] S. Baruah, K., V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, Aug. 2012.

[3] H. Li and S. Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *RTSS*, pages 183 –192, 2010.

A scheduling model inspired by control theory

Sanjoy Baruah
The University of North Carolina

Abstract

Certain control computations may be modeled as periodic tasks with the correctness requirement that for each task, the fraction of jobs of the task that complete execution by their respective deadlines be no smaller than a specified value. This appears to be a correctness requirement that has not previously been studied in the real-time scheduling theory community.

In feedback (or closed loop) control, the state of a plant is monitored, the *error* — deviation of the monitored value from a desired value – is determined, and a *control signal* that should decrease the error is computed and applied to the plant; this entire process is repeated periodically. The computation of the control signal in such control loops is often modeled as a periodic task, with each iteration of the loop represented by a job of the task. In many application systems there are multiple control loops, each responsible for controlling a different aspect of the system's behavior, that run simultaneously upon a shared platform; determining an appropriate strategy for scheduling these control computations is thus a periodic scheduling problem.

The property of bounded input bounded output (BIBO) stability is desired of certain control systems. (Loosely speaking, a control system is BIBO stable if every bounded input to the system results in a bounded output.) Branicky, Phillips, and Zhang [2] explored the effect upon stability of skipping the computation of the control signal during some iterations and instead reusing the value that was used during the previous iteration. They derived techniques for determining a minimal fraction of control-signal computations that must be completed in order to ensure stability; in follow-up work, Majumdar, Saha, and Zamani [5] derived techniques for determining the minimal fraction of such computations that must be completed in order to additionally achieve optimal disturbance rejection performance. The work in [2], [5] suggests the following task model for the scheduling of control computations.

Task Model: Each control loop is modeled as a control task τ_i that is characterized by three parameters: $\tau_i = (C_i, T_i, r_i)$, where C_i is the WCET, T_i the period, and r_i a positive real number ≤ 1 denoting the desired asymptotic completion rate for task τ_i. Such a task generates *jobs*; the k'th job generated by τ_i has a release time[1] $(k-1) \times T_i$, an execution requirement no greater than C_i, and a deadline $k \times T_i$, for all integer $k \geq 1$. It is not required that all the jobs generated by τ_i execute. Let k_1, k_2 denote positive integers with $k_1 \leq k_2$, and let $\text{COMP}_i(k_1, k_2)$ denote the number of jobs out of the k_1'th, (k_1+1)'th, (k_1+2)'th, $\cdots, (k_2-1)$'th, k_2'th jobs generated by τ_i that do complete execution by their deadlines in some schedule. For the schedule to be considered correct for τ_i, it is required that

$$\lim_{k \to \infty} \left(\frac{\text{COMP}_i(1, k)}{k} \right) \geq r_i \tag{1}$$

Note that this is an *asymptotic* notion of correctness, which does not restrict what may happen with any sequence of successive jobs of any particular length. Consider, for example, a task τ_i with $r_i = \frac{1}{2}$; as per the definition above, a schedule that skips every alternate job is equally acceptable as one that skips the first million jobs and then schedules the next million.[2]

The open problem

Given a collection of n control tasks of the kind described above, determine whether it can be scheduled correctly upon a preemptive uniprocessor.

Some obvious extensions of this open problem are also of interest, both individually and in combinations:

1) Determine appropriate *algorithms* for scheduling collections of control tasks, and design efficient *schedulability conditions* for these scheduling algorithms.
2) Consider the *multiprocessor* version of this problem.

1. I.e., the tasks are periodic rather than sporadic, and all tasks have an initial offset equal to zero.
2. Clearly, this fact indicates that something was lost in translation in [2], [5] from control loops to periodic tasks – some unstated assumption that bounds, for example, the maximum deviation of $\left(\text{COMP}_i(k_1, k_2)/(k_2 - k_1 + 1)\right)$ from r_i for all k_1, k_2.

3) Consider more general task models. This could include periodic tasks with *offsets*; *sporadic* (rather than periodic) arrivals; models in which tasks are characterized by a *relative deadline* parameter in addition to WCET and period; and further generalizations.

As stated in footnote 2, the control tasks model appears inadequate for truly expressing desired properties of schedules for control tasks. We believe that it is important and potentially interesting to construct a more realistic task model for control loops that does indeed specify some such additional constraint. We consider the construction of such a model as an important open problem, which is probably best tackled by scheduling-theory and control-theory researchers working collaboratively.

Relationship to prior work

Several models have previously been proposed in the real-time systems literature that allow for the specification of the fact that some jobs of a task may be skipped during execution. Such models include the (m, k)-firm model [3], models that allow for the specification of a skip factor [4], and the weakly-hard task model [1]. However, it should be evident that the control tasks model described above differs widely from these prior models, all of which specify allowed job-drops within an interval of a certain number of successive jobs. In contrast, the control tasks model only requires that the fraction of correctly-scheduled jobs be no smaller than a specified ratio asymptotically. This characteristic of a real-time requirement that is only required to hold over arbitrarily large time intervals (as time approaches infinity), appears to be novel and previously completely unexplored.

A necessary (but not sufficient) schedulability condition. Observe that the quantity $\left(r_i \times \frac{C_i}{T_i}\right)$ denotes a tight upper bound upon the fraction of the computing capacity of a shared unit-speed processor that may need to be devoted to executing the i'th control task τ_i; hence,

$$\sum_{i+1}^{N} r_i \frac{C_i}{T_i} \leq 1 \tag{2}$$

is clearly a necessary condition for the system $\{\tau_1, \tau_2, \ldots, \tau_N\}$ to be schedulable upon a unit-speed processor. However, this condition is not sufficient: consider the system $\{\tau_1, \tau_2, \tau_3\}$ with $\tau_1 = \tau_2 = \tau_3 = (6, 10, 1/2)$. The summation evaluates to 0.9, but this instance is not schedulable since only one job can "fit" into each period while each of the three tasks desires to be scheduled during half the periods. (This example serves as a counter-example to [5, (Proposition 1)], which had claimed that Condition 2 is an exact test.)

A sufficient (but not necessary) schedulability condition. A trivial sufficient schedulability condition is obtained by simply not exploiting the ability to skip jobs. In that case, it is obvious that

$$\sum_{i+1}^{N} \frac{C_i}{T_i} \leq 1 \tag{3}$$

is clearly a sufficient condition for the system $\{\tau_1, \tau_2, \ldots, \tau_N\}$ to be schedulable upon a unit-speed processor. However, this condition is not necessary: it is obtained by completely ignoring the fact that for each i, a fraction $(1 - r_i)$ of the jobs of the i'th control task may be dropped without penalty.

Acknowledgements

This research was supported in part by NSF grants CNS 1115284, CNS 1218693, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and a grant from General Motors Corp.

References

[1] G. Bernat, A. Burns, and A. Liamosi, "Weakly hard real-time systems," *Computers, IEEE Transactions on*, vol. 50, no. 4, pp. 308–321, Apr 2001.
[2] M. Branicky, S. Phillips, and W. Zhang, "Scheduling and feedback co-design for networked control systems," in *Decision and Control, 2002, Proceedings of the 41st IEEE Conference on*, vol. 2, Dec 2002, pp. 1211–1217 vol.2.
[3] M. Hamadaoui and P. Ramanathan, "A dynamic priority assignment technique for streams with (m, k)-firm deadlines," in *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing (FTCS '94)*. IEEE Computer Society Press, Jun. 1994, pp. 196–205.
[4] G. Koren and D. Shasha, "D^{over}: An optimal on-line scheduling algorithm for overloaded real-time systems," Computer Science Department, New York University, Tech. Rep. TR 594, 1992.
[5] R. Majumdar, I. Saha, and M. Zamani, "Performance-aware scheduler synthesis for control systems," in *Proceedings of the Ninth ACM International Conference on Embedded Software*, ser. EMSOFT '11. ACM, 2011, pp. 299–308.

Dual-Priority Scheduling: Current Result, Observation and Open Problem

Risat Mahmud Pathan
Chalmers University of Technology, Sweden

Consider the problem of scheduling n implicit-deadline periodic tasks $\{(C_i, T_i) \mid i = 1, 2 \dots n\}$ on uniprocessor. The dynamic-priority based EDF algorithm can schedule such task set with total utilization 100% while fixed-priority (FP) based RM algorithm cannot always utilize 100% processor's capacity. In 1993, Burns and Wellings [1] proposed Dual-Priority Scheduling (DPS) based on so called *priority promotion* to give pure FP scheduling minimal dynamic-priority flavor. In DPS, a task's initially priority is increased (called, *promoted priority*) at most once, at fixed offset (called, *promotion points*) relative to the release of each job. A preemptive priority-based scheduling is used to schedule the tasks. This work presents challenges, some results, observations and open problems related to DPS.

Major Challenge: The main challenge of priority-promotion based scheduling is how to determine the promotion points and promoted priorities of the tasks (called, *promotion assignment problem*) such that all the deadlines are met (called, *schedulability testing problem*). Currently, the only way to tackle these problems (according to [1]) is by simulating the schedule up to the hyperperiod (i.e., least common multiple of the tasks' periods) for some given promotion assignment; and if certain deadline is missed during such simulation, then the schedule is again simulated after searching another potential promotion assignment, and so on. Such search is computationally impractical for large hyperperiod. Moreover, there exists no proof showing that such search would always find a feasible promotion assignment for a task set with total utilization less than or equal to 100% (which is a conjecture in [1]). In 2010, Burns [2] presented an open problem at the RTSOPS seminar: Is the utilization bound of DPS for implicit-deadline tasks on uniprocessor 100%? While Burns [2] solved this problem for task sets having $n = 2$ tasks, the answer to this question for $n > 2$ tasks is still unknown.

Current Result: Burns also presented a related open problem in [2]: Is there a m-priority promotion scheme that provides 100% utilization bound for any n? In m-priority promotion scheme, a task may have m promotion points where $m \geq 2$. This open problem is recently solved in [3] where a priority-promotion based scheduling, called Fixed-Priority with Priority Promotion (FPP), is proposed. In FPP, implicit-deadline tasks are indexed in rate-monotonic order (i.e., for any two tasks τ_k and τ_i where $k < i$, we have $T_k \leq T_i$). Assume that lower priority value implies higher priority. The promotion assignment problem in FPP is solved as follows: task τ_i's initial priority is i, and τ_i has $(i-1)$ priority promotions. The initial priority i is promoted to priority $(i-1), (i-2), \dots 2, 1$ respectively at promotion points $(T_i - T_{i-1}),\ (T_i - T_{i-2}), \dots\ (T_i - T_2),\ (T_i - T_1)$ as is shown in in Figure 1.

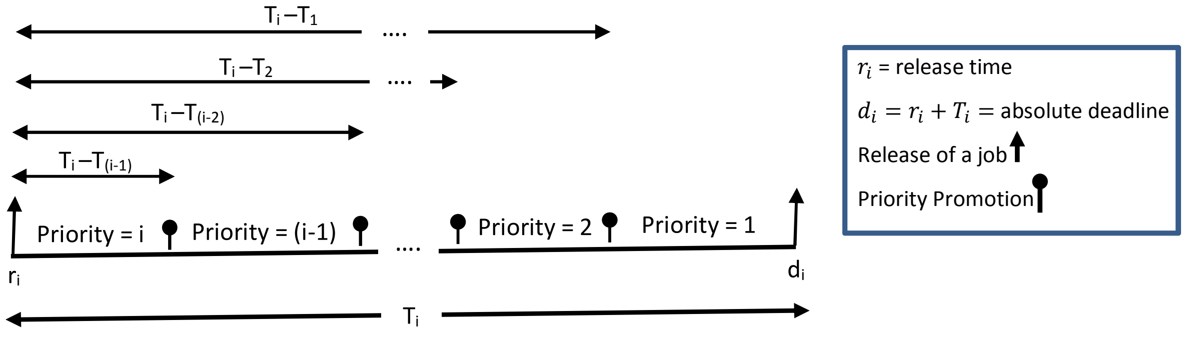


Figure 1. Promotion assignment (i.e., promoted priorities and promotion points) for task τ_i in FPP scheduling. The kth promotion point is $(T_i - T_{i-k})$ time units later than the release time r_i for $k = 1, 2 \dots (i-1)$. Between two consecutive promotion points at t_a and t_b, where $t_a < t_b$, the priority of the job remains at the priority level set at the earlier promotion point t_a.

