Overview of real-time scheduling problems

J. Goossens[1] and P. Richard[2]

[1] Université Libre de Bruxelles
e-mail: joel.goossens@ulb.ac.be
[2] Laboratoire d'Informatique Scientifique et Industrielle
ENSMA (France) e-mail: pascal.richard@ensma.fr

Keywords. real-time scheduling, periodic tasks, feasibility analysis.

1 Introduction

A computerized real-time system is required to complete its work on a timely basis. Typical applications are digital control, command and control, signal processing and communication systems. The aim of real-time scheduling is to build-up a sequence of jobs that meets hard timing constraints at run-time. In opposition with the classical scheduling literature, real-time scheduling is not necessarily on-line scheduling. The main characteristic of real-time systems is the behavioral predictability. Timing constraints will be met whatever happens in the system.

Real-time operating systems define *recurring tasks*. Real-time tasks are often periodically released. Basically, a *periodic task* gets its input data from sensors and command the controlled process by sending data to actuators. Every execution of a task is called a *job*. In the general model of periodic tasks, a task τ_i is defined by an offset O_i, that defines the release time of its first job; its worst-case execution time C_i; its period T_i between two successive releases; and its relative deadline D_i, that defines the time window in which the job has to be executed since its release time. The release time of k^{th} job of a task τ_i is $O_i + (k-1)T_i$. If the delay between two successive releases of a task is defined by a minimum delay, the tasks are said *sporadic*. An *aperiodic task* is released once and corresponds to the concept of job in the shop scheduling theory.

Timing constraints are basically set be assigning deadlines to the jobs. Restrictive models have been studied in the literature. In *implicit-deadline* systems, each task has its relative deadline equal to its period (Liu and Layland, 1973). In a *constrained-deadline* system, every task has its deadline not larger than its period. In a *arbitrary-deadline task* system, deadlines are not related to periods. A system is *synchronous*, if all tasks start simultaneously; otherwise it is *asynchronous*. In most systems, tasks can be preempted and resume later without any incurring costs (no overhead).

A periodic (resp. sporadic) task system τ is a finite collection of periodic (resp. sporadic) tasks $(\tau_1, \ldots, \tau_n)$. The utilization of a periodic or sporadic task τ_i is defined by the ratio of its execution requirement to its period: $U(\tau_i) \overset{\text{def}}{=} C_i/T_i$. The utilization of a periodic or sporadic task system τ is the sum of the utilization of all tasks in τ: $U(\tau) \overset{\text{def}}{=} \sum_{i=1}^{n} U(\tau_i)$. A schedule is *feasible* if every deadline is met.

Basic task systems have been extended in order to take into account practical factors such as non-preemption, self-suspension of tasks, precedence constraints, etc.

These results can be found in numerous textbooks dealing with real-time scheduling, among those (Liu, 2000), (Buttazzo, 1997), (Cottet et al., 2003).

2 Scheduling Environment

Tasks are executed upon a virtual machine defined by the hardware and the real-time kernel. A task is highly dependent of this *scheduling environment*. It defines the scheduling policy, the resource sharing policy, the medium access control for distributed applications, etc. We next detail some important features, while considering periodic task systems.

An *off-line schedule or time-driven schedule* assumes that all tasks have always their execution requirements equal to their worst-case execution times. At run-time, if a task is completed earlier than its worst-case execution time, then the processor is left idle until this upper limit is reached. Since tasks are periodically released, then the schedule has an infinite length. Due to the periodicity of tasks, the whole schedule is also periodic of period equals to $\mathrm{lcm}(T_1, \ldots, T_n)$. Often, for synchronous systems, the schedule enters immediately in steady state, while (often) asynchronous systems enter in steady state by time $\max_{1,\ldots,n}(O_i) + \mathrm{lcm}(T_1, \ldots, T_n)$. Consequently, the table storing the sequence of jobs (off-line schedule) has an exponential length. In practice, such a huge size can be a problem in the implementation of numerous real-time embedded systems.

