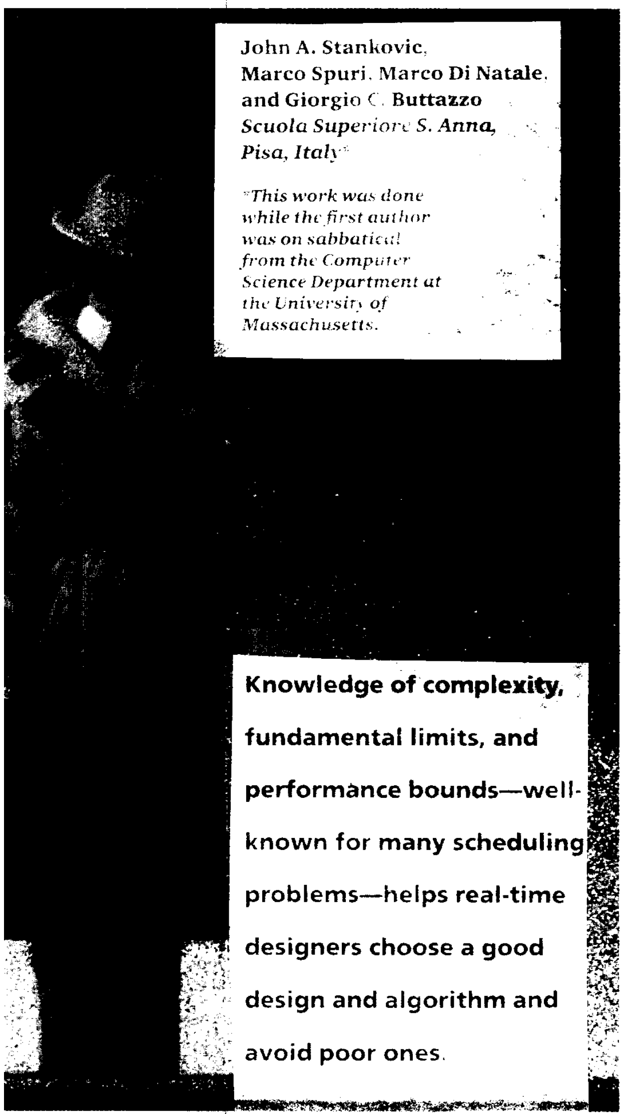
Implications of Classical Scheduling Results for Real-Time Systems



John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C. Buttazzo
*Scuola Superiore S. Anna, Pisa, Italy**

**This work was done while the first author was on sabbatical from the Computer Science Department at the University of Massachusetts.*

Knowledge of complexity, fundamental limits, and performance bounds—well-known for many scheduling problems—helps real-time designers choose a good design and algorithm and avoid poor ones.

Every real-time systems designer should be familiar with a set of classical scheduling theory results from the literature on complexity theory and operations research. These results rarely provide direct solutions, but they do provide insight in choosing a good system design and scheduling algorithm and avoiding poor or erroneous choices.

The scheduling theory literature is so vast that we can't pretend to be comprehensive, but this article does present a minimum set of results and their implications. The set includes Jackson's rule, Smith's rule, McNaughton's theorem, Liu and Layland's rate-monotonic rule, Mok's theorems, and Richard's anomalies. Besides learning about these results, readers should be able to answer, at least, the following questions:

- What do we really know about earliest deadline scheduling?
- What is known about uniprocessor real-time scheduling problems?
- What is known about multiprocessing real-time scheduling problems?
- What anomalous behavior can occur, and can it be avoided?
- Where is the boundary between polynomial and NP-hard scheduling problems?
- What task-set characteristics cause NP-hardness?
- What type of bounds analysis is useful for real-time systems?
- What is the impact of overloads on the scheduling results?
- How does the metric used in the theory impact the result's usefulness in a real-time system?
- What different results exist for static and dynamic scheduling?

The scheduling problem has so many dimensions that it has no accepted taxonomy. We divide scheduling theory between uniprocessor and multiprocessor results. In the uniprocessor section, we begin with independent tasks and then consider shared resources and overload. In the multiprocessor section, we divide the work between static and dynamic algorithms.

PRELIMINARIES

First, we need to clarify a few basic concepts. These include the differences between static, dynamic, off-line, and on-line scheduling; an

0018-9162/95/$4.00 © 1995 IEEE

overview of various metrics and their implications; and definitions for NP-complete and NP-hard, terms used throughout the article.

Static versus dynamic scheduling

Most classical scheduling theory deals with static scheduling. In static scheduling, the scheduling algorithm has complete knowledge of the task set and its constraints, such as deadlines, computation times, precedence constraints, and future release times. This set of assumptions is realistic for many real-time systems. For example, a simple laboratory experiment or a simple process-control application might have a fixed set of sensors and actuators and a well-defined environment and processing requirements; the static scheduling algorithm operates on this set of tasks and produces a single schedule that is fixed for all time. If all future release times are known when the algorithm is developing the schedule, then it is still a static algorithm.

A dynamic scheduling algorithm (in the context of this article) has complete knowledge of currently active tasks, but new task activations, not known to the algorithm when it is scheduling the current set, may arrive. Therefore, the schedule changes over time. For example, teams of robots cleaning up a chemical spill or military command and control applications require dynamic scheduling, but as we will see, there are few known results for real-time dynamic scheduling algorithms.

Off-line scheduling is often equated to static scheduling, but this is wrong. In building a real-time system, off-line scheduling analysis should always be done, regardless of whether the final runtime algorithm is static or dynamic. In many real-time systems, designers can identify the maximum set of tasks with their worst-case assumptions and apply a static scheduling algorithm to produce a static schedule. This schedule is then fixed and used on line with well-understood properties such as, given that all assumptions remain true, all tasks will meet their deadlines. In other cases, the off-line analysis might produce a static set of priorities to use at runtime. The schedule itself is not fixed, but the priorities that drive it are fixed. (This is common in the rate-monotonic approach discussed later.)