It is shown in [3] that FPP essentially generates the EDF schedule of the tasks and thus has 100% utilization bound. Since the schedule generated by FPP is same as the EDF schedule, existing EDF schedulability test is used to determine the schedulability of a task set in FPP. In summary, the promotion assignment problem for FPP is solved to generate the EDF schedule, which avoids solving the schedulability testing problem thanks to existing EDF tests.

FPP scheduling requires more than one priority promotion per task and generates the EDF schedule. An interesting question is whether the EDF schedule can be generated using at most one priority promotion per task? Unfortunately, the answer is *negative* as is shown by Masson *et al.* in [4]. To see why consider a task set with $n = 3$ tasks where the WCET and periods (C_i, T_i) of the tasks are $\tau_1 = (8, 16),\ \tau_2 = (2, 31),\ \tau_3 = (26, 60)$. This task set is EDF schedulable since its total utilization is ≈ 0.9987. Priority-promotion based scheduling generates the EDF schedule of these three tasks as follows: tasks are executed in RM priority order and the priority of τ_3 is promoted twice – once at offset 31 over priority of τ_2 and again at offset 48 over priority of τ_1 – while the priorities of τ_1 and τ_2 are never promoted. Figure 2 shows the schedule in interval [0, 62).

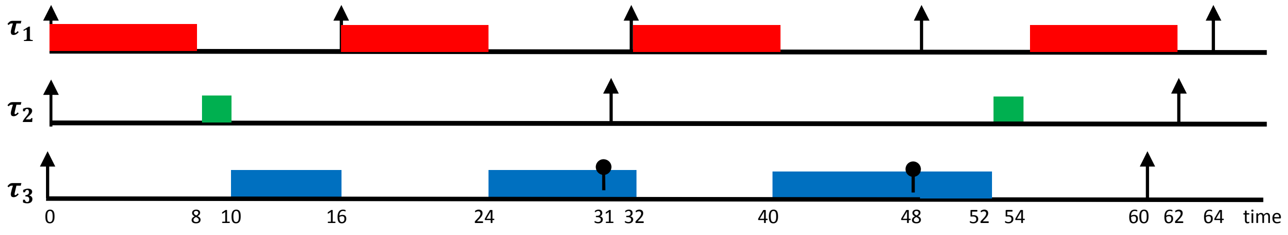


Figure 2: Each task has RM priority. The priority of task τ_3 is promoted ***twice*** while priorities of tasks τ_1 and τ_2 are never promoted. The resultant schedule is EDF and generated using priority-promotion based fixed-priority scheduling.

Observation: There exist task set for which some task must need to have more than one priority promotion to generate the EDF schedule. In general, DPS cannot always generate the EDF schedule.

By modifying the EDF schedule in Figure 2, a (modified) schedule of the tasks with at most one priority promotion per task can be derived. Consider the schedule in Figure 3 where τ_3's priority is promoted only once at offset 54 over priority of τ_1. The schedule in Figure 3 is generated by modifying the EDF schedule given in Figure 2.

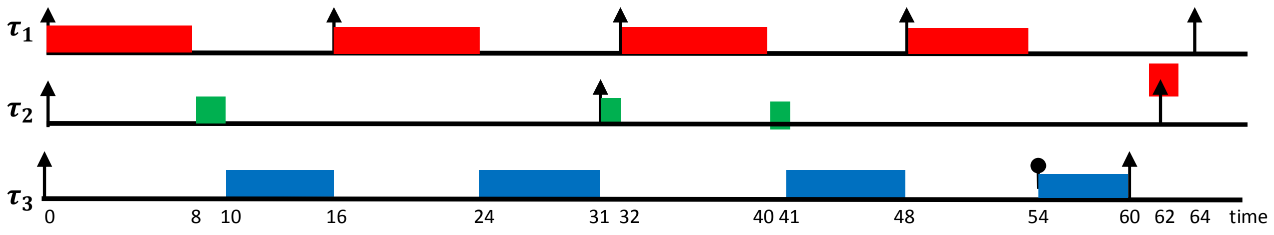


Figure 3: The priority of task τ_3 is promoted only ***once*** at time 54 over priority of τ_1 and the priorities of tasks τ_1 and τ_2 are never promoted. We simulated the schedule up to the hyperperiod and found that all deadlines are met using DPS.

Open Problem: Is there any efficient way to modify the EDF schedule of a task set without missing any deadline such that each task needs at most one priority promotion to generate this modified schedule using DPS?

Another problem of analyzing DPS is that the critical instant is not known [4]. The problem of not knowing the critical instant is also one of the major obstacles to perform exact schedulability analysis of global fixed-priority scheduling for multiprocessors. The state-of-the-art global fixed-priority schedulability test is a sufficient test [5].

Open Problem: How can we derive a *sufficient* dual-priority schedulability test for a given promotion assignment? If such a sufficient test is derived, how do we solve the promotion assignment problem for DPS? Should we try to solve the promotion assignment and schedulability testing problems simultaneously, as Burns [2] solved for $n = 2$?

Importance of Solving DPS: Fixed-priority scheduling is widely used in industry due to its ease of implementation (e.g., managing jobs in the ready queue) in comparison to EDF. The work in [5] proposed efficient mechanism to manage jobs in the ready queue of FPP scheduling. However, if priority promotion of multiple tasks in FPP is implemented using timers, then FPP could incur overhead in managing multiple timers in a timer queue. If DPS problem is solved, then there is at most one priority promotion per job and the ready queue management of FPP can be adopted to implement DPS very efficiently.

References

[1] A. Burns and A. Wellings. Dual Priority Assignment: A Practical Method of Increasing Processor Utilisation. In Proc. of the Fifth Euromicro Workshop on Real-time Systems, 1993.
[2] A. Burns. Dual Priority Scheduling: Is the Processor Utilisation bound 100%? In Proc. of the 1st International Real-Time Scheduling Open Problems Seminar (RTSOPS), in conjunction with the ECRTS, 2010.
[3] Risat Mahmud Pathan,''Unifying Fixed- and Dynamic-Priority Scheduling based on Priority Promotion and an Improved Ready Queue Management Technique'', In Proc of the RTAS, 2015.
[4] Damien Masson, Laurent George and Jöel Goossens, Dual Priority and EDF: a closer look, WiP session of RTSS, 2014.
[5] Risat Mahmud Pathan and Jan Jonsson, ''Interference-Aware Fixed-Priority Schedulability Analysis on Multiprocessors'', Real-Time Systems Journal, Vol. 50(4), 2014.

Analysis of self-interference within DAG tasks

José Fonseca, Vincent Nélis, Geoffrey Nelissen and Luís Miguel Pinho
CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Portugal
Email: {jcnfo,nelis,grrpn,lmp}@isep.ipp.pt

I. Introduction

Few years ago, the frontier separating the real-time embedded domain from the high-performance computing domain was neat and clearly defined. Nowadays, many contemporary applications no longer find their place in either category as they manifest both strict timing constraints and work-intensive computational demands. The only way forward to cope with such orthogonal requirements is to embrace the parallel execution programming paradigm on the emergent scalable and energy-efficient multi-core/many-core architectures.

Although the parallel programming paradigm is nothing fundamentally new, the real-time community has only recently started to actively investigate the repercussions of using parallel tasks in the analysis of the worst-case behavior of the systems. Several parallel task models and respective schedulability analysis have been proposed in order to enrich classical schedulability theory with an answer to these new challenges [1]–[4]. Small steps have been taken so far, and the progress is still strongly coupled to the well-established literature for sequential tasks.

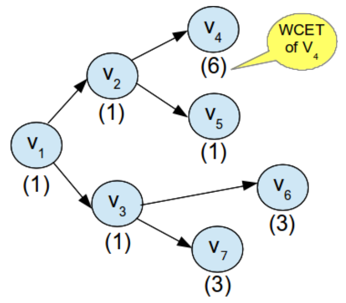
The most general model studied by the real-time community is the Directed Acyclic Graph (DAG) model [3]. It reflects sophisticated types of dynamic, fine-grained and irregular parallelism that can be achieved by using, e.g., the OpenMP programming model [5]. The resulting individual computational units (nodes of the DAG) are conventionally numerous and short-living for speed up purposes. Moreover, the nodes are mapped to OpenMP threads, and then scheduled through OS threads, according to a rather complex run-time dispatcher. Current works bypass all these run-time decisions and focus solely on the OS thread-to-core scheduling in a black-box manner. Unfortunately, the dynamic nature of the standard run-time manager along with the flexibility of the model complicate the derivation of accurate worst-case estimates both on execution time and on response time, which is even further aggravated by the choice of global scheduling schemes. In the following we describe one of the major sources of pessimism associated to such settings.

II. Problem description

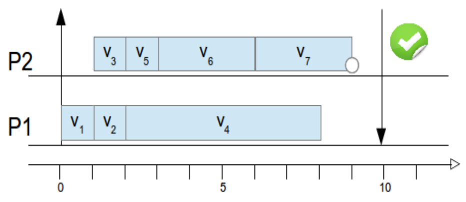
Under the real-time DAG model, parallel tasks are typically characterized by a minimum inter-arrival time T_i, an end-to-end relative deadline D_i, and a directed acyclic graph $G_i = (V_i, E_i)$. Each node $v \in V_i$ represents a sequential sub-task and is associated with a wost-case execution time (WCET), whereas each arc $(v_a, v_b) \in E_i$ represents a unidirectional precedence constraint between two different nodes, with the interpretation that sub-task v_b cannot begin execution until sub-task v_a completes. No other restriction is imposed by the model (e.g., global synchronization points are not mandatory and there is no limit on the degree of parallelism). Additionally, critical path L_i and workload W_i are two metrics of interest also defined in the model. Critical path corresponds to the longest (w.r.t. WCET) path of sub-tasks for which there are arcs connecting every two adjacent sub-tasks. Workload is the maximum execution requirement of the task, i.e. the sum of all nodes WCETs. From a feasibility perspective, only the following two necessary (but not sufficient) conditions must be met: 1) $L_i \leq D_i$, and 2) $\sum \frac{W_i}{T_i} \leq m$, where m is the number of cores in the system.

Regarding the priority assignment scheme, most of the schedulability-related works that assume the DAG model consider a task- or job-level priority system, whether the priorities are static or dynamic. That is, no unique priorities are assigned to the individual sub-tasks, and thus all sub-tasks are assumed to inherit the priority of the parent task/job (i.e. the priority of the DAG). In this sense, the actual sub-task scheduling is unknown in the analysis. In practice, it is the run-time environment who decides on-the-fly which sub-tasks are executed first. Without entering into details, FIFO or LIFO, in conjunction Breadth-First Scheduling (BFS) or Work-First Scheduling (WFS), are the usual policies enforced by the run-time environment to arbitrate the sub-task-to-thread assignment. Therefore the dispatching decisions greatly depend on the time instant at which sub-tasks are released, which in turn varies according to the completion time of their predecessors. If on top of that we consider a global scheduling algorithm, then broadly speaking, ready sub-tasks may execute anywhere and in many different orders.

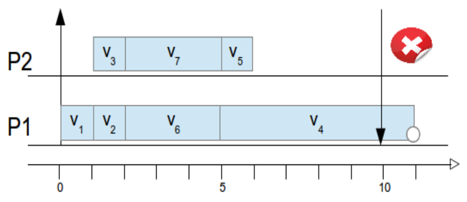
Due to this flexibility, interference of a DAG task on itself (referred as "self-interference" hereinafter) becomes a problem that should be considered by the scheduling algorithms and subsequently taken into account by the timing analysis in order to improve the utilization of the systems. Self-interference corresponds to the unexpected delay caused on the response time of a parallel task by its own sub-tasks. Although in general self-interference is an intrinsic property of parallel tasks, the main issue under the specified settings is that self-interference is not static. Meaning that two non-dependent sub-tasks may delay each other's execution in different instances of the same DAG task. The self-interference is thus mutual. Consequently, every path is penalized from a worst-case perspective. Let us consider a DAG task τ_i (as depicted in Fig. 1a) with deadline equal to 10 to be globally scheduled on a 2 cores platform. No other tasks co-exist in the system. Fig. 1b shows a schedule when sub-tasks v_4 and v_5 are dispatched first, resulting in a response time of 9 which clearly demonstrates that τ_i is feasible. However, it is possible that sub-tasks v_6 and v_7 get picked ahead, for instance due to a marginal early completion of v_3. In this scenario, the produced schedule (see Fig. 1c) causes τ_i to miss its deadline by 1 time unit, making the overall system infeasible. The fundamental difference resides on the self-interference shifted partially upon the critical path.