Most of real-time kernels implements a *priority-driven scheduler*. Priorities can be *fixed* or *dynamic*. In fixed-priority systems, every task is assigned to a different priority level. For dynamic priority schedulers, the priority of a task is recomputed during the execution of tasks. The schedule is defined at run-time by choosing the highest priority task[s] to run among available tasks. At any time, the task having the highest priority is executed. According to the on-line scheduling literature (see Sgall (1998) and Pruhs et al. (2004)), real-time scheduling can be viewed as a on-line scheduling paradigm having only one parameter that is provided on-line: the exact execution requirements of tasks is known when they complete (we only know an upper limit of execution requirements). A real-time scheduling algorithm is *optimal* if the algorithm finds a feasible schedule if any.

In the on-line setting, scheduling anomalies have to be handled. Consider for instance tasks sharing resources in a nested way. Due to on-line priority rules, a higher priority task (τ_a) can be blocked by a lower priority task (τ_b) holding a resource. Assume that an independent task (τ_c) is released with an intermediate priority level. Then, τ_c is executed although there exists an available higher priority task. This is known as the *priority inversion phenomenon*. To limit this problem, resources are controlled by a protocol that bounds the interval of time where priority inversions occur. The behavior of the protocol is a constraint on the set of feasible schedules, but defines the maximum delay for a given task to be blocked while waiting for a resource (blocking factor). Similar problems also arise when preemption is not allowed or when tasks exchange messages through a network.

3 Scheduling Problems

In the off-line setting, there is no distinction between the design of scheduling algorithm and checking feasibility of a task system. The scheduling algorithm provides a feasible sequence of jobs where tasks are always completed by their deadlines. In the on-line setting, the real-time scheduling theory focuses on two different scheduling problems:

- *the feasibility analysis problem*: given a task system and a scheduling environment, determine whether there exists a schedule that will meet all deadlines.
- *the run-time scheduling problem*: given a task system that is known to be feasible, determine a scheduling algorithm that schedules the system to meet all deadlines, according to the supported scheduling environment.

We can distinguish three different approaches to check the feasibility/schedulability of a task system:

- *Simulation* of a scheduling algorithm until the task system is in the periodic state. This interval of time is called the *feasibility interval.* These tests can be performed if the scheduling algorithm is *robust* (i.e. can support run-time variations of execution requirements of tasks).
- *Utilization factor based analysis*: these tests are usually polynomial-time algorithms that provide sufficient schedulability conditions (Liu, Layland (1973)).
- *Time-demand based analysis*: these tests determine the worst-case workload of jobs (Time Bound Function) within an interval of time. Simple extensions of these tests allow to compute the *worst-case response times* of tasks.

4 Uniprocessor

4.1 Dynamic-priority Scheduling

(Optimal) Scheduling algorithms Dynamic-priority scheduling algorithms place no restrictions upon the manner in which priorities are assigned to individual jobs. Within the context of preemptive uniprocessor scheduling, it has been shown (Liu, Layland (1973)) that the earliest deadline first scheduling algorithm (EDF), which at each instant in time chooses for execution the currently-active job with the smallest deadline (with ties broken arbitrarily), is an *optimal* scheduling algorithm for scheduling arbitrary collections of independent real-time jobs in the following sense: If it is possible to preemptively schedule a given collection of independent jobs such that all the jobs meet their deadlines, then the EDF-generated schedule for this collection of jobs will meet all deadlines as well. Notice that EDF and the Jackson's rule (Earliest Due Date) are the very same scheduling algorithm.

Observe that EDF is a dynamic-priority scheduling algorithm. As a consequence of the optimality of EDF for preemptive uniprocessor scheduling, the run-time scheduling problem for preemptive uniprocessor dynamic-priority scheduling is essentially solved (the absence of additional constraints) — EDF is the algorithm of choice, since any feasible task system is guaranteed to be successfully scheduled using EDF. It may be noticed that there is a another optimal dynamic-priority scheduling algorithm: the *least laxity first* (LLF) defined by Mok et al. (Mok et al. (1978)). The *laxity* of a job is defined as the maximal amount of time that the job can wait and still meets its deadline. LLF gives the highest priority to the active job with the smallest laxity.