If the real-time system is operating in a more dynamic environment, meeting the assumptions of static scheduling (everything is known a priori) is not feasible. In this case, designers choose an algorithm and analyze it off line for the expected dynamic environmental conditions. Usually, they can make less precise statements about the overall performance. On line, this same dynamic algorithm executes.

Generally, a scheduling algorithm (possibly with some modification) can be applied to static or dynamic scheduling and used off or on line. The important difference is what is known about the algorithm's performance in each case. For example, consider earliest-deadline-first (EDF) scheduling. When applied to static scheduling, EDF is optimal in many situations (enumerated below), but when applied to dynamic scheduling on multiprocessors, it is not optimal.

Metrics

Classical scheduling theory typically uses metrics such as minimizing the sum of completion times, minimizing the weighted sum of completion times, minimizing schedule length, minimizing the number of processors required, or minimizing the maximum lateness. In most cases, deadlines are not even considered. When deadlines are considered, they are usually added as constraints—for example, creating a minimum schedule length subject to the constraint that all tasks must meet their deadlines. If one or more tasks miss their deadlines, there is no feasible solution.

Which of these classical metrics (where deadlines are not included as constraints) are of most interest to real-time systems designers? The sum of completion times is generally not of interest because there is no direct assessment of timing properties (deadlines or periods). However, the weighted sum is very important when tasks have different values that they impart to the system upon completion. Value is often overlooked in real-time systems, many of which simply focus on deadlines rather than a combination of value and deadlines. Minimizing schedule length has secondary importance in minimizing required system resources, but it does not directly address the fact that individual tasks have deadlines. The same is true for minimizing the number of processors required. Minimizing the maximum lateness metric can be useful at design time when resources can be added until the maximum lateness is less than or equal to zero. In that case, no task misses its deadline. On the other hand, minimizing the maximum lateness doesn't necessarily prevent one, many, or even *all* tasks from missing their deadlines (see Figure 1).

Instead of using the above metrics, much real-time computing work looks for optimal algorithms defined as follows: An optimal scheduling algorithm is one that may fail to meet a deadline only if no other scheduling algorithm can meet it. This article mentions all the above metrics,

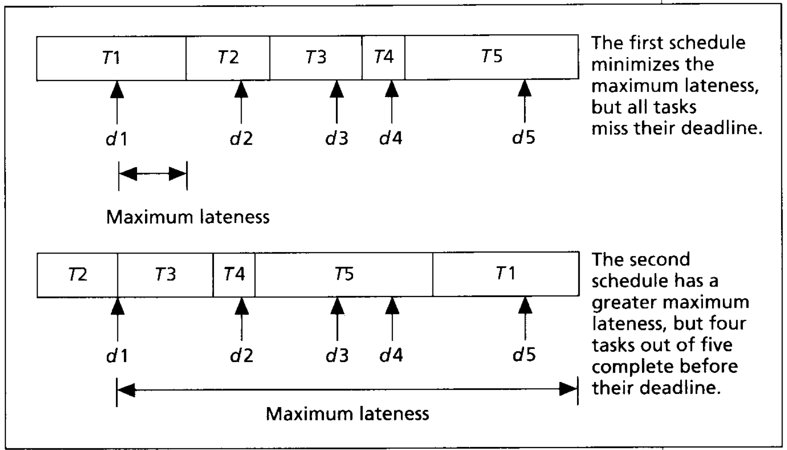

T1
T2
T3
T4
T5
d1
d2
d3
d4
d5
Maximum lateness
The first schedule minimizes the maximum lateness, but all tasks miss their deadline.
T2
T3
T4
T5
T1
d1
d2
d3
d4
d5
Maximum lateness
The second schedule has a greater maximum lateness, but four tasks out of five complete before their deadline.

Figure 1. Minimizing maximum lateness example.

either because of their direct applicability to real-time systems or to show their limited applicability (even if a nice theoretical result exists).

Related to metrics is the complexity of the various scheduling problems themselves. Many scheduling results are NP-complete or NP-hard. *NP* is the class of all decision problems that can be solved in polynomial time by a nondeterministic machine. A recognition problem R is NP-complete if $R \in NP$ and all other problems in *NP* are polynomial transformable to R. A recognition or optimization problem R is NP-hard if all problems in *NP* are polynomial transformable to R, but we can't show that $R \in NP$.

UNIPROCESSOR SYSTEMS

In general, we follow Lawler's notation,[1] in which the problem definition has the form $\alpha \mid \beta \mid \gamma$, where α indicates the machine environment (in this section, $\alpha = 1$, indicating a uniprocessor machine), β indicates the job characteristics (preemptable, nonpreemptable, independent, precedence constrained, deadline, etc.) and γ indicates the optimality criterion (minimizing maximum lateness, total tardiness, etc.). Note that the optimality criterion depends on the metric chosen, which relies strongly on the system objectives and the task model.

Preemption versus nonpreemption: Jackson's Rule

Suppose there are n independent jobs (job, process, and task are used interchangeably throughout the scheduling literature), with each job j having a processing time p_j and a due date d_j. For any given scheduling sequence, each job will have a defined completion time C_j. Lateness of a job j is defined as $L_j = C_j - d_j$. Suppose we want to minimize the maximum lateness, assuming the jobs are executed nonpreemptively; that is, we want to solve the problem

If our real-time system requires a more sophisticated programming model, one of the first extensions that might be needed is the introduction of release times.

$$1 \mid nopmtn \mid L_{\max}$$

where 1 stands for single machine, *nopmtn* stands for nonpreemption and the objective function to minimize is

$$L_{\max} = \max_j \{L_j\}$$

A very simple solution to this problem, the earliest-due-date (EDD) algorithm, is

> **Theorem 1 (Jackson's Rule[2]).** Any sequence is optimal that puts the jobs in order of nondecreasing due dates.