(a) DAG struture of τ_i. $L_i = 8$ and $W_i = 16$.



(b) A feasible global schedule for τ_i.



(c) A infeasible global schedule for τ_i.

Fig. 1: Feasibility problem of a DAG task τ_i with $D_i = 10$, under global scheduling.

While this simple example illustrates how dynamic self-interference may affect the feasibility of DAG tasks under global scheduling, the problem is further exacerbated when other tasks contend for access to the cores. The main reason behind relates to the fact that multiple paths contribute to the worst-case response time of lower priority tasks, and each one of such paths must be previously bounded according to the self-interference pattern that delays them the most.

It should also be noted that, although this paper focuses on core-level scheduling problems, the concept of self-interference propagates to the other sub-systems (e.g., memory access), posing even more challenges for WCET and overhead analysis.

III. Discussion on potential solutions

The self-interference problem has its roots on a too flexible software stack. In this section, we briefly argue about three different ways to restrict this flexibility and thus confine the problem. The first solution focuses on the sub-task scheduling, the second on the sub-task affinity, and the third on the DAG generation. Each one of the potential solutions minimizes the problem to a limited extent, hence a combination of them may lead to more satisfactory results.

1) **2-level priorities.** A separated fixed set of priorities is associated with the sub-tasks of each DAG task, so that the sub-task-to-thread assignment is no longer decided on-the-fly and the task scheduling is unaffected. Primarily the tasks dispute the cores, and only then the sub-tasks of the selected task contend for the access to its threads of execution. Sub-tasks run to completion because the preemption cost is generally unbearable for fine-grained parallel programming frameworks. Advantages: the self-interference is not completely static but it is easier to bound. Disadvantages: complexity of the queue management; how to assign priorities to the sub-tasks?
2) **Partitioned scheduling.** Each sub-task can execute only in one core. Different sub-tasks of the same DAG task may execute on different cores. The dispatching of sub-task is still done on a FIFO or LIFO basis. Advantages: the number of self-interference scenario may be largely reduced; some cores can be dedicated to critical paths with minimal slack; simplifies WCET-related analysis. Disadvantages: mapping problem is very likely to be NP-Hard.
3) **Extra precedence constraints**. Based on the feedback of timing analysis, create a procedure to provide the application's code with additional precedence constraints between certain sub-tasks. This kind of extra-functional dependencies can be enforced using OmpSs [6]. Advantages: optimize worst-case paths by establishing some order of execution. Disadvantages: cumbersome iterative process; restricts the degree of parallelism.

Acknowledgments

This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER Research Centre); by the North Portugal Regional Operational Programme (ON.2 – O Novo Norte) under the National Strategic Reference Framework (NSRF), through the ERDF, and by National Funds through FCT/MEC, within project NORTE-07-0124-FEDER-000063 (BEST-CASE, New Frontiers); and by the European Union under the Seventh Framework Programme (FP7/2007-2013), grant agreement nr. 611016 (P-SOCRATES).

References

[1] K. Lakshmanan, S. Kato, and R. R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *the 31st RTSS*, 2010, pp. 259–268.

[2] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *the 32nd RTSS*, 2011, pp. 217–226.

[3] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *the 33rd RTSS*, 2012, pp. 63–72.

[4] J. C. Fonseca, V. Nélis, G. Raravi, and L. M. Pinho, "A multi-dag model for real-time parallel applications with conditional execution," in *the 30th ACM/SIGAPP SAC*, 2015.

[5] OpenMP Architecture Review Board, "OpenMP application program interface version 4.0," 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[6] S. Royuela, A. Duran, and X. Martorell, "Compiler automatic discovery of ompss task dependencies," in *the workshop on Languages and Compilers for Parallel Computing*, 2012.

Online Admission of Parallel Real-Time Tasks

Cláudio Maia, Luís Nogueira, Luis Miguel Pinho
CISTER-ISEP / INESC-TEC, Porto, Portugal
{crrm, lmn, lmp} @isep.ipp.pt

I. Context and Problem Description

Parallel real-time tasks can be assigned into a multiprocessor system in many different ways, with regards to the schedulability of the task system. In the best case a parallel task may be completely parallelized, i.e., when the platform has a number of processors no smaller than the degree of parallelism of the task and the interference from higher priority tasks does not compromise its parallel behaviour; in the worst-case (considering the parallelism) the task needs to be completely serialized. This latter scenario requires the adjustment of the timing properties of the parallel task so that the worst-case parallel behaviour is considered in the analysis in order to avoid missing any deadlines. Moreover, this scenario is very pessimistic with respect to the parallel structure of the task. Considering parallelism and real-time, it is important to achieve a good trade-off between timing guarantees and the parallel execution of tasks so that efficient execution of such tasks may be possible.

The schedulability of a task system composed of parallel tasks can be analysed by using different techniques (e.g., response-time analysis [1], decomposition) and considering different task models (fork/join, synchronous or directed acyclic graphs). Decomposition-based techniques ([2], [3]) decompose a parallel task into a set of sequential tasks (usually by assigning intermediate deadlines between sub-tasks in order to maintain the precedence constraints between them), so that existing sequential schedulability analysis techniques for multiprocessors can be applied. An important aspect of these techniques is that they require the task structure to be known offline in order to apply decomposition. Alternatively, non-decomposition techniques such as the one proposed in [4] do not require offline knowledge of the tasks' structure.

While the above techniques are suitable for offline schedulability analysis, one cannot simply apply them in dynamic systems (i.e., systems where tasks may arrive and leave the system at any time), due to their algorithmic complexity. In dynamic systems task acceptance is done according to an admission control test that takes as input the current state of the system and the new task, and provides as output a decision whether the system remains schedulable after accepting the task. As these decisions must be made online, the test must be as efficient as possible so that the test itself does not become an overhead.

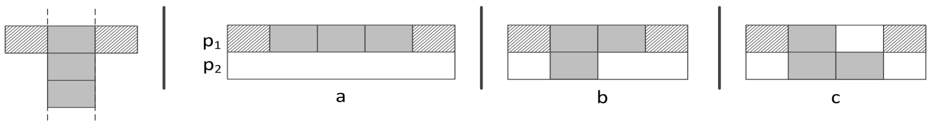


Fig. 1. Example of possible allocations of a fork/join parallel task with three segments, with one, three and one sub-tasks, respectively.

The analysis of such a system not only depends on the scheduling algorithm, but also on the properties of each parallel task. Let us consider a simple fork/join task composed of a parallel region with 3 sub-tasks and a platform of 2 identical cores as depicted in Figure 1 [1]. If a fully-partitioned scheduling algorithm is used, then the parallel task would have to be serialized, in order to be executed on a processor (see Figure 1, a); or if the task structure is known in advance, then it is possible to consider each sub-task individually, their release offsets and intermediate deadlines, and use a partitioned scheduling approach to assign each sub-task into a core (see Figure 1, b and c). Therefore, this will allow us to avoid migration overheads. If a global approach is used instead, then all the depicted allocations (a, b and c) may occur at runtime for different task instances, depending on the state of the system at a particular time instant.

The above example illustrates that different allocations are possible at runtime. Depending on the scheduling approach, the schedulability of the system might be affected according to several factors as, for instance, the priority of the parallel task under admission. Moreover, the example considers that the parallel task has a utilization not greater than one, otherwise it could not be serialized and assigned into a single processor (see Figure 1, a). If one considers the online admission of parallel tasks with a utilization greater than 1 (which is the most likely format for a parallel task), then one is faced with the forced parallelization of the task in order to obtain task schedulability.

Global approaches usually consider that a parallel task may have a utilization greater than one but not greater than the number of processors in the system, and these may not require the knowledge of the task structure in advance, at the cost of some pessimism in the analysis. On the other hand, if one considers partitioned approaches, then the task structure must be known in advance in order to decide where to allocate each sub-task, and therefore apply a schedulability test on each core. This leads to an online bin-packing problem, which is a NP-hard problem.

An admission control test that considers parallel tasks should be able to support all possible structures and timing properties of the tasks. Nevertheless, while it is possible to adapt existing techniques to consider parallel tasks with utilization no greater than 1, to the best of our knowledge there is **no admission control test that fits parallel real-time tasks with utilization greater than 1**.

[1]The figure does not depict any interference the task may suffer and the presented allocations are also valid if processor p_1 is switched by p_2 and vice-versa.

II. Possible solutions

Possible solutions for the problem of online admission control of parallel tasks may include variations of existing techniques for partitioned approaches such as the one proposed in [2]. Specifically, this approach decomposes a parallel task into a master thread which is assigned to one core in such a way that it fills the capacity of the core (100%). Then, the remaining parallel threads of the task are treated as constrained-deadline sub-tasks which are analysed by taking into account a release offset and intermediate deadlines. These constrained-deadline sub-tasks are assigned to processors by using the heuristic proposed in [5] (entitled FBB-FFD) which can be applied to task sets composed of sequential sporadic tasks if one selects partitioned deadline-monotonic scheduling. This approach has two limitations: (1) it requires an empty processor where the master thread of each task can be allocated; (2) it needs to be generalized to support different parallel task structures (originally it only supports fork/join tasks). From a conceptual standpoint, it seems to be a promising starting point with polynomial-time complexity. Following the same principle, another possible solution is the combination of the decomposition of parallel tasks with the constant-time admission control-test proposed by [6] and the first-fit decreasing (FFD) heuristic, which can be used for testing the schedulability of tasks scheduled by using a partitioned-EDF scheduler.

The previous approaches show that a possible solution to the online admission control for parallel real-time tasks using partitioned scheduling is one that during runtime decomposes a parallel task into its sub-tasks, and then each of these sub-tasks is subject to admission control during runtime. Then, one is faced with an online bin-packing problem that can be solved by combining an allocation heuristic such as FFD and an efficient schedulability test to evaluate the schedulability of the sub-tasks.

Global scheduling solutions for admission control of parallel tasks are more challenging to develop due to the need of knowing the global state of the system when new a task arrives. This may be affected by the migration of tasks that occurs among the cores. Another important aspect that should be taken into account is the interference that a task may suffer from the contention occurring in the hardware resources, which is difficult to determine.

III. Discussion

This paper describes the problem of online admission control of parallel tasks with a utilization greater than one in a multiprocessor system. Considering these tasks, not only their allocation is important, but also the schedulability test that is used when performing the admission control of the tasks. As presented in the previous section, it seems that a natural candidate solution for the online admission of parallel tasks is the decomposition of tasks into sub-tasks and then the resulting sub-tasks are submitted to admission control before assigning them into the processors. However, this approach is not the perfect solution as it presents problems concerning the need for synchronization and the computation of release offsets which may suffer some release jitter due to the preemption of a sub-task that precedes some parallel region. Besides the presented solutions, no other solution is envisioned yet and to the best of our knowledge this is still a limitation in the existing literature.

Having a solution for this problem is important for the community, as it allows one to employ parallel real-time tasks in more dynamic scenarios where applications (e.g., multimedia applications) still need to provide timing guarantees. At the same time, with a solution for admission control, applications are able to take advantage of the multiprocessor platform in order to improve their efficiency as well as the utilization of the platform.

Other general questions may arise in the study of this problem, such as if it is possible to perform online admission of parallel tasks without knowing their internal structure. Nevertheless, such questions may have an answer after finding a good initial solution for this problem.

Acknowledgements

This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and when applicable, co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER Research Centre); also by FCT/MEC and ERDF through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project(s) FCOMP-01-0124-FEDER-020447 (REGAIN); by the European Union, under the Seventh Framework Programme (FP7/2007-2013), grant agreement nr. 611016 (P-SOCRATES); also by FCT/MEC and the ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH / BD / 88834 / 2012.

References

[1] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho, "Response-time analysis of synchronous parallel tasks in multiprocessor systems," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14. ACM, 2014, pp. 3–12.