(Complexity of the) Feasibility Analysis The feasibility analysis problem, however, turns out to be somewhat less straightforward. Specifically,

- Determining whether an arbitrary periodic task system τ is feasible has been shown to be intractable — co-NP-complete in the strong sense. This intractability result holds even if the utilization $U(\tau)$ of the task system τ is known to be bounded from above by an arbitrarily small constant.
- An exponential-time feasibility test is known, which consists essentially of simulating the behavior of EDF upon the periodic task system for a sufficiently long interval.
- The special case of *implicit-deadline systems* is however tractable: a necessary and sufficient condition for any implicit-deadline system τ to be feasible upon a unit-capacity processor is that $U(\tau) \leq 1$.
- The special case of *synchronous systems* is also not quite as difficult as the general problem. The computational complexity of the feasibility-analysis problem for synchronous systems is, to our knowledge, still open; for synchronous periodic task systems τ with $U(\tau)$ bounded from above by a constant strictly less than one, however, a pseudo-polynomial time feasibility-analysis algorithm is known.

4.2 Static-priority Scheduling

(Optimal) Scheduling Algorithm Recall that the *run-time scheduling problem* was rendered trivial for all the task models we had considered in the dynamic-priority case due to the proven optimality of EDF as a dynamic-priority run-time scheduling algorithm. Unfortunately, there is no static-priority result analogous to this result concerning the optimality of EDF; hence, the run-time scheduling problem is quite non-trivial for static-priority scheduling.

In the static-priority scheduling of periodic systems, all the jobs generated by an individual task are required to be assigned the same priority, which should be different from the priorities assigned to jobs generated by other tasks in the system. Hence, the run-time scheduling problem essentially reduces to the problem of associating a unique priority with each task in the system. The specific results known are as follows:

- For *implicit-deadline* synchronous periodic task systems, the **Rate Monotonic** (RM) priority assignment algorithm, which assigns priorities to tasks in inverse proportion to their period parameters with ties broken arbitrarily, is an optimal priority assignment. That is, if there is any static priority assignment that would result in such a task system always meeting all deadlines, then the RM priority assignment for this task system will also result in all deadlines always being met.
- For implicit-deadline periodic task systems that are not synchronous, however, RM is provably not an optimal priority-assignment scheme (Goossens et al. (1997)).
- For *constrained-deadline* synchronous periodic task sets, the **Deadline Monotonic** (DM) priority assignment algorithm, which assigns priorities to tasks in inverse proportion to their deadline parameters with ties broken arbitrarily, is an optimal priority assignment.

- For constrained-deadline (and hence also arbitrary) periodic task systems which are not necessarily synchronous, however, the computational complexity of determining an optimal priority-assignment remains open. That is, while it is known (see below) that determining whether a constrained-deadline periodic task system is static-priority feasible is co-NP-complete in the strong sense, it is unknown whether this computational complexity is due to the process of assigning priorities, or merely to validating whether a given priority-assignment results in all deadlines being met. Audsley, N. et al. (1993) have proposed an optimal priority assignment which considers at most n^2 priority assignments. This algorithm is based upon the fact if a task τ_i is *lowest-priority viable* (i.e., τ_i is assigned the lowest priority, the remaining tasks are assigned higher priorities in an any arbitrary order and jobs generated by τ_i may not miss any deadline) then there is a feasible static priority assignment for τ iff there is a feasible static priority assignment for $\tau \setminus \{\tau_i\}$.

(Complexity of the) Feasibility Analysis Determining whether an arbitrary periodic task system τ is feasible has been shown to be intractable — co-NP-complete in the strong sense. This intractability result holds even if $U(\tau)$ is known to be bounded from above by an arbitrarily small constant.