The proof of the theorem can be given by a simple interchange argument[1] (presenting that argument is beyond the scope of this article). At first, this result may not seem too useful in real-time systems design, which often requires that no task miss its deadline. But this is a static scheduling algorithm, and if the maximum lateness is greater than zero, the designer knows he must increase system computing power to meet the requirement of missing no deadlines. EDD is also optimal in many other situations. Note that since all tasks are known and ready to execute at time zero, preemption would not improve the situation.

If our real-time system requires a more sophisticated programming model, one of the first extensions that might be needed is the introduction of release times. We say that a job j has release time r_j if its execution cannot start before time r_j. Unfortunately, the above problem extended with release times

$$1 \mid nopmtn, r_j \mid L_{\max}$$

is NP-hard.[3]

In this case, we obtain great benefit by permitting job preemption at any instant. In fact, the problem

$$1 \mid pmtn, r_j \mid L_{\max}$$

is easy; that is, an algorithm for its solution exists and has polynomial complexity. Again, the algorithm is based on Jackson's rule, slightly modified to account for release times:

> **Theorem 2.** Any sequence that at any instant schedules the job with the earliest due date among all the eligible jobs (that is, those whose release time is less than or equal to the current time) is optimal with respect to minimizing maximum lateness.

Again, the result can be easily proved by an interchange argument. The proof obtained is very similar to the "time slice swapping" technique Dertouzos[4] and Mok[5] used to show the optimality of EDF and the least-laxity-first (LLF) algorithms, respectively.

These results imply that when practical considerations do not prevent its use, preemption is usually more beneficial than nonpreemption in terms of scheduling complexity. Unfortunately, dealing with shared resources in real-time systems requires addressing critical sections, and one technique—creating nonpreemptable code—again creates an NP-hard problem.

These theorems also imply that minimizing maximum lateness is optimal even when all deadlines must be met, because the maximum lateness can be required to be less than or equal to zero. In fact, Liu and Layland,[6] in their well-known paper on this aspect of EDF scheduling for a set of independent periodic processes, showed that full processor utilization is always achievable. They gave a very simple necessary-and-sufficient condition for task schedulability:

$$\sum_j \frac{p_j}{T_j} \le 1$$

where p_j is the processing time and T_j is the period of task j.

The EDF algorithm has also been shown to be optimal under various stochastic conditions. All of these results imply that EDF works well under many different situations. EDF variations are now being used in multimedia applications, robotics, and real-time databases. However, none of the above classical EDF results allows for precedence constraints, shared resources, or overloads.

Another important area, real-time scheduling of periodic tasks, often uses the rate-monotonic algorithm. This algorithm assigns each task a static priority inversely proportional to its period; that is, tasks with the shortest periods get the highest priorities. For a fixed set of independent periodic tasks with deadlines the same as the periods, we know:

Theorem 3 (Liu and Layland[6]). A set of n independent, periodic jobs can be scheduled by the rate monotonic policy if $\sum_{i=1}^{n} p_i/T_i \leq n(2^{1/n}-1)$, where T_i and p_i are the period and worst-case execution time, respectively.

For large n we obtain a 69 percent utilization bound, meaning that as long as CPU utilization is less than 69 percent, all tasks will make their deadlines. This is often referred to as the schedulability test. If periodic task deadlines can be less than the period, the above rule is no longer optimal. Rather, we must use a deadline-monotonic policy[7] where the periodic process with the shortest deadline is assigned the highest priority. This scheme is optimal in the sense that if any static priority scheme can schedule this set of periodic processes, then the deadline-monotonic algorithm can. Note that deadline monotonic is not the same as pure EDF scheduling because tasks may have different periods and the assigned priorities are fixed. The rate-monotonic algorithm has been extended in many ways, the most important deals with shared resources (see the next section), and schedulability tests have been formulated for the deadline-monotonic algorithm.[8]

The rate-monotonic scheduling algorithm has been chosen for the Space Station Freedom Project and the FAA Advanced Automation System (AAS). It also influenced the specification of the IEEE Futurebus+. The DoD's 1991 Software Technology Strategy says that rate-monotonic scheduling has a "major payoff," and "system designers can use this theory to predict whether task deadlines will be met long before the costly implementation phase of a project begins." In 1992, the Acting Deputy Administrator of NASA stated, "Through the development of Rate Monotonic Scheduling, we now have a system that will allow [Space Station] Freedom's computers to budget their time, to choose between a variety of tasks, and decide not only which one to do first but how much time to spend in the process." Rate monotonic is also useful for simple applications, such as real-time control of a simple experiment with 20 sensors whose data must be processed periodically or of a chemical plant with many periodic tasks and few alarms. These alarms can be treated as periodic tasks whose minimum interarrival time is equal to its period; then static scheduling, using the rate-monotonic algorithm, can be applied.

Shared resources

Multitasking applications commonly share resources. In general-purpose systems, resource sharing can be accomplished by, for example, mutual exclusion primitives, but in real-time systems, a straightforward application of this solution does not hold. Defining a runtime scheduler as totally on line if it has no knowledge about the future arrival times of the tasks, the following has been proven:

Theorem 4 (Mok[5]). When there are mutual exclusion constraints, it is impossible to find a totally on-line optimal runtime scheduler.

The proof is simply given by an adversary argument, and Mok also showed a much more negative result:

Theorem 5 (Mok[5]). The problem of deciding whether it is possible to schedule a set of periodic processes that use semaphores only to enforce mutual exclusion is NP-hard.

A transformation of the three-partition problem to this scheduling problem proves the theorem.

The rate-monotonic scheduling algorithm has been chosen for the Space Station Freedom Project and the FAA Advanced Automation System.

In Mok's opinion, "the reason for the NP-hardness of the above scheduling problem lies in the possibility that there are mutually exclusive scheduling blocks which have different computation times." This point of view is confirmed by ease of minimizing the maximum lateness of n independent unit-time jobs with arbitrary release times[1]:

$$1 \mid nopmtn, r_j, p_j = 1 \mid L_{\max}$$

Moreover, the problem is still easy[9] if we add precedence constraints and minimize the maximum completion time (makespan):

$$1 \mid nopmtn, prec, r_j, p_j = 1 \mid C_{\max}$$

The solution uses forbidden regions, intervals of time during which no task can start if the schedule is to be feasible. The idea is that because of the nonpreemption, scheduling a task at a certain point in time could force some other later task to miss its deadline.