[2] K. Lakshmanan, S. Kato, and R. R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, ser. RTSS '10. IEEE Computer Society, 2010, pp. 259–268.

[3] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, Nov 2011, pp. 217–226.

[4] J. Li, K. Agrawal, C. Lu, and C. Gill, "Analysis of global edf for parallel tasks," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, July 2013, pp. 3–13.

[5] N. Fisher, S. Baruah, and T. Baker, "The partitioned scheduling of sporadic tasks according to static-priorities," in *Real-Time Systems, 2006. 18th Euromicro Conference on*, 2006, pp. 10 pp.–127.

[6] A. Masrur, S. Chakraborty, and G. Farber, "Constant-time admission control for partitioned edf," in *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, July 2010, pp. 34–43.

Towards Energy and Feasibility Optimal Scheduling on big.LITTLE platforms

Hoon Sung Chwa*, Jaebaek Seo*, Hyuck Yoo* Jinkyu Lee†, Insik Shin*
*School of Computing, KAIST, Republic of Korea
†Department of Computer Science and Engineering, Sungkyunkwan University, Republic of Korea
insik.shin@cs.kaist.ac.kr

Extended Abstract

Advances on chip architectures towards multi-cores have provided two general options for real-time scheduling: global scheduling that allows a task to migrate from one core to another, and partitioned scheduling that disallows the migration. To explore their own advantages such as full utilization of the former and small overhead of the latter, real-time multi-core scheduling has been widely studied, mostly for homogeneous cores [1]. When it comes to heterogeneous multi-core scheduling, very few studies have focused on global scheduling mainly due to insufficient architectural support for migration in commercial heterogeneous multi-core chips.

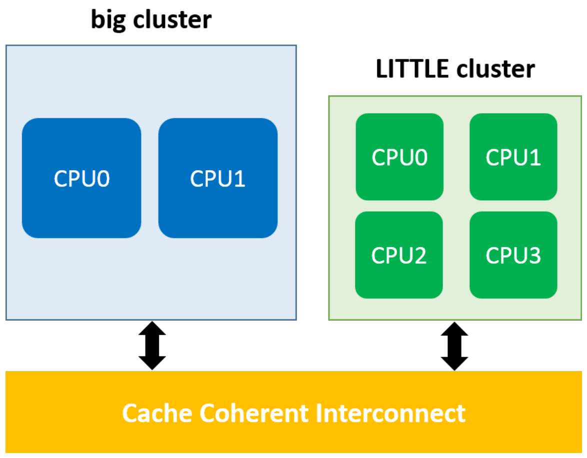


Fig. 1. ARM big.LITTLE heterogeneous multi-core chip

Recently, ARM has launched a two-type heterogeneous multi-core chip, called big.LITTLE [2], which has been deployed in the state-of-the-art smartphones, e.g., Samsung Galaxy S6 and Note 4. The big.LITTLE architecture consists of two heterogeneous clusters: one with high-performance "big" cores and the other with power-efficient "LITTLE" cores (see Fig. 1). One of the most distinguishable features of the big.LITTLE architecture is a practical support for migration; cores in two different types (big and LITTLE) not only deploy the same instruction-set architecture, but also share a specially designed interconnection bus for data transfer between the clusters. Through coupling big and LITTLE cores, the big.LITTLE architecture is capable of global scheduling to achieve high-performance with maximum energy efficiency. In real-time systems research, however, almost all energy-aware scheduling approaches for heterogeneous multi-core platforms have been focused on partitioned scheduling rather global scheduling [3], [4], [5], [6]. It has been shown that the problem of task-partitioning to minimize energy consumption is NP-hard [7]. Thus, they focused on developing efficient heuristic algorithms with approximation bounds. From the feasibility point of view, the scheduling problem on heterogeneous multiprocessors has been studied in the past [8], [9], [10], [11]. While most of them consider the task-partitioning problem, the only known study [11] addressed the heterogeneous multicore feasibility problem that determines whether a given task set can be executed on heterogeneous multi-cores under global scheduling while all timing constraints are met. However, the study did not consider energy minimization issues.

Motivated by the cutting-edge heterogeneous multi-core architecture, we focus on global scheduling, and demonstrate "how good global scheduling is for a big.LITTLE platform" compared to existing partitioned scheduling approaches from the both core utilization and energy consumption points of view. To this end, we would like to achieve the following goals:

G1. *Feasibility-optimality*—our solution schedules all jobs in a task set without any deadline miss of a job, as long as there exists such a feasible solution; and

G2. *Energy-optimality*—any feasible solution cannot yield less energy consumption than our solution.

To achieve the goals, we need to determine the following cluster configurations and job schedules:

D1. On-and-off status of each cluster;

D2. Voltage/frequency level of each cluster; and

D3. Schedule of each job, i.e., the time intervals in and the core on which each job executes.

For D1, we have three options: (a) both big and LITTLE clusters are turned on; (b) only the big cluster is turned on; and (c) only the LITTLE cluster is turned on. Note that we do not consider a core-level on-and-off policy, because it has been demonstrated that we hardly benefit from the core-level on-and-off policy in terms of energy saving [12]. Thereby, we assume that all cores are activated in a cluster when the cluster is turned on.

When it comes to D2, the big.LITTLE architecture provides dynamic voltage and frequency scaling (DVFS) per cluster. The big (likewise LITTLE) cluster provides nine (likewise five) discrete frequency/voltage levels [13]. We note that the big.LITTLE architecture supports only cluster-level DVFS, meaning that we can apply different voltage/frequency to the big and LITTLE clusters, but cores in the same cluster operate with the same frequency/voltage. Thus, we have cluster-level discrete choices. We consider static (rather than dynamic) voltage/frequency scaling on which the operating voltage/frequency does not change over time. In the previous literatures [14], [7], the energy optimal frequency is a constant when each job presents its worst-case workload behavior with the convex power consumption function. Therefore, the static scaling not only is simpler, but also minimizes the worst-case energy consumption (when each job presents its worst-case execution time). Such a worst-case behavior of energy consumption is important for mobile, battery-powered devices in which a big.LITTLE processor is deployed.

For D3, it is too complex to determine all the job schedules during the entire time interval of interest, because we should decide not only "when", but also "where" to execute a job, which yields different speed of execution and energy consumption. Therefore, we abstract the job schedule as the fractions of workload of its invoking task that are assigned to big and LITTLE clusters. This is because, the fraction of workload of a task assigned to each cluster not only indicates the amount of execution of the task on each cluster, but also determines the duration for a core to be on the active state due to the task's execution, which can be translated into the energy consumption. Using the fraction of workload of each task on each cluster, we divide the problem into the following two steps. At step 1, we determine how much portion of a task workload will be executed on big and LITTLE clusters. Then, at step 2, we develop a global scheduling algorithm that generates job schedules for given workload allocation on big and LITTLE clusters, determined by step 1. One of the major concerns on global scheduling is migration overhead. Hence, we need to incorporate migration overhead into developing scheduling algorithms.

In summary, this paper is motivated by an attempt to see how good global scheduling, beyond partitioned scheduling, can be for big.LITTLE platforms (one of the cutting-edge heterogeneous multi-core architectures) in the perspective of both core utilization and energy saving. To this end, we examine the problem of determining big.LITTLE platform configurations and global job schedules, so that the energy consumption is minimized without compromising feasibility.

REFERENCES

[1] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, vol. 43, pp. 35–44, 2011.

[2] ARM, "big.little technology: The future of mobile," 2013. [Online]. Available: http://www.arm.com/files/pdf/big-LITTLE-Technology-the-Futue-of-Mobile.pdf

[3] Y. Yu and V. K. Prasanna, "Resource allocation for independent real-time tasks in heterogeneous systems for energy minimization," *Journal of Information Science and Engineering*, vol. 19(3.

[4] J.-J. Chen and L. Thiele, "Energy-efficient task partition for periodic real-time tasks on platforms with dual processing elements," in *Proceedings of the 14th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2008)*, 2008.

[5] C.-Y. Yang, J.-J. Chen, T.-W. Kuo, and L. Thiele, "An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems," in *Proceedings of the Conference and Exhibition of Design, Automation, and Test in Europe (DATE 2009)*, 2009.

[6] J.-J. Chen, A. Schranzhofer, and L. Thiele, "Energy minimization for periodic real-time tasks on heterogeneous processing units," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2009)*, 2009.

[7] H. Aydin and Q. Yang, "Energy-aware partitioning for multiprocessor real-time systems," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2003)*, 2003.

[8] S. Baruah, "Task partitioning upon heterogeneous multiprocessor platforms," in *RTAS*, 2004.

[9] B. Andersson, G. Raravi, and K. Bletsas, "Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors," in *RTSS*, 2010.

[10] G. Raravi, B. Andersson, K. Bletsas, and V. Nelis, "Task assignment algorithms for two-type heterogeneous multiprocessors," in *ECRTS*, 2012.

[11] S. Baruah, "Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms," in *RTSS*, 2004.

[12] A. Carroll and G. Heiser, "Unifying DVFS and offlining in mobile multicores," in *RTAS*, 2014.

[13] Hardkernel co. Ltd., "ODROID-XU+E specification," 2014. [Online]. Available: http://odroid.com/dokuwiki/doku.php?id=en:odroid-xu

[14] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez, "Dynamic and aggressive scheduling techniques for power-aware real-time systems," in *Proceedings of IEEE Real-Time Systems Symposium*, 2001.

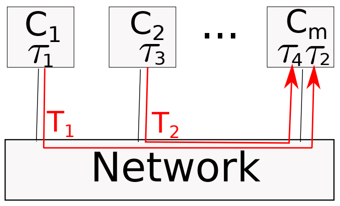
Multi-Periodic Time-Triggered Scheduling for Safety-Critical Systems

Anna Minaeva Benny Akesson Zdeněk Hanzálek
Czech Technical University in Prague

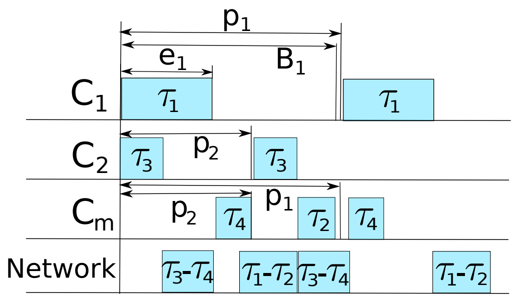
I. Motivation and Problem Formulation

The problem of scheduling safety-critical systems with multiple cores that have small local memory and communicate through a network, e.g. FlexRay, can be found in automotive, avionics and other industries. Periodic tasks, such as sensing and actuation, execute on these cores and need to communicate with each other. Tasks have hard real-time requirements that must be always satisfied to guarantee safe operation. Moreover, due to the nature of these systems, tasks have different periods, since control tasks release jobs with different frequencies. The time-triggered approach is commonly used in safety-critical systems due to highly predictable behaviour of the scheduled tasks, resulting in simplicity of both verification if a given solution satisfies all constraints and debugging when something goes wrong.

We consider the problem of *multi-periodic non-preemptive scheduling of tasks with precedence constraints on multiple resources*. The problem is shown in Figure 1a, where there are m cores $C_1, C_2, \cdots, C_m$ that communicate via a network. We suppose that there are several *transactions* T_k in the form $C_i \rightarrow Network \rightarrow C_j$, where first some task must be executed on core C_i, then there is communication via the network, followed by execution of a task τ_i with processing time e_i on core $C_j \neq C_i$. We can formalize each transaction as a chain of tasks $\tau_i \rightarrow$ Network $\rightarrow \tau_{i+1}$, released at the beginning of each period with implicit deadlines $D_i = p_i$, where each task τ_j, $j = 1, 2, \cdots, n$ is assigned to exactly one core where it must be executed. In the example in Figure 1a, there are two transactions $T_1 = \tau_1 \rightarrow$ Network $\rightarrow \tau_2$ and $T_2 = \tau_3 \rightarrow$ Network $\rightarrow \tau_4$. Each transaction T_k may have a different period p_k and a required maximum end-to-end delay, B_k, i.e. the maximum time from the beginning of the first task in a transaction to the end of the second and final task in the transaction is constrained by the user. Thus, the problem is to find a time-triggered schedule for the tasks on the cores and communication on the network such that the end-to-end delay of each transaction satisfies the requirement.



