Implications of Classical Scheduling Results For Real-Time Systems*

John A. Stankovic, Marco Spuri, Marco Di Natale, and
Giorgio Buttazzo†
Scuola Superiore "S.Anna"
via Carducci, 40 - 56100 Pisa (Italy)

June 23, 1994

Abstract

Important classical scheduling theory results for real-time computing are identified. Implications of these results from the perspective of a real-time systems designer are discussed. Uni-processor and multiprocessor results are addressed as well as important issues such as future release times, precedence constraints, shared resources, task value, overloads, static versus dynamic scheduling, preemption versus non-preemption, multiprocessing anomalies, and metrics. Examples of what scheduling algorithms are used in actual applications are given.

KEYWORDS: scheduling theory, real-time, uniprocessor scheduling, multiprocessor scheduling

1 Introduction

Every real-time systems designer should be familiar with a set of important classical scheduling theory results, i.e., those results largely taken from the literature in complexity theory and operations research. While knowledge of these results rarely provides a direct solution for the designer, the implications of the results provide important insight in choosing a good design and scheduling algorithm for the system, and in avoiding very poor or even erroneous choices. The literature in scheduling theory is so vast, that we make no pretense at being comprehensive. In this paper, a minimum set of results, together with their implications, is presented. For example, the scheduling theory results presented include: 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 what these important results are, we want the reader to be able to answer, at least, the following questions:

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

†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.

- what do we really know about earliest deadline scheduling,
- what is known about uni-processor 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 usefulness of the result in a real-time computing system, and
- what different results exist for static and dynamic scheduling?

There are so many dimensions to the scheduling problem that there is no accepted taxonomy. In this paper we divide the scheduling theory between uni-processor (section 2) and multiprocessor (section 3) results. In the uni-processor section we begin with independent tasks, then consider precedence constraints, shared resources, and overload. In the multiprocessor case, since most results address precedence and shared resources together, we divide the work between static and dynamic algorithms.

2 Preliminaries

Before presenting the major scheduling results a few basic concepts must be clearly understood. Here we discuss the differences between static, dynamic, off-line and on-line scheduling as well as various metrics and their implications. NP-complete and NP-hard, terms used throughout the paper, are defined.

2.1 Static versus Dynamic Scheduling

Most classical scheduling theory deals with static scheduling. Static scheduling refers to the fact that the scheduling algorithm has complete knowledge regarding 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, real-time control of 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. In these types of real-time systems, the static scheduling algorithm operates on this set of tasks and produces a single schedule that is fixed for all time. Sometimes there is confusion regarding future release times. If all future release times are known when the algorithm is developing the schedule then it is still a static algorithm.

In contrast, a dynamic scheduling algorithm (in the context of this paper) has complete knowledge of the currently active set of tasks, but new arrivals may occur in the future, not known to the algorithm at the time it is scheduling the current set. The schedule therefore changes over time. Dynamic scheduling is required for real-time systems such as teams of robots cleaning up a chemical spill or in military command and control applications. As we will see in this paper very few theoretical results are known about real-time dynamic scheduling algorithms.

Off-line scheduling is often equated to static scheduling, but this is wrong. In building any 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, the 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 the assumptions remain true, all tasks will meet the deadlines. In other cases, the off-line analysis might produce a static set of priorities to use at run time. The schedule itself is not fixed, but the priorities that drive the schedule are fixed. This is common in the rate monotonic approach (to be discussed later).

If the real-time system is operating in a more dynamic environment, then it is not feasible to meet the assumptions of static scheduling (i.e., everything is known a priori). In this case an algorithm is chosen and analyzed off-line for the expected dynamic environmental conditions. Usually, less precise statements about the overall performance can be made. On-line, this same dynamic algorithm executes.

Generally, a scheduling algorithm (possibly with some modifications) can be applied to static scheduling or dynamic scheduling and used off-line or on-line. The important difference is what is known about the performance of the algorithm in each of these cases. As an example, consider earliest deadline first (EDF) scheduling. When applied to static scheduling we know that it is optimal in many situations (to be enumerated below), but when applied to dynamic scheduling on multiprocessors it is not optimal, in fact, it is known that no algorithm can be optimal.

2.2 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 in these results. When deadlines are considered, they are usually added as constraints, where, for example, one creates a minimum schedule length, subject to the constraint that all tasks must meet their respective deadline. If one or more tasks miss their deadlines, then 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. Using value is often overlooked in many real-time systems where the focus is simply on deadlines and not a combination of value and deadline. Minimizing schedule length has secondary importance in possibly helping minimize the resources required for a system, but 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 where resources can be continually added until the maximum lateness is less than or equal to zero. In this case no tasks miss their deadlines. On the other hand, the metric is not always useful because minimizing the maximum lateness doesn't necessarily prevent one, many, or even ALL tasks from missing their deadlines. See Figure 1.