At this point, several choices are possible. One, followed by Mok, is to enforce the use of mutually exclusive scheduling blocks having the same computation time. Another, followed by Sha et al.[10] and Baker,[11] is to efficiently find a suboptimal solution with a clever allocation policy, guaranteeing at the same time a minimum level of performance.

The idea in Mok's solution, called kernelized monitor, is to assign the processor in time quantums of length q such that

$$q \geq \max_i \{l(CS_i)\}$$

where $l(CS_i)$ is the length of the ith critical section. In other words, system granularity is increased. Furthermore, ready times and deadlines can be previously modified according to some partial order on tasks. Adjusting the EDF scheduler with the forbidden-region technique, the following theorem can be proven:

Theorem 6 (Mok[5]). If a feasible schedule exists for an instance of the process model with precedence constraints and critical sections, then the kernelized monitor scheduler can be used to produce a feasible schedule.

Sha et al.[10] introduced the priority ceiling protocol (PCP), an allocation policy for shared resources that works with a rate monotonic scheduler. Chen and Lin[12] extended PCP to an EDF scheduler.

The main goal of PCP and similar protocols is to bound the (usually uncontrolled) priority inversion, a situation in which lower priority jobs block a higher priority job for an indefinite period (recall that a block can occur if a job tries to enter a critical section already locked by another job). A priority inversion bound lets us evaluate the worst-case blocking times and account for them in the schedulability guaranteeing formulas—in other words, evaluate the worst-case performance loss.

Stack resource policy handles a more general situation that allows multiunit resources, both static and dynamic priority schemes, and sharing of runtime stacks.

PCP seeks to prevent multiple priority inversions by early blocking of tasks that could cause them and to minimize a priority inversion's length by allowing a temporary rise in the blocking task's priority. This is done by (1) defining a critical section's ceiling as the priority of the highest priority task that currently locks or could lock the section and (2) locking a critical section only if the requesting task's priority is higher than the ceiling of all currently locked sections. In case of blocking, the task holding the lock inherits the requesting task's priority until it leaves the critical section.

Sha et al.[10] also showed that PCP has the following properties:

- A job can be blocked at most once before it enters its first critical section.
- PCP prevents the occurrence of deadlocks.

Of course, the first property is used to evaluate the jobs' worst-case blocking times.

Baker[11] describes a similar protocol, the stack resource policy. SRP handles a more general situation that allows multiunit resources, both static and dynamic priority schemes, and sharing of runtime stacks. The protocol relies on two conditions:

- To prevent deadlocks, a job should not be permitted to start until the resources currently available are sufficient to meet its maximum requirements.
- To prevent multiple priority inversions, a job should not be permitted to start until the resources currently available are sufficient to meet the maximum requirement of any single job that might preempt it.

The idea is to block a job early if there is any chance of either deadlock or priority inversion. This earlier blocking saves unnecessary context switches and permits simple, efficient implementation by means of a stack.

In summary, dealing with shared resources is of utmost importance in a real-time system. The classical methods described are good for handling uniprocessor resources, but many researchers feel these techniques do not work well in multiprocessors or distributed systems. These systems typically use on-line planning algorithms[13-15] or static schedules developed with off-line heuristics. Both approaches avoid blocking over shared resources by scheduling competing tasks at different points in time.

Overload and value

EDF and LLF are optimal with respect to different metrics, but Locke's[16] experiments show that these algorithms perform very poorly in overload conditions. That's because they give the highest priority to processes that are close to missing their deadlines.

A typical phenomenon that may happen with EDF when the system is overloaded is the "domino effect," since the first task that misses its deadline may cause all subsequent tasks to miss their deadlines. In such a situation, EDF does not provide any type of guarantee on which tasks will meet their timing constraints. This is a very undesirable behavior in practical systems, since in real-world applications intermittent overloads may occur due to exceptional situations, such as modifications in the environment, arrival of a burst of tasks, or cascades of system failures. As a real world example, this situation could cause a flexible manufacturing application to produce no completed products by their deadlines.

To gain control over tardy tasks in overload conditions, we usually associate a value with each task that reflects its importance within the set. Sets of tasks with values can be scheduled by Smith's rule.

Theorem 7 (Smith's rule[17]). Finding an optimal schedule for

$$1 \mid\mid \sum w_j C_j$$

is given by any sequence that puts jobs in order of nondecreasing ratios $\rho_j = p_j / w_j$.

Smith's rule resembles the common shortest-processing-time-first rule and is equivalent to SPT when all tasks have equal weights. However, it is not sufficient to solve the problem of scheduling with general precedence constraints. The problems

$$1 \mid prec \mid \sum w_j C_j$$

$$1 \mid d_j \mid \sum w_j C_j$$

turn out to be NP complete,[3] and the same is true even for the simpler ones

$$1 \mid prec \mid \sum C_j$$

$$1 \mid prec, p_j = 1 \mid \sum w_j C_j$$

Interesting solutions have been found for particular precedence relations; in fact, optimal polynomial algorithms have been found for

$$1 \mid chain \mid \sum C_j$$

$$1 \mid series\text{-}parallel \mid \sum C_j$$

$$1 \mid d_j \mid \sum C_j$$

Unfortunately, in real-time systems the precedence constraints imposed on tasks are often more general. A heuristic proposed in the Spring project combined deadline- and cost-driven algorithms with rules to dynamically revise values and deadlines in accordance with precedence relations.[18]

A number of heuristic EDF algorithms have also been proposed to deal with EDF overloads,[19,20] thus improving the performance of EDF.