(a) Problem description



(b) Example schedule

Fig. 1: Multi-periodic scheduling problem description with examples of transactions and solution

A Gantt chart of a possible solution to the problem with transactions T_1 and T_2 from Figure 1a is shown in Figure 1b. Here, tasks $\tau_1, \tau_2, \tau_3, \tau_4$ and their communication are scheduled on cores C_1, C_2, C_m and on the network. We can see that the periods of tasks and their processing times are different and all tasks and transactions satisfy their deadlines and end-to-end delay requirements. The solution to our problem is defined as a set of start times and durations (equal to the processing time of each job) during a scheduling interval, called the hyperperiod. The hyperperiod is a least common multiple of all the periods of the transactions.

Applying the stated problem on a real system brings additional constraints on the solution. First of all, the small local memories of embedded systems can only store schedules of limited length, i.e. the hyperperiod of the final schedule may not exceed a size L defined by the user. Moreover, the size of an average problem is in the range of *thousands of transactions*, which requires the approach to be scalable, since it has to be possible to find a schedule to the problem in reasonable time. Thus, an ILP approach does not seem to be a good final solution due to the large number of required decision variables, resulting in very long computation time. However, some advanced exact optimization methods or heuristic approaches are good candidates.

Problems similar to the presented problem were considered in different formulations before. In [1], the authors solved similar problem with mono-periodic transactions, i.e. $p_1 = p_2 = \cdots = p_N$. Although it is possible to schedule all transactions with the minimum possible period also in our problem, it results in large over-allocation and restricts the problem instances schedulable by the approach. Over-allocation here means that more time is reserved in the schedule than the tasks can consume. In [2], [3], [4] and [5] the authors require perfectly periodic scheduling, where the whole schedule for all jobs from a task is completely defined solely by the starting time in the first period. This results in over-allocation and significantly restricts the set of solvable instances as well. The main difference of our problem with [2] is that the athours do not put any constraints on end-to-end delays of transactions. Works [3], [4] and [5] consider the problem of multi-hop networks scheduling, which is a generalization of our problem as it allows complicated network structure instead of a single communication channel in our case and it adds some network constraints. Moreover, the authors in [3] target their solution to the problem with harmonic periods (i.e. for each pair of periods $p_i \geq p_j$ holds $p_i|p_j$). The authors in [6] consider a similar problem without precedence constraints and with periods that are determined by the hardware to be $m \cdot 2^k$, where $m \in \mathbf{N}$ is a constant and $k \in \mathbf{N}$ can take any integer value. Their solution cannot be used for the described problem due to the absence of precedence constraints between tasks.

II. Open Problems

The problems mentioned in Section I influence both the considered problem and the applied solution: 1) bounded storage capacity, 2) the requirement not to over-allocate, 3) the requirement to be able to solve as many real-life problems as possible, and 4) the scalability issue. The formulation of the problem is equally important as the method chosen to solve it. The reason is that the problem, being formulated in a too constrained manner, can suffer unreasonably high rate of problem instances with no possible solution. For instance, restricting ourselves to perfectly periodic allocation has following advantages: a straight-forward analysis and the fact that the amount of memory required to save the obtained schedule grows linearly with the number of transactions in contrast to the general case, where relatively prime periods can result in a very large hyperperiod. Moreover, the problem of over-allocation is automatically solved as well, since it is not possible to over-allocate with perfectly periodic allocation, considering initial periods. However, it seems that requiring perfectly periodic allocation on the initial problem is too restrictive.

Another question is "is it possible to slightly change the problem in order to have almost all the advantages of perfectly periodic allocation at a lower cost in terms of number of unschedulable problem instances?". By converting the initial problem to a problem with harmonic periods similarly to [7], it is possible to schedule the tasks in a simple way with significantly reduced length of the schedule. For instance, having a problem instance with three transactions with periods $p_1 = 2$, $p_2 = 5$ and $p_3 = 16$ it is possible to convert it to the problem, where $p_2 = 4$ with the rest of the periods unchanged. Then, instead of having the schedule length of $lcm(2, 5, 16) = 80$ we have $lcm(2, 4, 16) = 16$, significantly saving the local memory space, but resulting in over-allocation since now we schedule the tasks in T_2 every 4 time units instead. Furthermore, scheduling becomes easier, since there is less collisions in the schedule when allocating tasks perfectly periodically. However, this solution can cause significant over-allocation, resulting in impossibility to solve instances with high load.

Ideally, it is required not just to solve the satisfiability problem, but also care about the utilization of the final schedule. Typically, there are not only periodic tasks in the system, but also sporadic tasks that are required to be scheduled in an event-triggered manner. It means that once in some relatively short interval there should be a gap in the schedule that we want to maximise for sporadic tasks to be efficiently scheduled.

Acknowledgement

This work was supported by the Grant Agency of the Czech Republic under the Project GACR P103/12/1994, the Ministry of Education of the Czech Republic under project number CZ.1.07/2.3.00/30.0034, and Eaton European Innovation Centre.

References

[1] Z. Hanzalek, P. Burget, and P. Sucha, "Profinet io irt message scheduling with temporal constraints," *Industrial Informatics, IEEE Transactions on*, vol. 6, no. 3, pp. 369–380, Aug 2010.

[2] L. Cucu and Y. Sorel, "Non-preemptive multiprocessor scheduling for strict periodic systems with precedence constraints," *Proceedings of 23rd Annual Workshop of the UK Planning and Scheduling Special Interest Group, PLANSIG'04*, 2004.

[3] W. Steiner, "An evaluation of smt-based schedule synthesis for time-triggered multi-hop networks," *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, vol. 31, pp. 375–384, November 2010.

[4] S. S. Craciunas and R. S. Oliver, "Smt-based task- and network-level static schedule generation for time-triggered networked systems," in *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14. New York, NY, USA: ACM, 2014, pp. 45:45–45:54. [Online]. Available: http://doi.acm.org/10.1145/2659787.2659812

[5] A. Biewer, B. Andres, J. Gladigau, T. Schaub, and C. Haubelt, "A symbolic system synthesis approach for hard real-time systems based on coordinated smt-solving," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 357–362. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755753.2755834

[6] J. Dvorak and Z. Hanzalek, "Multi-variant time constrained flexray static segment scheduling," *Factory Communication Systems (WFCS), 2014 10th IEEE Workshop on*, pp. 1–8, 2014.

[7] K.-J. Lin and A. Herkert, "Jitter control in time-triggered systems," *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on*, vol. 1, pp. 451–459, 1996.

How do we prove that probabilistic worst case response time is a Gumbel?

Adriana Gogonel and Liliana Cucu-Grosjean
INRIA Paris-Rocquencourt, France*

1 Introduction

A defibrillator, a mobile phone and an airplane, they all share the same property of being embedded computing systems. Beside constraints like power consumption, size and weight, some embedded systems may have time constraints to meet and those systems are called time critical embedded systems. The design of time critical embedded systems is mainly based on commercial processors with an excellent average time behavior. Unfortunately an excellent average time behavior processor ensures safe average timing behavior (execution time of a task on the processor for instance), but it might implies important worst case deviation. Commercial processors with excellent average case performance (in terms of the time taken to execute code) may have a very large deviation (orders of magnitude) from this performance in the worst-case. During the last twenty years different solutions have been proposed to time critical system designers through a pessimistic estimation of performances of the processors (thus increased costs) while using average time behavior processors.

The probabilistic description of some parameters of existing models is a possible solution to the problem of designing time critical embedded systems. A probabilistic description provides richer information to the designer from one phase of conception to another one. A probabilistic bound on all execution time probability distributions of a task may define such probabilistic description. At a higher level the worst case response time of that task becomes its probabilistic worst case response time (pWCRT)[1].

2 An open problem

The pWCRT of a task may be obtained either analytically [2] or by using measurement-based approaches [1]. The measurement-based approaches have indicated that the pWCRT converges to the Gumbel probability distribution function (pdf) [1]. The Gumbel family together with Fréchet and Weibull are the three possible pdfs that may upper bound the maximum of a set of random variables (see Figure 1).

For a given set of tasks and a given processor one may show by comparing for instance the pWCET obtained in [2] against the pWCRT obtained in [1] that the pWCRT belongs to Gumbel pdfs. But **how do we prove that pWCRT is a Gumbel for any task and any processor ?**

*firstname.lastname@inria.fr

[1] An interested reader may find more detailed definitions of these notions in [2].

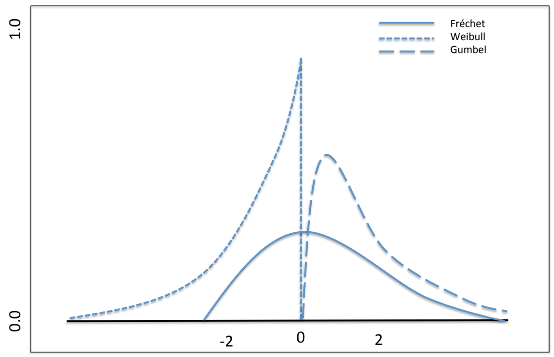


Figure 1: Three possible upper bounds for the maximum of a set of random variables

3 A possible proof

The design of a time critical embedded system may have basically three main phases: (i) the description of the physical process that should be controlled (control theory), (ii) the description of the functional requirements that should be fulfilled (synchronous and asynchonous models) and (iii) the description of the implementation of the time critical embedded system (scheduling or verification).

A proof for our open problem could be based on the probabilistic description of parameters at different levels of design of a time critical embedded system:

- *Probabilistic approaches for control theory for mixed criticality systems*. Solving a control system problem consists in finding the sampling frequency and we identify it as the first property to be described probabilistically.
- *Probabilistic approaches for synchronous models for mixed criticality systems*. The transition between states might be the first property to be described probabilistically by relaxing the synchrony hypothesis.
- *Probabilistic approaches for asynchronous models taking into account mixed criticality systems*. Here the transition between states may be the first to be described probabilistically.
- *Probabilistic approaches for real-time scheduling analysis for mixed criticality systems*.
- *Probabilistic approaches for model checking for mixed criticality systems*. The integration of rare events probability distributions in current probabilistic model checking seems to be the first reasonable step.

References

[1] Y. Lu, T. Nolte, I. Bate, and L. Cucu-Grosjean. A statistical response-time analysis of real-time embedded systems. In *the 33rd IEEE Real-Time Systems Symposium (RTSS)*, pages 351–362, 2012.

[2] D. Maxim and L. Cucu-Grosjean. Response time analysis for fixed-priority tasks with multiple probabilistic parameters. In *the 34th IEEE Real-Time Systems Symposium (RTSS)*, pages 224–235, 2013.

Open Problems on Non-preemptive Scheduling of Mixed-Criticality Real-Time Systems

Mitra Nasri and Gerhard Fohler
Chair of Real-time Systems, Technische Universität Kaiserslautern, Germany
{nasri,fohler}@eit.uni-kl.de

I. Introduction

In many industrial systems, design complexity is partly reduced by the use of *non-preemptive* message passing or execution. Doing so, context switch overheads are diminished and resource sharing mechanisms are simplified. However, in a mixed-criticality system, long blocking time caused by non-preemptive execution of a task which has lower priority than some other high criticality (HC) tasks might threaten schedulability of the HC tasks in their HC mode. In single criticality systems, it has been proven that non-preemptive scheduling is an NP-Hard problem for periodic [1] or harmonic tasks [2], where each period is an integer multiple of shorter periods. In this work, we want to extend the existing solutions for single criticality systems to two criticality systems to solve the scheduling problem of the Vestal task model for mixed criticality systems [3].

Recently, an online non-preemptive non-work-conserving scheduling algorithm called *Precautious-RM* has been introduced in [4]. This algorithm prioritizes the jobs according to rate monotonic, however, it schedules a job only if its execution will not cause a deadline miss for the next instance of the task with the smallest period. Otherwise, it schedules an idle interval until the next release of the highest priority task. It has $O(n)$ computational complexity and it guarantees the schedulability of harmonic tasks in three cases: a) period ratio is 2, b) period ratio is greater than 2, and c) tasks have arbitrary period ratio, yet there are enough *vacant intervals*. A vacant interval is defined as the slack of two consecutive instances of the task with the smallest period. Period ratio is defined as the ratio of two consecutive periods, i.e., T_i/T_{i-1} if the periods are sorted in ascending order. In the third case, suggested conditions are only sufficient since the general problem is NP-Hard, i.e., system is schedulable by Precautious-RM if each task has its guaranteed slot before its deadline.