Utilization-based feasibility analysis. For the special case of *implicit-deadline* periodic task systems, a simple sufficient utilization-based feasibility test is known: an implicit-deadline periodic task system τ is static-priority feasible if its utilization $U(\tau)$ is at most $n(2^{\frac{1}{n}}-1)$, where n denotes the number of tasks in τ. Since $n(2^{\frac{1}{n}}-1)$ monotonically decreases with increasing n and approaches $\ln 2$ as $n \to \infty$, it follows that any implicit-deadline periodic task system τ satisfying $U(\tau) \leq \ln 2$ is static-priority feasible upon a preemptive uniprocessor, and hence can be scheduled using rate-monotonic priority assignment.

This utilization bound is a sufficient, rather than exact, feasibility-analysis test: it is quite possible that an implicit-deadline task system τ with $U(\tau)$ exceeding the bound above be static-priority feasible. It may be also noticed that the bound is tight.

If $U(\tau) > \ln 2$ for implicit-deadline task systems or if we consider synchronous constrained-deadline task systems, Liu, C. et al. (1973) have proposed an exact test based on the response time notion. For a request of task τ_i, we define the *response time* as the time between the arrival of the request and the completion of its processing. Liu et al. (1973) have shown that for periodic synchronous task set with constrained deadlines the response time of the first request of any task τ_i (say r_i) is maximum among all requests of task τ_i. Consequently, a synchronous constrained-deadline system is feasible iff $r_i \leq D_i \; \forall i$. For asynchronous cases, an exponential-time feasibility test is known, which consists essentially of simulating the behavior of scheduling algorithm upon the periodic task system for a sufficiently long interval.

An another feasibility approach consists to consider the *time demand*, see for instance, Lehoczky et al. (1989), Joseph et al. (1996) or Lehoczky (1990)).

5 Fundamental results for multiprocessor systems

In multiprocessor computing machines there are several processors available upon which jobs may execute. In this section, we consider the scheduling of hard-real-time systems upon multiprocessors, under the assumptions that while job preemption and interprocessor migration is permitted *intra-job parallelism* is forbidden.

5.1 Multiprocessor platforms

In much results from the literature, it has been assumed that all the processors are identical. However, scheduling theorists distinguish between at least two different kinds of multiprocessors platforms:

Identical parallel machines: There are multiprocessors in which all the processors are identical, in the sense that they have the same computing power.

Uniform parallel machines: By contrast, each processor in a *uniform parallel* machines is characterized by its own computing capacity, with the interpretation that a job that executes on a processor of computing capacity s for t time units completes $s \times t$ units of computation.

Real-time scheduling theorists have extensively studied *uniprocessor* hard-real-time scheduling; recently, steps have been taken towards obtaining a better understanding of hard-real-time scheduling on *identical multiprocessors* (see below). However, not much is know about hard-real-time scheduling on uniform multiprocessors.

5.2 (Optimality) of Online Scheduling Algorithms

Online scheduling algorithms make scheduling decisions at each time-instant based upon the characteristics of the jobs that have arrived so far, with no knowledge of jobs that may arrive in the future. Several uniprocessor online scheduling algorithms, such as EDF are know to be *optimal*. For multiprocessors systems, however, no online scheduling algorithm can be optimal: This was shown for the simplest (identical) multiprocessors model by Dertouzos et al. (1989) and the techniques can be directly extended to the more general uniform machine model. An important advance in the study of online scheduling upon multiprocessors was obtained by Phillips et al. (1997), who explored the use of *resource-augmentation* techniques for online scheduling of real-time jobs. Phillips et al. investigated whether an online algorithm, if provided with faster processors than necessary for feasibility, could perform better than is implied by the above non-optimality results. They showed that the obvious extension to the EDF algorithm for identical multiprocessors could make the following performance guarantee: If a real-time instance is feasible on m identical processors, then the same instance will be scheduled to meet all deadlines by EDF on m processors in which the individual processors are $(2 - \frac{1}{m})$ times as fast as in the original system.