Baruah et al.[21] have shown that there's an upper bound on the performance of any on-line (preemptive) algorithm working in overload conditions. They measured an on-line algorithm's "goodness" with respect to a clairvoyant scheduler (one that knows the future) by means of the competitive factor, which is the ratio r of the cumulative value achieved by the on-line algorithm to the cumulative value achieved by the clairvoyant schedule. The value associated with each task is equal to the task's execution time if the task request is successfully scheduled to completion; a value of zero is given to tasks that do not terminate within their deadline. Using this metric, they proved the following:

> **Theorem 8 (Baruah et al.[21]).** There does not exist an on-line scheduling algorithm with a competitive factor greater than 0.25.

That is, no on-line scheduling algorithm can guarantee a cumulative value greater than one fourth the value obtainable by a clairvoyant scheduler. These bounds are true for any load but can be refined for a given load. For example, if the load is less than 1, the bound is 1; as the load just surpasses 1, the bound immediately drops to 0.385. For loads greater than 1 up to 2, the bound gradually drops from 0.385 to 0.25, and for all loads greater than 2, the bound is 0.25.

However, the above bound is achieved under very restrictive assumptions: all tasks in the set have zero laxity, the overload has an arbitrary (but finite) duration, task execution time is arbitrarily small, and task value is equal to computation time. Since tasks are much less restrictive in most real-world applications, the one-fourth bound has only theoretical validity. More work is needed to derive other bounds based on more knowledge of the task set.

Summary of uniprocessor results

Many basic algorithms and theoretical results have been developed for uniprocessor scheduling. A number are based on earliest deadline or rate-monotonic scheduling and have been extended to handle precedence and resource sharing. Thus, real-time system designers have a wealth of information concerning uniprocessor scheduling, but they need more results on overload and fault-tolerant scheduling (although fault tolerance usually requires multiple processors as well). We also need—to name a few issues—a more integrated, comprehensive scheduling approach that addresses periodic and aperiodic tasks, preemptive and nonpreemptive tasks in the same system, tasks with values, and combined CPU and I/O scheduling. For example, the A-7E aircraft's operational flight program, which has 75 periodic and 172 aperiodic processes with significant synchronization requirements, could use rate-monotonic extensions that integrate periodic and aperiodic tasks.

MULTIPROCESSOR REAL-TIME SCHEDULING

More and more real-time systems are relying on multiprocessors. Unfortunately, we know less about real-time scheduling for multiprocessor-based systems than for uniprocessors. This is partly because complexity results show that most real-time multiprocessing scheduling is NP-hard. Also, because of our minimal experience with such systems, the number of existing heuristics is relatively low. Despite the negative implications of complexity analysis, designers need to understand certain results:

- Understanding the boundary between polynomial and NP-hard problems can provide insights into developing useful heuristics that can be used as a design tool or as an on-line scheduling algorithm.
- Understanding the algorithms that achieve some of the polynomial results can again provide a basis for such heuristics.
- Understanding the fundamental limitations of on-line algorithms will help designers create robust systems and avoid misconceptions and serious scheduling anomalies.

Deterministic (static) scheduling

In this section, we present multiprocessing scheduling results for deterministic (static) scheduling with and without preemption.

NONPREEMPTIVE MULTIPROCESSING RESULTS. For our multiprocessing model, assume a set of P processors, T tasks, and R resources. The processors are identical. Each task is nonpreemptive and has a worst-case execution time of τ. Tasks may be related by a partial order indicating, for example, that task $T(i)$ must complete before task $T(j)$. Note that in most scheduling theory results, tasks are considered to have constant execution times. In most computer applications, however, execution time is not constant—a fact that raises one of the interesting multiprocessing anomalies of real-time scheduling (see the section on anomalies). For each resource $R(k)$, there is a number that indicates its capacity. Tasks can then require a portion of that resource. This directly models a resource like main memory. It can also model a mutually exclusive resource by requiring the task to access 100 percent of the resource.

Unfortunately, we know less about real-time scheduling for multiprocessor-based systems than for uniprocessors.

Complexity results where tasks are nonpreemptive and have a partial order among themselves, resource constraints (even a single resource constraint), and a single deadline show that most of the problems are NP-complete. To delineate the boundary between polynomial and NP-hard problems and to present basic results that every real-time designer should know, we list the following theorems without proof and compare them in Table 1. The metric used in the theorems is the amount of computation time required for determining a schedule that satisfies the partial order and resource constraints and completes all required processing before a given fixed deadline.

Table 1. Summary of basic multiprocessor scheduling theorems.

Theorem number	Processors	Resources	Ordering	Computation time	Complexity
9	2	0	Arbitrary	Unit	Polynomial
10	2	0	Independent	Arbitrary	NP-complete
11	2	0	Arbitrary	1 or 2 units	NP-complete
12	2	1	Forest	Unit	NP-complete
13	3	1	Independent	Unit	NP-complete
14	N	0	Forest	Unit	Polynomial
15	N	0	Arbitrary	Unit	NP-complete

Theorem 9 (Coffman and Graham[22]). The multiprocessor scheduling problem with two processors, no resources, arbitrary partial order relations, and every task having a unit computation time is polynomial.

Theorem 10 (Garey and Johnson[23]). The multiprocessor scheduling problem with two processors, no resources, independent tasks, and arbitrary computation times is NP-complete.

Theorem 11 (Garey and Johnson[23]). The multiprocessor scheduling problem with two processors, no resources, arbitrary partial order, and task computation times of either 1 or 2 units of time is NP-complete.

Theorem 12 (Garey and Johnson[23]). The multiprocessor scheduling problem with two processors, one resource, a forest partial order, and each computation time of every task equal to 1 is NP-complete.

Theorem 13 (Garey and Johnson[23]). The multiprocessor scheduling problem with three or more processors, one resource, all independent tasks, and each task computation time equal to 1 is NP-complete.

Theorem 14 (Hu[24]). The multiprocessor scheduling problem with n processors, no resources, a forest partial order, and each task having a unit computation time is polynomial.

Theorem 15 (Ullman[25]). The multiprocessing scheduling problem with n processors, no resources, arbitrary partial order, and each task having a unit computation time is NP-complete.