II. Open Problems

At this stage, we limit the solution to uni-processor systems and harmonic task sets. We assume tasks are indexed according to their period so that $T_1 \leq T_2 \leq \ldots \leq T_n$ where T_i is the period of task τ_i and n is the number of tasks. We consider a dual criticality task set with HC and low criticality (LC) tasks which are independent from each other. To guarantee the schedulility of the tasks, we use Precautious-RM as the underlying scheduling algorithm. In this section, we investigate schedulability challenges in two cases; a) all tasks are non-preemptive, b) HC tasks are non-preemptive while LC tasks can be preempted.

A. Schedulability Challenges for Fully Non-preemptive Task Sets with Two Criticality Levels

Although Precautious-RM guarantees the schedulability of harmonic tasks in some cases, it cannot be easily applied in mixed-criticality systems because it only schedules a low priority task, i.e., a task with larger period, if this task will not cause a deadline miss for the next instant of the task with the smallest period in the task set, i.e., τ_1. As a result, the following two challenges must be solved to be able to use Precautious-RM with mixed-criticality task sets:

- Precautious-RM does not distinguish between different execution times of a task, hence, it may schedule an HC task which is not yet switched to its HC mode without paying attention to possible mode changes during the execution of this task. As a result, it may cause a long blocking time for other HC tasks because of a late mode change. If those HC tasks have earlier deadlines than the finish time of the executing HC task, their schedulability will be endangered. We call it the *long blocking problem*.
- Before scheduling a low priority task, Precautious-RM checks if this task causes a deadline miss for the next instance of the task with the smallest period. If it does, this task will not be scheduled, instead, an idle interval will be scheduled until the next release of the task with the smallest period. However, in mixed-criticality systems, the task with the smallest period might be a low criticality task which can be ignored after switching the mode to HC. As a result, the schedulability condition must be revised according to the occurrence of a mode change during the execution of a task.

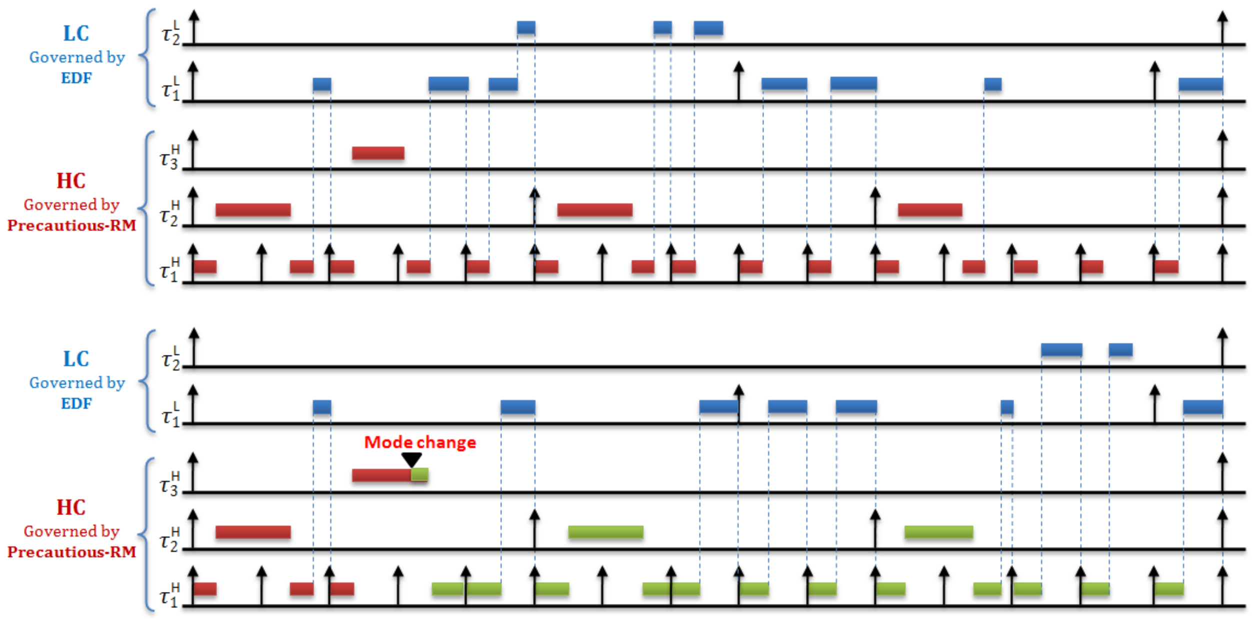


Fig. 1. An example for a hierarchical scheduling to support non-preemptive execution of HC tasks and preemptive execution of LC tasks

B. Schedulability Challenges for Non-preemptive HC Tasks Scheduled Together with Preemptive LC Tasks

If the system is non-preemptive, schedulability conditions become conservative because of the fact that in a feasible non-preemptive schedule, the maximum execution time of each task cannot be larger than two times the slack of the task with the smallest period [2], [4]. For the low criticality tasks, the precision in the WCET is not as important as HC tasks. Moreover, in some cases, they have larger execution time than HC tasks and cannot fit into the mentioned limited slack. For example, to keep an unmanned aerial vehicle in the air, usually control tasks with small execution times and periods are used, while, for example, the navigation task has larger execution time and period [5]. In other words, there is usually a difference between the time granularity of the required responses from LC and HC tasks.

To support mixed execution of preemptive LC tasks with non-preemptive HC tasks, we can use a hierarchical scheduling approach with two levels. The first level is governed by Precautious-RM and is responsible for HC tasks while the second level handles LC tasks using EDF within the idle intervals produced by Precautious-RM. This hierarchy separates interference between LC and HC tasks. Fig. 1 shows an example of this hierarchy. In this example there are 3 high criticality tasks which are scheduled by Precautious-RM. Assume that the challenges in Sect. II-A are solved and we can come up with a feasible approach to schedule non-preemptive HC tasks. At the times when Precautious-RM schedules an idle interval, we can use EDF to schedule low critical tasks. In this example, period of the low criticality tasks is an integer multiple of period of τ_1^H which is the first HC task with the smallest period.

The open problem in this setup is how to guarantee schedulability of some of the low priority tasks at design-time. As shown in Fig. 1, LC tasks can be executed within the structured schedule which is produced by Precautious-RM. It is called *structured*, because in the worst-case, each low priority HC task is scheduled between two consecutive instances of τ_1^H.

The remaining challenge is that if HC tasks switch to HC mode, like the example in the bottom of Fig. 1, the slots which are available for LC tasks will change. The consequence of such changes must be considered in the schedulability condition of low critical tasks.

References

[1] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *IEEE Real-Time Systems Symposium (RTSS)*, 1991, pp. 129–139.

[2] Y. Cai and M. C. Kong, "Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems," *Algorithmica*, vol. 15, no. 6, pp. 572–599, 1996.

[3] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *IEEE Real-Time Systems Symposium (RTSS)*, 2007, pp. 239–243.

[4] M. Nasri and M. Kargahi, "Precautious-RM: a predictable non-preemptive scheduling algorithm for harmonic tasks," *Real-Time Systems*, vol. 50, no. 4, pp. 548–584, 2014.

[5] J.-H. Kim, S. Sukkarieh, and S. Wishart, "Real-time navigation, guidance, and control of a uav using low-cost sensors," in *Field and Service Robotics*, ser. Springer Tracts in Advanced Robotics, S. Yuta, H. Asama, E. Prassler, T. Tsubouchi, and S. Thrun, Eds. Springer Berlin Heidelberg, 2006, vol. 24, pp. 299–309.

Partitioning the Network-on-Chip to Enable Virtualization on Many-Core Processors

Matthias Becker*, Dakshina Dasari†, Vincent Nélis‡, Moris Behnam*, Thomas Nolte*
*MRTC / Mälardalen University, Sweden
{matthias.becker, moris.behnam, thomas.nolte}@mdh.se
† Research and Technology Centre, Robert Bosch, India
dakshina.dasari@in.bosch.com
‡CISTER/INESC-TEC, ISEP, Portugal
nelis@isep.ipp.pt

I. Introduction

Technological advances have increased the transistor density, thereby ushering in multi- and more recently many-core systems, distinguished by the presence of hundreds of cores on a single chip. For such a platform, the Network-on-Chip (NoC) has emerged as a scalable and efficient interconnect fabric to realize the communication across an ever increasing number of processor cores, memories, and specialized IP blocks both on- and off-chip. This shift in the interconnect mechanism has overcome the performance barriers of traditional shared buses that do not scale well and cross-bars that consume a lot of chip resources. Given the computational capabilities that this platform offers, several applications, with possibly different criticality/safety requirements, can be consolidated on one platform. System designers have been considering various mechanisms to harness these new platforms; one among them being virtualization. Virtualization techniques [1] are commonly used to partition one platform into several, independent, Virtual Machines (VM), with each partition given its own set of physical resources. While virtualization techniques are mature in single core systems, the clear differences in the processor architectures of many-cores compared to their precedents requires a different approach: (i) The large number of simple cores diminishes the need of partitioning on core level, i.e. a subset of cores will be allocated to one partition/application. (ii) The bandwidth on the different links of the NoC, on the other hand, must be partitioned.

a) Service Guarantees on the NoC: The NoC implemented in today's many-core processors generally provides services on a best-effort basis. With the need for guaranteed services, traffic regulation is proposed for flows with real-time constraints. In order to decrease the hardware complexity, traffic shaping is not implemented within the NoC routers. It is rather implemented in the source nodes, and thus limits the injection of messages into the NoC. As an example, Kalray's MPPA 256 processor provides a hardware based traffic shaper which allows to bound communication times using the (σ, ρ) network calculus [2]. Munk et al. present a software based traffic shaper [3] to allow for guaranteed communication services on many-core processors without hardware support.

In general, such a traffic shaper is configured with the parameters, T_w (the time window) and N_{max} (max. packets). According to [2], the translation to the (σ, ρ) parameters is done by $\rho = N_{max}/T_w$ and $\sigma = N_{max}(1 - N_{max}/T_w)$, as shown in Fig. 1. Once a message is scheduled for transmission, the traffic shaper checks if the number of packets transmitted during the last T_w cycles plus the number of packets of the scheduled message is less than N_{max}. If this is the case the message is sent. Otherwise the traffic shaper holds the message until the condition is true.

II. Challenges

In this section, we discuss the challenges in deploying applications onto the flow regulated NoC described above. From the application point of view, the challenge for the hypervisor is to configure the network flow limits across the NoC in such a way that all the application requirements are met, especially so when there is diversity in the requirements (ranging from soft or hard timing constraints) and also with varying traffic injection patterns. Though the ability to configure the flow regulation parameters is a useful feature, setting inappropriate values can have a negative impact on the performance. On the extremities, it could either lead to over-provisioning of the bandwidth, resulting in an under-utilized platform or it could lead to under-provisioning, resulting in severe degradation of the application performance (and possibly a violation of the timing constraints). Thus a network-wide default setting of the flow-limits may not satisfy the requirements of all the hosted applications. Additionally, it needs to be considered that flow regulation is done only at the source level. Flows, however, might have different paths, periods, and message sizes.

Designing the network virtualization layer to implement the above can be complex: Firstly it has to provide the basic functionality which is to abstract the physical network (links, routers, buffers, routing algorithms, etc.) and move it into the virtualization software layer, making it totally transparent to the user. Secondly, it may be necessary to provide the required resource reservations on the network for applications with higher priority, either by reserving fixed links and associated buffers on the network (like virtual circuits), by setting flow regulations as described above, or configuring rules in the routers to enforce this. All this can be challenging if limited configuration options are provided or it is required to implement it dynamically in the case of changing traffic loads. Additionally, in order to provide spatial isolation to safeguard some critical applications hosted

The work presented in this paper was partially supported by the Swedish Knowledge Foundation via the research project PREMISE and by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER Research Centre); by FCT/MEC and the EU ARTEMIS JU within project ARTEMIS/0001/2013 - JU grant nr. 621429 (EMC2); by the North Portugal Regional Operational Programme (ON.2 O Novo Norte) under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), and by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology), within project NORTE-07-0124-FEDER-000063 (BEST-CASE, New Frontiers).