5.3 Scheduling of periodic tasks

Partitioned and global scheduling. In designing scheduling algorithms for multiprocessors environments, one can distinguish between at least two approaches. In *partitioned* scheduling, all jobs generated by a task are required to execute on the *same* processor. *Global scheduling,* by contrast permits task migration (i.e., different jobs of an individual task may execute upon different processors) as well as job-migration: an individual job that is preempted may resume execution upon a different from the one upon which it had been executing prior to preemption.

It has been proven by Leung et al. (1982) that the partitioned and global approaches to static-priority scheduling on identical multiprocessors are *incomparable*, in the sense that (i) there are task systems that are feasible on m identical processors under partitioned approach but for which no priority assignment exists which would cause all tasks to meet their deadlines under global scheduling on the same m processors; and (ii) there are task systems that are feasible on m identical processors under the global approach, but which cannot be partitioned into m distinct subsets such that each individual partition is uniprocessor static-priority feasible. This result of Leung et al. provides a very strong motivation to study both partitioned and the non-partitioned approaches to static-priority multiprocessors scheduling, since it is provably true that neither approach is strictly better than the other.

In Andersson et al. (2001), the rate-monotonic scheduling of periodic task systems upon identical multiprocessor platforms was studied. A *utilization bound* was derived such that any periodic task system with cumulative utilization no larger than this bound is guaranteed to be successfully scheduled by algorithm RM upon an identical multiprocessor platform. In Baruah et al. (2003), the authors generalized the result for uniform multiprocessor platforms.

(Optimal) Fair Scheduling In Baruah et al. (1996), the authors proved that the problem of *optimally* scheduling periodic tasks on identical multiprocessors could be solved *on-line* in polynomial time using PFair scheduling algorithms. Under PFair, each periodic task is executed at an (approximately) uniform rate (corresponding to its utilization factor) by breaking it into a series of quantum-length *subtasks.* Recent interests in fair scheduling can be found in Anderson et al. (2004).

6 Fundamental results for distributed systems

Distributed systems are uniprocessor systems that exchange data over networks. There is no shared memory. Usually, a task gets its input message at its beginning and sends its output messages at its end. No communication nor synchronization occur within the body of every task. From the scheduling point of view, networks are viewed as additional processors; transmitted messages can be modeled as non-preemptive tasks and their execution requirements as worst-case transmission delays. Under this approach, communications can be viewed as precedence constraints. A job is *activated* when all its input data have been delivered by the communication interface. The interval of time between the release of a job and its activation is its *release jitter.* Release jitters capture tasks and messages dependencies on a timely basis.

Consider a task τ_i that sends a message m to a task τ_j, then the worst-case release jitter of the message is equal to the worst-case response time of τ_i and the worst-case release jitter of τ_j is equal to the worst-case response time of m. Assume that worst-case response times of tasks upon its processor are computed by a function *Response Time(*$J_1, \ldots, J_n$*)* having release jitters as parameters. Such a function can be defined for every processor and network. Then, computing worst-case response times of tasks and messages can be achieved by the following method, called the *holistic analysis* (Tindell, et al 1994). First, compute the response times of tasks using the previous functions considering that release jitters are equal to zero (or to lower bounds of their worst-case values), then update worst-case release jitters according to these response times. The same process can be apply iteratively until release jitters and response times are still unchanged within two successive iterations (convergence is ensured if response-time functions are non-decreasing in function of the release jitters). These principles have been extended and improved by numerous authors (see for instance Palencia et al, (2002)).

7 Emergent Problems

In this section, we will list some scheduling problems or extension which are from our knowledge emergent.

Heterogeneous Platforms. In Section 5, we considered briefly the scheduling of multiprocessor systems, we considered (mainly) processors of the same type, or *identical.* A relatively new area of research concerns the more general platforms where the processors are different and cannot be used interchangeably. The model of heterogeneous processors (which includes uniform platforms) considered in recent works is known as the *unrelated processor* model. According to this model, each job can execute on some types of processors. Different types of processors may have different speeds. The execution times of each job on different types of processors are *unrelated.* (See Liu, 2000 for details.)