From these theorems we can see that for nonpreemptive multiprocessing scheduling almost all problems are NP-complete, implying that heuristics must be used for such problems. Basically, we see that nonuniform task computation time and resource requirements cause NP-completeness immediately. These results imply that designs using only local resources (such as object-based systems and functional language-based systems) and unit-time-slot scheduling have significant advantages as far as scheduling complexity is concerned. Of course, few if any real-time systems have unit tasks, and any attempt to carve a process into unit times creates difficult maintenance problems and can waste processing cycles when tasks consume less than the allocated unit of time. Also, the above results assume a single deadline for all tasks. If each task has a deadline, the problem is exacerbated.

PREEMPTIVE MULTIPROCESSING REAL-TIME SCHEDULING. Generally, the scheduling problem is easier if tasks are preemptable, but in certain situations, there is no advantage to preemption. The following classical results pertain to multiprocessing scheduling where tasks are preemptable; that is,

$$P \mid pmtn \mid \sum_j w_j C_j$$

Theorem 16 (McNaughton[26]). For any instance of the multiprocessing scheduling problem with P identical machines, preemption allowed, and minimizing the weighted sum of completion times, there exists a schedule with no preemption for which the value of the sum of computation times is as small as for any schedule with a finite number of preemptions.

Here we see that, for a given metric, there may be no advantage to preemption. However, to find such a schedule with or without preemption is NP-hard. Note that if the metric is the sum of completion times, the shortest-processing-time-first greedy approach solves the problem and is not *NP*. Here again, preemption offers no advantage. This result can have an important implication when creating a static schedule. We certainly prefer to minimize preemption for practical reasons at runtime, so knowing there is no advantage to preemption, a designer would not create a static schedule with any preemptions.

Theorem 17 (Lawler[1]). The multiprocessing problem of scheduling P processors with task preemption allowed and with minimization of the number of late tasks is NP-hard.

This theorem indicates that one of the most common forms of real-time multiprocessing scheduling—that is,

$$P \mid pmtn \mid \sum U_j$$

where U_j are the late tasks—requires heuristics.

Dynamic multiprocessor scheduling

There are so few real-time classical scheduling results for dynamic multiprocessing scheduling that we treat preemptive and nonpreemptive cases together.

In a uniprocessor, dynamic earliest-deadline scheduling is optimal under certain conditions. Is this algorithm optimal in a multiprocessor? The answer is no.

Theorem 18 (Mok[5]). Earliest-deadline scheduling is not optimal in the multiprocessor case.

To illustrate why this is true, consider the following example. We have three tasks to execute on two processors. The task characteristics, given by task number (computation time, deadline), are $T_1(1,1)$, $T_2(1,2)$, and $T_3(3,3.5)$. Scheduling by earliest deadline would execute T_1 on P1 and T_2 on P2, and T_3 would miss its deadline. However, if we schedule T_3 first, on P1, and then T_1 and T_2 on P2, all tasks make their deadlines. An optimal algorithm does exist for the static version of this problem (all tasks exist at the same time) if one considers both deadlines and computation time,[27] but this algorithm is too complicated to present here.

A number of good heuristics exist for dynamic multiprocessor scheduling, and we are beginning to see stochastic analysis of these conditions.

Now, if dynamic earliest-deadline scheduling for multiprocessors is not optimal, the next question is whether any dynamic algorithm is optimal in general. Again, the answer is no.

Theorem 19 (Mok[5]). For two or more processors, no deadline scheduling algorithm can be optimal without complete a priori knowledge of deadlines, computation times, and task start times.

This implies that any of the classical scheduling theory algorithms that require start-time knowledge cannot be optimal if used on line. This also points out that we cannot hope to develop an optimal, general on-line algorithm. But optimal algorithms may exist for a given set of conditions.

One important example of this situation is assuming that all worst-case situations exist simultaneously. If this scenario is schedulable, then it will also be schedulable at runtime—even if the arrival times are different—because later arrivals can't make conditions any worse. When such a worst-case approach is not possible for a given system, usually because sufficient conditions cannot be developed or because ensuring these conditions is too costly, more probabilistic approaches are needed.

A number of good heuristics exist for dynamic multiprocessor scheduling, and we are beginning to see stochastic analysis of these conditions. It is especially valuable to be able to create algorithms that operate with levels of guarantee. For example, even though the system operates stochastically and nonoptimally, it might provide a minimum level of guaranteed performance.

As mentioned, various heuristics exist for real-time multiprocessor scheduling with resource constraints.[13] However, in general, these heuristics use a nonpreemptive model. The advantages of a nonpreemptive model are few context switches, better understandability, and easier testing than for the preemptive model; also, blocking can be avoided. The main disadvantage of the nonpreemptive model is (usually) less-efficient processor use. Heuristics also exist for a preemptive model.[15] The advantages of a preemptive model are high use and low latency to newly invoked work. The disadvantages are many context switches, difficulty in understanding the runtime execution and its testing, and blocking is common. All these heuristics, whether preemptive or nonpreemptive, are fairly expensive in terms of absolute on-line computation time compared to very simple algorithms such as EDF. Thus, they sometimes requires additional hardware support in terms of a scheduling chip.

As mentioned earlier, overload and performance bounds analysis are important issues. Now assume we have a situation with sporadic tasks, preemption permitted. Also assume that if the task meets its deadline, then a value equal to the execution time is obtained; otherwise, no value is obtained. The system has two processors and operates in both normal and overload conditions.

Theorem 20 (Baruah, et al.[21]). No on-line scheduling algorithm can guarantee a cumulative value greater than one half for the dual processor case.

For uniprocessor bounds results (presented in the section on overload and value), the implications of this theorem are very pessimistic. As before, some of the pessimism arises because of assumptions concerning lack of knowledge of the task set. In reality, we do have significant knowledge. (We know the arrival of new instances of periodic tasks, or because of flow control, we may know that the maximum arrival rate is capped or the minimum laxity of any task in the system is greater than some value). If we can exploit this knowledge, the bounds may not be so pessimistic, but we do need more algorithms that directly address multiprocessing system performance in overload conditions.