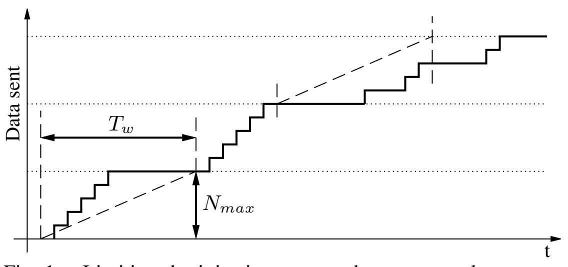
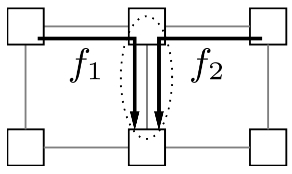
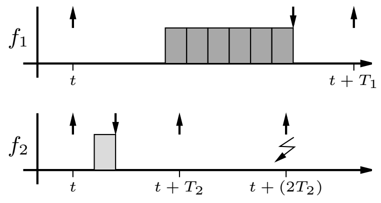


Fig. 1. Limiting the injection rate on the source nodes.



(a) Paths of two flows on a NoC as result of task mapping.



(b) Possible traffic pattern on the highlighted channel.

Fig. 2. Example of unschedulable messages as result of the task mapping.

on particular (safe) guest zones, the hypervisor may also have to restrict traffic from insecure sources from reaching the safe zones. Given the context above, in the next section we enlist specific open problems that must be addressed.

III. Discussion and Open Problems

Having traffic regulation on the source nodes enables guaranteed services for the messages on the NoC. However, as discussed above, the selection of the flow regulation parameters is not trivial. One solution could be to configure individual routers with different flow limits, but this can be pretty challenging considering the scale and it also clearly needs a fixed application mapping as a pre-requisite, in order to arrive at an optimal configuration. An uninformed mapping on the other hand may make it infeasible for the hypervisor to arrive at a network-wide optimal solution. This is for example the case in Fig. 2. The task placement leads to the message paths of the periodic messages, f_1 and f_2, with implicit deadlines (Fig 2(a)). Fig. 2(b) depicts an exemplary worst-case scenario on the channel which is shared by f_1 and f_2. Since the period T_2 is less than the time it takes to transmit a whole message of f_1 across one channel, the configuration is not schedulable. Because the best effort NoCs do not allow preemptions, traffic regulation can not be used to arrive at a schedulable system.

Software based traffic shaping at flow level, in tandem with the traffic shaping of the outgoing link could be applied in order to guarantee safe operation of each flow on the NoC. This way, the traffic shaping at flow level is used to limit the bandwidth of the flow, i.e. protect other parts of the system against malfunctioning. The second level can then be used to shape the outgoing traffic. Having such a configuration has many similarities to hierarchical scheduling. The period T_w of a traffic shaper could for example be selected similar to the techniques presented by Shin in [4].

As described above there are certain open problems which need to be addressed. Applications in the context of this paper can be modeled as a set of periodic tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task can have precedence constraints, data dependencies, and core affinity, i.e. a task can be required to execute on a certain core. We describe a task thus by the tuple $\tau = \{C_i, T_i, D_i\}$, where C_i is the Worst Case Execution Time (WCET), T_i is the task period, and D_i the deadline. Additionally, let us define the set of cores by $\pi = \{\pi_1, \pi_2, \ldots \pi_m\}$. We now define a mapping $\mathcal{M}$ from the tasks to cores given by $\mathcal{M} : \Gamma \mapsto \pi$. We also define a network wide flow configuration $\mathcal{W} : (\sigma, \rho)$ at each of the routers in the set $\{\mathcal{R}\}$ given by $\mathcal{W} \mapsto \mathcal{R}$.

1) Given a network with a fixed flow-limit configuration $\mathcal{W} \mapsto \mathcal{R}$ and a mapping $\mathcal{M} : \Gamma \mapsto \pi$, with given traffic characteristics and timing requirements, determine if all the requirements can be satisfied.
2) Given a mapping $\mathcal{M} : \Gamma \mapsto \pi$, find a suitable flow-limit configuration $\mathcal{W} \mapsto \mathcal{R}$ so that the task requirements are met. An ILP based solution is provided in [2]. A low complexity solution, however, is still an open problem. A solution needs be scalable as well in order to cope with the large platforms expected in the near future.
3) Given a default flow-limit configuration $\mathcal{W} \mapsto \mathcal{R}$, find a suitable mapping $\mathcal{M} : \Gamma \mapsto \pi$ such that all the requirements of the application are met. This is the inverse case of 2). Such a scenario is important in dynamic systems. Applications need to be added during runtime, e.g. other applications should not be influenced. Additionally a reconfiguration of all traffic regulators during runtime is connected with increased overheard.
4) Given no flow-limit configuration, find a suitable task mapping $\mathcal{M} : \Gamma \mapsto \pi$ such that all the requirements of the application are met. This is the traditional problem of distributed systems. The regular NoC architecture and other characteristics of the platform can be exploited when focusing on many-cores. The problem complexity increases given the small and typically distributed memory close to the compute cores and the large delays when off-chip memory is accessed.

In all the above problems the additional constraints could be (i) presence of interdependent applications (ii) presence of mixed criticality applications (iii) fixed sized buffer lengths (iv) the need to form isolated islands to minimize interference to critical applications, (v) presence of heterogeneous cores.

In this paper, we highlighted some key problems in NoC based architectures that must be addressed before the deployment of real-time applications onto these platforms becomes possible. A paradigm shift from function centric to data and communication centric approaches is required. Combining hardware and software based flow-regulation seems to be the only way to ensure that NoCs go beyond the best-effort service and address the requirements of diverse applications.

References

[1] G. Heiser, "The role of virtualization in embedded systems," in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems (IIES)*, 2008, pp. 11–16.

[2] B. D. de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti, "Guaranteed services of the NoC of a manycore processor," in *Proceedings of the International Workshop on Network on Chip Architectures (NoCArc'14)*, 2014, pp. 11–16.

[3] P. Munk, M. Freier, J. Richling, and J.-J. Chen, "Dynamic guaranteed service communication on best-effort networks-on-chip," in *23rdEuromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2015, pp. 353–360.

[4] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS)*, 2003, pp. 2–13.

Scheduling in Real-time Rule-based Reasoning Systems

Ying Qiao, Hongan Wang, Chang Leng

Institute of Software, Chinese Academy of Sciences
No. 4, South Fourth Street, Zhong Guan Cun, Hai Dian District, Beijing 100190, China

qiaoying@iscas.ac.cn, hongan@iscas.ac.cn, lengchanster@gmail.com

I. Motivations

As the functions to be automated in many real-time applications such as process control, medical monitoring and pilot associate are more and more complex, the incorporation of reasoning techniques with conventional systems appears as a promising approach to deal with this complexity. However, while real-time systems require predictable and continuous operations, reasoning techniques rely on time-consuming algorithms with unpredictable or highly variable performances. This raises great difficulties that have considerably reduced the scope of traditional reasoning approaches. To attack this problem, some real-time reasoning approaches such as anytime algorithm[1], approximate processing[2,3], imprecise computation and progressive processing [4] have been proposed in recent decades. These approaches attempt to satisfy the timeliness of inference process via the trade-off between runtime and result quality. Since these approaches ignore the difference in the urgency and resource needs of individual rules, they may obtain unacceptable result quality under the serious resource-bounded scenarios and lack predication of dynamically changed resource needs of the inference process. To avoid these problems, we present a new real-time reasoning paradigm, called real-time rule-based reasoning (referred as RT-rule reasoning), in which both acceptable result quality and predictable resource needs are considered.

In RT-rule reasoning, each rule has its deadline, which is defined as the constraint imposed on the elapse time from the time the last event associated with the rule is detected to the time the action defined in the rule is completed. The goal of RT-rule reasoning is to execute correct rules and make sure every satisfied rule meets its deadline. (Here, a rule is called a satisfied rule if events defined in the IF part of this rule occur.) For this purpose, RT-rule reasoning approach should be able to predict the dynamically changed resource needs of inference process and obtain acceptable result quality even under intensive resource contention by elaborately managing the deadlines of individual rules. Thereby, rule scheduling becomes the core of the RT-rule reasoning approach.

II. Open Problems and Challenges

The rule scheduling problem in real-time rule-based reasoning system is be summarized as follows: Given a set of predefined rules, $R = \{r_1, r_2, \dots, r_n\}$. The rule scheduling problem is to find a rule scheduling method such that every rule $r_i (1 \le i \le n)$ is guaranteed to meet its relative deadline D_i, i.e, the action defined in $r_i (1 \le i \le n)$ is completed within the relative deadline D_i if $r_i (1 \le i \le n)$ is identified as a satisfied rule.

Currently, the most prevailing way to fulfill the rule-based reasoning is to exploit graph-based approaches[5-7]. These inference approaches usually represent rules as a directed acyclic graph and obtain the reasoning results via various searching policies on the rule graph. A typical graph-based inference approach is described as follows: At the beginning, the entrance node associated with the first arrival primitive event instance is activated and the iterative search starts from it. (An entrance node is a node with zero in-degree on the rule graph.) When a node is activated, it will perform a feasibility check to decide if the event represented by this node occurs. If the node passes the feasibility check, it will exploit a certain heuristic policy to select one of its parent nodes to be activated and above process will be repeated. If the feasibility check fails, the entrance node representing the primitive event associated with the next arriving event instance is activated and above process will be also repeated. During the iterative searching process, as soon as an exit node is activated, the rule associated with this exit node is identified to be a satisfied rule and is immediately executed, i.e., the action represented by this exit node is immediately executed. (An exit node is a node with zero out-degree on the rule graph.)

Based on the graph-based inference approach, an effective way to solve the rule scheduling problem is to schedule the operations of nodes on the rule graph so as to ensure the deadline of each rule during the inference process. However, following challenges should be met to achieve rule scheduling in real-time rule-based reasoning systems.

(1) How to model the graph-based inference process via tasks

Constructing task model for the graph-based inference process is the basis to fulfill rule scheduling in real-time rule-based reasoning systems. How to map the operations of nodes on a rule-graph into a set of inference tasks will affect the schedulability of rules. Basically, there are two types of mapping methods: One mapping method, called Rule-Mapping, relies on mapping the operations on the sub rule graph associated with a rule into an inference task. Another one, referred as Node-Mapping is to map the operations performed by a node on the rule graph into an inference task. We need to tradeoff the mapping costs and schedulability of rules during inference process modeling.

(2) How to estimate resource demands of an inference task set

To allow RT-rule reasoning to predict the dynamically changed resource needs of inference process, the rule scheduling should be able to estimate resource demands of an inference task set as precisely as possible. However, the inference process is a highly unpredictable process since it strongly depends on the dynamically arrival events. A node only performs its operations such as feasibility check when the result of feasibility check of its precedent node is positive. It is hard to know in advance how the nodes on the rule graph will be activated at the time when events arrive. Thus, in case of node-mapping method used in inference process modeling, whether a task is activated depends on the results of their precedent tasks. This makes the inference task set of real-time rule-based reasoning system differ from a traditional task set in which tasks have precedence relations but all tasks' activations are deterministic. Apparently, the non-deterministic activations of the inference tasks incur the difficulties in estimating resource demands of an inference task set.

III. References

[1] T. Dean and M. Boddy, "An Analysis of Time-Dependent Planning", Proceedings of AAAI, 1988, pp.49-54.

[2] V.R. Lesser, J. Pavlin, E. Durfee, "Approximate Processing in Real-time Problem Solving", AI Magazine, Vol.9, No.1, pp.49-61.

[3] F. Garvey, V. Lesser, "Design-to-time Scheduling", IEEE Transactions on Systems, Man and Cybernetics, Vol.23, No.6, pp.1491-1502.

[4] A.I. Mouaddib, F. Charpillet, J.P. Haton, "GREAT: a Model of Progressive Reasoning for Real-time Systems", Proceedings of International Conference on Tools with Artificial Intelligence, 1994, pp.521-527.

[5] J. Kang, A. Cheng, "Shortening matching time in OPS5 production systems", IEEE Transactions on Software Engineering, Vol.30, No. 7, July 2004, pp.448 – 457.