Power-Aware Embedded Systems. Energy saving is a major problem in the design of embedded systems. Some processors can dynamically change their speeds at run-time to reduce energy consumption. Real-time tasks must complete by their deadlines and the total energy consumption must be minimized. Two scheduling models have been studied in the literature: the speed of the processor is a continuous function or a step function. In the continuous case, the first off-line optimal algorithm has been proposed by Yao et al., 1995. A linear time algorithm, based on a shortest path problem in a graph, has been recently proposed (see Gaujal et al. 2003). In the discrete case, an optimal off-line fixed priority scheduling method has been proposed in Yun et al. 2002. More generally, the discrete case is NP-hard and is related to the multiple-choice knapsack problem (see Rusu et al. 2002).

Hierarchical Scheduling. The classical scheduling model consists of three elements: a resource model, a scheduling algorithm, and a work load model (e.g., the periodic task model). There has been a growing attention to a *hierarchical scheduling framework* that supports hierarchical resource sharing under *different* scheduling algorithm for different scheduling services. One of the motivations for developing such scheduling frameworks is the recognition of the fact that individual applications may be comprised of several sub-applications, each application would request a certain amount of the shared resource, and then further distribute this resource among the various sub-applications that comprise it. (See Lipari et al. 2001, for instance.)

Firm Real-Time Systems. Many real-time systems have firm real-time requirements which allow occasional deadline violations but discard any tasks that are not finished by their deadlines. To measure the goodness of such a system, a quality of service (QoS) metric is needed. Examples of such a metric include the average deadline miss rate, the average response time, etc. (See Liu et al. 2003, for instance.)

Memory Management (Garbage Collection). Automatic memory management and real-time systems are usually thought to be incompatible. As expected, difficulties occur when we consider the integration of a garbage collector, i.e., the mechanism responsible for the reclamation of unused memory in the heap, in a real-time system. Recent research works give first results to that kind of problems (see Pfeffer et al. 2004, for instance) but the problem in general remains open for further research.

References

ANDERSON, J., HOLMAN, PH. AND SRINIVASAN, A. (2004): Fair Scheduling of Real-Time Tasks on Multiprocessors: *Handbook of Scheduling: Algorithms, Models and Performance Evaluation.* Chapman et al, to appear.

ANDERSSON, B., BARUAH, S., JANSSON, J. (2001): Static-priority scheduling on multiprocessors, *Proceedings of the IEEE Real-time systems symposium* 193–202, IEEE Computer Society Press, 2001.

AUDSLEY, N.C., BURNS A., DAVIS R.I., TINDELL K.W. and WELLINGS A.J. (1995): Fixed Priorty preemptive scheduling: An historical perspective *Real-Time Systems* 8:173–198.

AUDSLEY, N.C., TINDELL, K.W. and BURNS, A. (1993): The end of the line for static cyclic scheduling? *Proceedings of the EuroMicro Conference on real-Time Systems*: 36–41, IEEE Computer Society Press, 1993.

BARUAH, S., COHEN, N., PLAXTON C.G. AND VARTEL, D. (1996): Proportionate progress: A notion of fairness in resource allocation, *Algorithmica* 15:600–625, 1996.

BARUAH, S., FUNK, S., GOOSSENS, J. (2003): Robustness results concerning EDF scheduling upon uniform multiprocessors, *IEEE Transactions on Computers* 52(9): 1185–1195, 2003.

BARUAH, S., HOWELL R. and ROSIER L. (1990): Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor, *Real-Time Systems* 2: 301–324.

BARUAH, S. and GOOSSENS, J. (2004): Scheduling Real-Time Tasks: Algorithms and complexity (J.Y.T Leung Eds): *Handbook of Scheduling: Algorithms, Models and Performance Evaluation.* Chapman et al, to appear.

BUTTAZZO, G.C. (1997):*Hard Real-Time Computing: Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers.

CARPENTER, J., FUNCK S., HOLMAN P., SRIVINASAN A., ANDERSON J. and BARUAH S. (2004): A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms (J.Y.T. Leung, Ed.): *Handbook of Scheduling: Algorithms, Models and Performance Evaluation.* Chapman et al, to appear.