Multiprocessing anomalies

Designers should be aware of several important anomalies, called Richard's anomalies, so that they can avoid them. Assume that a set of tasks are optimally scheduled on a multiprocessor with some priority order, a fixed number of processors, fixed execution times, and precedence constraints.

Theorem 21 (Graham[28]). For the stated problem, changing the priority list, increasing the number of processors, reducing execution times, or weakening the precedence constraints can increase the schedule length.

This result implies that if tasks have deadlines, then the accompanying increase in schedule length due to the anomaly can invalidate a previously valid schedule, and tasks can now miss deadlines. Initially, it's counter intuitive to think that adding resources (for example, an extra processor) or relaxing constraints (less precedence among tasks or fewer execution time requirements) can make things worse. But that's the insidious nature of timing constraints and multiprocessing scheduling. An example can best illustrate why this theorem is true. Consider an optimal schedule where we now reduce the time required for the first task *T*1 on the first processor. This means that the second task *T*2 on that processor can begin earlier. However, doing this may now cause some task on another processor to block over a shared resource and miss its deadline. If *T*2 had not executed earlier, then no blocking would have occurred, and all tasks would have made their deadlines because it was originally an optimal schedule. (See Figure 2.)

Note that for most on-line scheduling algorithms, we must deal with the problem of tasks completing before their worst-case times. A simple solution that avoids the anomaly is to have tasks that complete early simply idle, but this can be very inefficient. Algorithms such as Shen's[14] strive to reclaim this idle time, while carefully addressing the anomalies so that they will not occur.

Summary of multiprocessor results

Most multiprocessor scheduling problems are *NP*, but for deterministic scheduling this is not a major problem. We can use a polynomial algorithm and develop an optimal schedule if the specific problem is not NP-complete, or we can use off-line heuristic search techniques based on classical theory implications. These off-line techniques usually need to find only feasible schedules, not optimal ones. Many heuristics perform well in the average case and only deteriorate to exponential complexity in the worst (rare) case. Good design tools would allow users to provide feedback and redesign the task set to avoid the rare case. So the static, multiprocessor, scheduling problem is largely solved in the sense that we know how to proceed. However, good tools with implemented heuristics are still necessary, and many extensions that treat more sophisticated tasks and system characteristics are still possible. On-line multiprocessing scheduling must rely on heuristics and would be substantially helped by special scheduling chips. Any such heuristics must avoid Richard's anomalies.[14] Better results for operation in overloads, better bounds that account for typical a priori knowledge found in real-time systems, and algorithms that can guarantee various performance levels are required. Dynamic multiprocessing scheduling is in its infancy.

On-line multiprocessing scheduling must rely on heuristics and would be substantially helped by special scheduling chips.

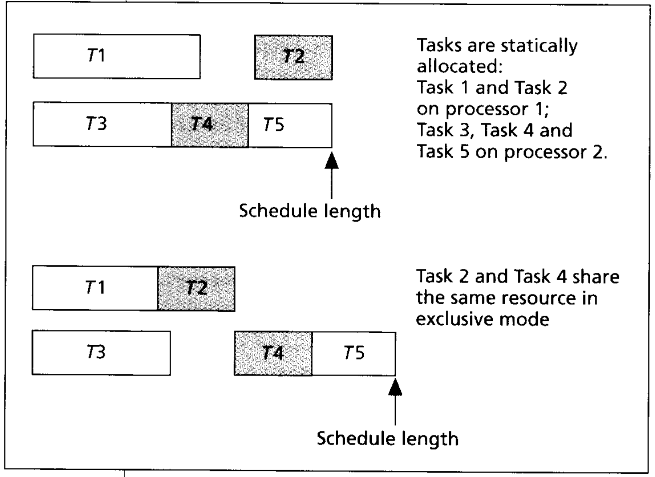

T1
T2
T3
T4
T5
Tasks are statically allocated: Task 1 and Task 2 on processor 1; Task 3, Task 4 and Task 5 on processor 2.
Schedule length
T1
T2
T3
T4
T5
Task 2 and Task 4 share the same resource in exclusive mode
Schedule length

Figure 2. One example of Richard's anomalies.

AS WE'VE SHOWN, CLASSICAL SCHEDULING THEORY PROVIDES a basic set of results for real-time system designers. Many results are known for uniprocessors, but for multiprocessors, we need new results that deal more directly with relevant metrics and realistic task characteristics. Of course, real-time system designers must still take the basic, available facts and apply them to their problems, which in many cases is a difficult engineering problem.

Acknowledgment

This work has been supported, in part, by NSF under grants IRI 9208920 and CDA 8922572, by ONR under grant N00014-92-J-1048, and by the IRI of Italy.

References

1. E.L. Lawler, "Recent Results in the Theory of Machine Scheduling," *Mathematical Programming: The State of the Art*, A. Bachen et al., eds., Springer-Verlag, New York, 1983, pp. 202-233.
2. J.R. Jackson, "Scheduling a Production Line to Minimize Maximum Tardiness," Research Report 43, Management Science Research Project, Univ. of Calif., Los Angeles, 1955.
3. J.K. Lenstra and A.H.G. Rinnooy Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," *Ann. Discrete Math.*, No. 5, 1977, pp. 287-326.
4. M.L. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes," *Information Processing 74*, North-Holland, 1974.
5. A.K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment," PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., May 1983.
6. C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *JACM*, Vol. 20, No. 1, 1973, pp. 46-61.
7. J. Leung and J. Whitehead, "On the Complexity of Fixed Priority Scheduling of Periodic, Real-Time Tasks," *Performance Evaluation*, Vol. 2, No. 4, 1982, pp. 237-250.
8. N. Audsley et al., "Hard Real-Time Scheduling: The Deadline Monotonic Approach," *Proc. IEEE Workshop on Real-Time Operating Systems*, 1992.
9. M.R. Garey et al., "Scheduling Unit-Time Tasks with Arbitrary Release Times and Deadlines," *SIAM J. Computing*, Vol. 10, No. 2, May 1981.