[6] A. Cheng, S. Fuji, " Self-stabilizing real-time OPS5 production systems", IEEE Transactions on Knowledge and Data Engineering, Vol. 16, No.12, December 2004, pp. 1543 – 1554.

[7] Y. Qiao, H.A. Wang, X. Li, K. Zhong , K.M. Zhang, "RTRS: A Novel Real-time Reasoning System Based on Active Rules ", ACM Applied Computing Review, Vol. 13, No. 2,2013, pp.66-76.

What is the Exact Speedup Factor for Fixed Priority Pre-emptive versus Fixed Priority Non-pre-emptive Scheduling?

Robert I. Davis[1] , Oliver Gettings[1], Abhilash Thekkilakattil[2] Radu Dobrin[2], Sasikumar Punnekkat[2]
[1]Real-Time Systems Research Group, Department of Computer Science, University of York, York, UK.
[2]Mälardalen Real-Time Research Center, Mälardalen University, Sweden.
rob.davis@york.ac.uk, og501@york.ac.uk, abhilash.thekkilakattil, radu.dobrin, sasikumar.punnekkat {@mdh.se}

Introduction

The performance of real-time scheduling algorithms can be compared in a number of different ways. *Empirical techniques* typically rely on generating a large number of task sets with parameters chosen from some appropriate distributions. The performance of the scheduling algorithms is then compared by determining task set schedulability according to exact or sometimes sufficient schedulability tests and plotting a graph of the success ratio (i.e. the proportion of task sets that are deemed schedulable) at different utilisation levels. An alternative *theoretical method* of comparing real-time scheduling algorithms is to determine the *resource augmentation* bound or *speedup factor* [7] required. This approach focuses on those task sets that are particularly difficult to schedule using one algorithm but easy to schedule using another.

The *Speedup Factor* $S(A, B)$ comparing two real-time scheduling algorithms *A* and *B* is given by the minimum factor by which the speed of the processor needs to be increased to ensure that any task set that is schedulable according to algorithm *B* is guaranteed to be schedulable by algorithm *A*. When comparison is made against an optimal algorithm (*OPT*), then $S(A, OPT)$ is referred to as the *sub-optimality* of algorithm *A*.

Combining the utilisation bounds for fixed priority pre-emptive (FP-P) and EDF pre-emptive (EDF-P) scheduling from the seminal paper of Liu and Layland [9] shows that the speedup factor *S*(*FP-P, EDF-P*) $= 1/\ln(2) \approx 1.44270$ for implicit deadline task sets. Since EDF-P is an optimal uniprocessor scheduling algorithm [5] this result also determines the sub-optimality of FP-P for implicit deadline task sets. In 2009, Davis et al. [2] derived the exact sub-optimality of FP-P for constrained-deadline task sets; *S*(*FP-P, EDF-P*) $= 1/\Omega \approx 1.76322$ (where Ω is the mathematical constant defined by $\ln(1/\Omega) = \Omega$, hence, $\Omega \approx 0.567143$). In 2009, Davis et al. [3] gave upper and lower bounds of *S*(*FP-P, EDF-P*) = 2 and $1/\Omega \approx 1.76322$ for the case of arbitrary deadline task sets. In 2010 [4] they gave upper and lower bounds for the non-pre-emptive case of *S*(*FP-NP, EDF-NP*) = 2 and $1/\Omega \approx 1.76322$ for implicit, constrained, and arbitrary deadline task sets. In 2013, Thekkilakattil et al. [10] provided an upper bound on the sub-optimality of non-pre-emptive EDF (i.e. *S*(*EDF-NP, EDF-P*) parameterised by the shortest task deadline and the longest task execution time. In 2015, this upper bound was tightened by Abugchem et al. [1].

Open Problem: Speedup Factor for FP-P v. FP-NP

EDF-P is an optimal uniprocessor scheduling algorithm [5] and so dominates EDF-NP, FP-P, and FP-NP. Further, EDF-NP dominates FP-NP [6]; however, there are no such dominance relationships between pre-emptive and non-pre-emptive fixed priority scheduling. Hence there are non-trivial speedup factors in both directions: *S*(*FP-P, FP-NP*) and *S*(*FP-NP, FP-P*). An exact speedup factor for the latter problem has recently been derived by the authors and is under review. The former problem, determining the exact speedup factor needed so any task set that is schedulable under fixed priority non-pre-emptive scheduling is guaranteed to be schedulable under fixed priority pre-emptive scheduling is to the best of our knowledge open and forms the focus of this abstract.

Upper bounds on *S*(*FP-P, FP-NP*): since EDF-P dominates FP-NP, then the exact values of $1/\ln(2) \approx 1.44270$, $1/\Omega \approx 1.76322$, and *2* for *S*(*FP-P, EDF-P*) provide upper bounds on *S*(*FP-P, FP-NP*) for implicit, constrained and arbitrary deadline task sets respectively.

Theorem 1: A *lower bound* on *S*(*FP-P, FP-NP*) for constrained or arbitrary-deadline task sets is $S = \sqrt{2}$.

Proof: Consider the following task set scheduled on a processor of speed 1.

$$\tau_A : C_A = \sqrt{2} - 1,\ D_A = 1,\ T_A = 1,$$
$$\tau_B : C_B = (2 - \sqrt{2})/2,\ D_B = \sqrt{2},\ T_B = \infty$$
$$\tau_C : C_C = (2 - \sqrt{2})/2,\ D_C = \sqrt{2},\ T_C = \infty$$

As the task set has constrained deadlines, then under FP-P scheduling, Deadline Monotonic Priority Order (DMPO) [8] is optimal. It is easy to see that the task set is only just schedulable with this priority ordering, since any increase in execution times would cause task τ_C to miss its deadline – see Figure 1. Next consider the same task set scheduled under FP-NP on a processor of lower speed: $f = \sqrt{2}/(2 - \varepsilon)$ where ε takes an infinitesimally small value. Again assume the task priorities are in DMPO. Now the sum of the execution times of task τ_A and task τ_B (or task τ_A and τ_C) can be expressed as follows:

$$2(\sqrt{2}-1)(2-\varepsilon)/2\sqrt{2}+(2-\sqrt{2})(2-\varepsilon)/2\sqrt{2}=(2-\varepsilon)/2<1 \tag{1}$$

Hence task τ_A is schedulable when blocked by either of tasks τ_B or τ_C. Further, once the first jobs of tasks τ_A and τ_B have executed, the first job of task τ_C is able to start executing before the second job of task τ_A is released. Thus the first job of task τ_C has a response time of:

$$(\sqrt{2}-1)(2-\varepsilon)/\sqrt{2}+(2-\sqrt{2})(2-\varepsilon)/\sqrt{2}=(2-\varepsilon)/\sqrt{2}<\sqrt{2} \tag{2}$$

and so meets its deadline at $\sqrt{2}$. Continuing on through the busy period, the second job of task τ_A completes at: $2+\varepsilon<2$ meeting its deadline. At this point the busy period ends. Thus we have shown that all of the tasks are schedulable. Task τ_A has a worst-case response time of $(1-\varepsilon)/2<1$ and tasks τ_B and τ_C have worst-case response times of $(2-\varepsilon)/\sqrt{2}<\sqrt{2}$. Hence the task set is schedulable at speed *f* under FP-NP scheduling. The speedup factor required such that this task set is schedulable under FP-P therefore tends to $\sqrt{2}$ as ε tends to zero □

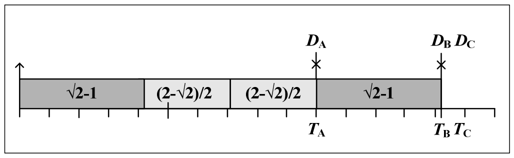


Figure 1: FP-P schedule

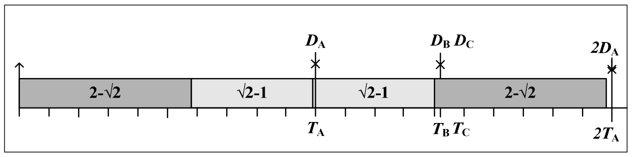


Figure 2: FP-NP schedule

Figure 3 shows the results of an empirical investigation into the speedup factor *S*(*FP-P, FP-NP*). These results were produced using a genetic algorithm which explored a wide range of different values for the task parameters (execution time, period, and deadline). The results are for 400 generations of a population of 20,000 task sets, i.e. 8 million task sets for each task set cardinality and deadline type.

For three or more tasks, then with constrained or arbitrary deadlines, the maximum speedup factor found by the genetic algorithm is very close to $\sqrt{2}\approx 1.414213562$. With implicit deadline task sets, the largest speedup factor found was somewhat lower at 1.3405. The fact that the maximum value found empirically (1. 4139) is very close to but does not exceed $\sqrt{2}\approx 1.414213562$ gives credence to the hypothesis that the theoretical lower bound (of $\sqrt{2}$) on the speedup factor is the exact value. It remains an interesting open question whether or not this is the case.

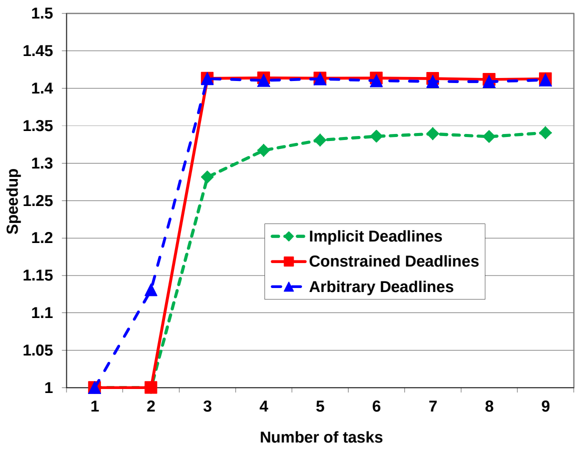


Figure 3: Empirical results for *S*(*FP-P, FP-NP*)

References

[1] Abugchem F., Short M., Xu D. A note on the sub-optimality of non-preemptive real-time scheduling. IEEE Embedded Systems Letters, 2015.

[2] Davis R.I., Rothvoß T., Baruah S.K., Burns A., "Exact Quantification of the Sub-optimality of Uniprocessor Fixed Priority Pre-emptive Scheduling." *Real-Time Systems*, Volume 43, Number 3, pages 211-258, November 2009.

[3] Davis, R.I., Rothvoß, T., Baruah, S.K., Burns, A., "Quantifying the Sub-optimality of Uniprocessor Fixed Priority Pre-emptive Scheduling for Sporadic Task sets with Arbitrary Deadlines". *In proceedings of Real-Time and Network Systems (RTNS'09)*, pages 23-31, October 26-27th, 2009.

[4] Davis R.I., George L., Courbin P., "Quantifying the Sub-optimality of Uniprocessor Fixed Priority Non-Pre-emptive Scheduling". In proceedings 18th International Conference on Real-Time and Network Systems (RTNS'10), pages 1-10, November 4-5th, 2010.

[5] Dertouzos M.L., "Control Robotics: The Procedural Control of Physical Processes". *In Proc. of the IFIP congress*, pages 807-813, 1974.

[6] George, L., Muhlethaler, P., Rivierre, N., "Optimality and Non-Preemptive Real-Time Scheduling Revisited," Rapport de Recherche RR-2516, INRIA, Le Chesnay Cedex, France, 1995.

[7] Kalyanasundaram B., Pruhs K., "Speed is as powerful as clairvoyance". *In Proceedings of the 36th Symposium on Foundations of Computer Science*, pages 214-221, 1995.

[8] Leung J.Y.-T., Whitehead J., "On the complexity of fixed-priority scheduling of periodic real-time tasks". *Performance Evaluation*, 2(4), pages 237-250, 1982.

[9] Liu C.L., Layland J.W., "Scheduling algorithms for multiprogramming in a hard-real-time environment", *Journal of the ACM*, 20(1) pages 46-61, 1973.

[10] Thekkilakattil, A.; Dobrin, R.; Punnekkat, S., "Quantifying the Sub-optimality of Non-preemptive Real-Time Scheduling," *25th Euromicro Conference on Real-Time Systems (ECRTS)*, pp.113,122, 9-12 July 2013