COTTET, F., DELACROIX, J., KAISER C., and MAMMERI, Z. (2002) *Scheduling in Real-Time Systems: A course with exercices and solutions.*, John Wiley & Sons, 2002.

DERTOUZOS, M. and MOK, A.K. (1989): Multiprocessor scheduling in a hard real-time environment. *IEEE Trans. Software Eng.* 15(12):1497–1506, 1989.

GAUJAL B., NAVET N., WALSH C., A Linear Algorithm for Real-Time Scheduling with Optimal Energy Use: *INRIA Research report*, 4886, 2003.

GOOSSENS, J. and DEVILLERS, R. (1997): The non-optimality of the monotonic priority assignements for hard real-time offset free systems. *Real-Time Systems: The International Journal of Time-Critical Computing* 13(2):107–126, 1997.

JOSEPH, M. and PANDYA, P. (1996): Finding Response Times in a Real-Time System. *The Computer Journal* 29(5):390–395, 1996.

LEHOCZKY, J. (1990): Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. *Proceedings IEEE real-time systems symposium* 201–213, 1990.

LEHOCZKY, J., SHA, L. and DING, Y. (1989): The rate monotonic scheduling algorithm: exact characterization and average case behavior. *Proceedings IEEE real-time systems symposium* 166–171, 1989.

LEUNG, J. and WHITEHEAD, J. (1982): On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation* 2:237–250, 1982.

LIPARI, G. and BARUAH, S. (2001): A Hierarchical extension to the constant bandwidth server framework. *Proceedings of the seventh IEEE Real-Time Technology and Applications Symposium*: 26–35, 2001.

LIU, C. and LAYLAND, J. (1973): Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM* 20(1):46–61, 1973.

LIU, D., HU, X.S., LEMMON M.D., and LING, Q. (2003): Firm Real-Time System Scheduling Based on a Novel QoS Constraint. *Proceedings of the 24th IEEE Internationazl Real-Time Systems Symosium*: 386–397, 2003.

LIU, J.W.S. (2000): *Real-Time Systems* Prentice Hall.

MOK, A. and DERTOUZOS, M. (1978): Multiprocessor scheduling in a hard real-time environment. *Proceedings of the seventh Texas Conference on Computing Systems*: 1978.

PALENCIA, J.C., GONZALES-HARBOUR, M. (2002): Response Time Analysis of EDF Distributed Real-Time Systems *Journal of Embedded Computing*, to appear.

PFEFFER, M., UNGERER, T., FUHRMANN, S. KREUZINGER, J. and BRINKSCHULTE, U. (2004): Real-Time Garbage Collection for a Multithread Java Microcontroller. *Real-Time Systems* 26(1): 89–106, January 2004.

PHILIPS, C.A., STEIN, C., TORNG, E., and WEIN, J. (1997): Optimal time-critical scheduling via resource augmentation. *Proceedings 29th Ann. ACM Symp. Theory of Computing*: 140–149, 1997.

PRUHS, K., SGALL, J., TORNG, E. (2004): Online Scheduling. (J.Y.T Leung Eds): *Handbook of Scheduling: Algorithms, Models and Performance Evaluation.* Chapman et al, to appear.

RUSU C., MELHELM R., MOSSE D., Maximizing the system value while satisfying time and energy constraint: *proc. Real-Time Systems Symposium (RTSS'02)*, 2002.

SGALL, J. (1998): On-line scheduling*in: Online algorithmes: The state of the art*: 196–231, 1997.

TINDELL, K., CLARK J., (1994): Holistic Schedulability Analysis for Distributed Hard Real-Time Systems: *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, 40, 117–134.

YAO F., DEMERS A., SHENKER S., A scheduling model for reduced CPU energy: *proc. 36th Symposium on Foundations of Computer Science (FOCS'95)*,1995, 374-382.

YUN H.S., KIM J.,On Energy-optimal Off-line scheduling for fixed-priority Real-Time Systems on a variable voltage processor: *ACM Transactions on Embedded Computing Systems*,2(3), 2002,393–430.