10. L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Computers*, Vol. 39, No. 9, Sept. 1990, pp. 1,175-1,185.
11. T.P. Baker, "Stack-Based Scheduling of Real-Time Processes," *J. Real-Time Systems*, Vol. 3, No. 1, Mar. 1991, pp. 67-99.
12. M. Chen and K. Lin, "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems," *J. Real-Time Systems*, Vol. 2, No. 4, Nov. 1990, pp. 325-340.
13. K. Ramamritham, J. Stankovic, and P. Shiah, "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems," *IEEE Trans. Parallel and Dist. Computing*, Vol. 1, No. 2, Apr. 1990, pp. 184-194.
14. C. Shen, K. Ramamritham, and J. Stankovic, "Resource Reclaiming in Multiprocessor Real-Time Systems," *IEEE Trans. Parallel and Dist. Computing*, Vol. 4, No. 4, Apr. 1993, pp. 382-397.
15. W. Zhao, K. Ramamritham, and J. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints," special issue on real-time systems, *IEEE Trans. Computers*, Vol. 36, No. 8, Aug. 1987, pp. 949-960.
16. C.D. Locke, "Best-Effort Decision Making for Real-Time Scheduling," PhD thesis, Computer Science Dept., Carnegie Mellon Univ., Pittsburgh, 1986.
17. W. Smith, "Various Optimizers for Single-Stage Production," *Naval Research Logistics Quarterly*, 3, 1956, pp. 59-66.
18. S. Cheng, J. Stankovic, and K. Ramamritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed, Hard Real-Time Systems," *Proc. Real-Time Systems Symp.*, IEEE CS Press, 1986, 166-174.
19. P. Thambidurai and K.S. Trivedi, "Transient Overloads in Fault-Tolerant Real-Time Systems," *Proc. Real-Time Systems Symp.*, IEEE CS Press, 1989, pp. 126-133.
20. J.R. Haritsa, M. Livny, and M.J. Carey, "Earliest Deadline Scheduling for Real-Time Database Systems," *Proc. Real-Time Systems Symp.*, IEEE CS Press, 1991, pp. 232-242.
21. S. Baruah et al, "On the Competitiveness of On-Line Real-Time Task Scheduling," *Proc. Real-Time Systems Symp.*, IEEE CS Press, 1991, pp. 106-115.
22. E.G. Coffman and R. Graham, "Optimal Scheduling for Two-Processor Systems," *ACTA Informat.*, 1, 1972, pp. 200-213.
23. R. Garey and D. Johnson, "Complexity Bounds for Multiprocessor Scheduling with Resource Constraints," *SIAM J. Computing*, Vol. 4, No. 3, 1975, pp. 187-200.
24. T.C. Hu, "Parallel Scheduling and Assembly Line Problems," *Operations Research*, 9, Nov. 1961, pp. 841-848.
25. J.D. Ullman, "Polynomial Complete Scheduling Problems," *Proc. Fourth Symp. Operating System Principles*, ACM, New York, 1973, pp. 96-101.
26. R. McNaughton, "Scheduling With Deadlines and Loss Functions," *Management Science*, Vol. 6, No. 1, Oct. 1959, pp. 1-12.
27. W. Horn, "Some Simple Scheduling Algorithms," *Naval Research Logistics Quarterly*, Vol. 21, 1974, pp. 177-185.
28. R. Graham, "Bounds on the Performance of Scheduling Algorithms," *Computer and Job Shop Scheduling Theory*, E.G. Coffman, ed., John Wiley and Sons, 1976, pp. 165-227.

***John A. Stankovic**, a professor in the Computer Science Department at the University of Massachusetts, Amherst, and an IEEE fellow, has held visiting positions at Carnegie Mellon University, at INRIA in France, and at Scuola Superiore S. Anna in Pisa, Italy. His current research interests include investigating various approaches to real-time scheduling, developing flexible, distributed, and fault tolerant, real-time operating systems, and developing and performing experimental studies on real-time active databases. He received the BS degree in electrical engineering and the MS and PhD degrees in computer science, all from Brown University, Providence, Rhode Island, in 1970, 1976, and 1979, respectively.*

***Marco Spuri** is a doctoral student at the Scuola Superiore di Studi Universitari e di Perfezionamento S. Anna of Pisa. His research interests include real-time computing, operating systems, and distributed systems. He received his Computer Science degree from the University of Pisa, Italy, in 1990. In the same year, he also received the diploma of the Scuola Normale Superiore of Pisa.*

***Marco Di Natale** is a PhD student at the Scuola Superiore di Studi Universitari e Perfezionamento S. Anna of Pisa. He spent the last year working in the Spring project at the University of Massachusetts at Amherst, doing research on real-time scheduling. His main research interests are in real-time and distributed systems and in programming and design tools He received his electronic engineering degree from the University of Pisa, Italy, in 1991*

***Giorgio C. Buttazzo** is an assistant professor of computer engineering at the Scuola Superiore S. Anna of Pisa. His research areas include real-time computing, advanced robotics, sensor-based control, and neural networks. He graduated in electronic engineering at the University of Pisa, Italy, in 1985. He then spent a year working on robotic artificial perception at the GRASP Laboratory of the University of Pennsylvania's Computer Science Department, where he received the MS degree in computer science. In 1988, he joined the Scuola Superiore S. Anna to work on real-time robot control architectures and obtain a PhD degree in robotics, which he received in 1991.*

Readers can contact Stankovic at the Computer Science Dept., University of Massachusetts, Amherst, MA 01003; his e-mail address is stankovic@cs.umass.edu.

Additional Information

A longer version of this article that also discusses precedence constrained scheduling and similarity of real-time scheduling to bin packing is available as Technical Report 95-23 (revised Jan. 1994), from the Dept. of Computer Science, University of Massachusetts, Amherst, MA 01003.