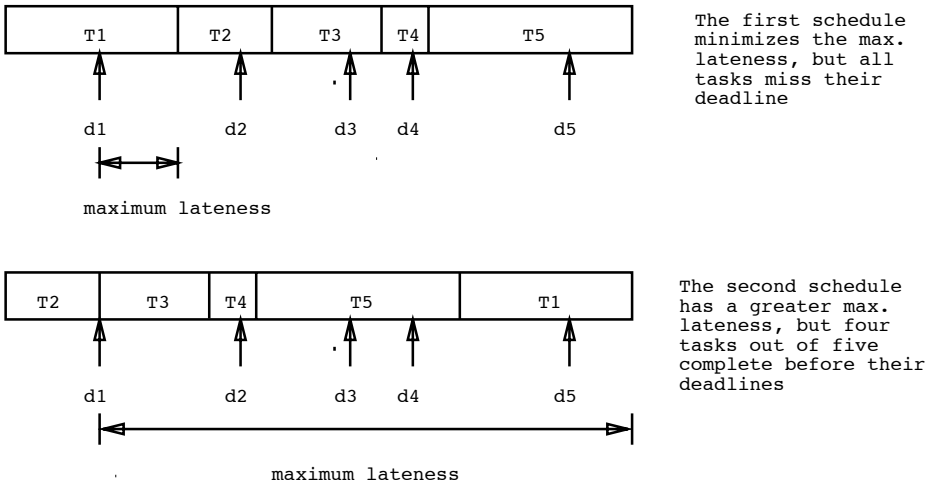

Figure 1: Minimizing Maximum Lateness Example

Rather than these above mentioned metrics much real-time computing work minimizes the number of tasks that miss deadlines or looks for optimal algorithms defined in the following manner: An optimal scheduling algorithm is one which may fail to meet a deadline only if no other scheduling algorithm can. In this paper, all of the above metrics will be mentioned, either because they are directly applicable to real-time systems, or to show where even though a nice theoretical result exists, there is limited applicability to real-time systems.

Related to metrics is the complexity of the various scheduling problems themselves. As we shall see, 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 $R \in NP$.

3 Uni-processor Systems

In general we follow the notation of [18], in which the problem definition has the form $\alpha \mid \beta \mid \gamma$, where α indicates the *machine environment* (in this section of the paper $\alpha = 1$, indicating a uni-processor machine), β indicates the *job characteristics* (preemptable, nonpreemptable, independent, precedence constrained, deadline, etc.) and γ indicates the *optimality criterion* (maximum lateness, total tardiness, etc.). Note that the optimality criterion depends on the metric chosen, which strongly relies on the system objectives and the task model.

3.1 Preemption vs NonPreemption: Jackson's Rule

Suppose there are n independent jobs (the words job, process and task will be used interchangeably just as they are throughout the scheduling literature), with each job j having a *processing time* p_j and a *due date* d_j. For any given sequence of scheduling, each job will have a defined *completion time* C_j too. Let us define the lateness of a job j 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

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

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 as follows:

Theorem 3.1 *(Jackson's Rule [16]). 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 [18], but presenting that argument here is beyond the scope of this paper. At first, this result may not seem too useful to a real-time systems designer because we often require that no task miss its deadline. But, since this is a static scheduling algorithm and if the maximum lateness is greater than zero, then the designer knows that he must increase the computing power of his system to meet the requirements of missing no deadlines. Further, as we shall see, EDD is optimal in many other situations also. 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 to consider 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 problem above extended with release times, that is

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

is NP-hard [19].

In this case we obtain a great benefit if we permit jobs to be preempted 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 the Jackson's rule, slightly modified in order to take the release times into account:

Theorem 3.2 *Any sequence that at any instant schedules the job with the earliest due date among all the eligible jobs (i.e., those whose release time is less then or equal to the current time) is optimal with respect to minimizing maximum lateness.* □

The result again can be easily proven by an interchange argument. The proof obtained in this way is very similar to the "time slice swapping" technique used in [9] and [24] to show the optimality of the *earliest deadline first* (EDF from now on) and the *least laxity first* (LLF) algorithms, respectively.

One implication of these results is that when practical considerations do not prevent us from using it, preemption usually gives greater benefit than nonpreemption in terms of scheduling complexity. Unfortunately, when we deal with shared resources in real-time systems we have to address critical sections and one technique is to create nonpreemptable code; this again creates an NP-hard problem.

Another implication of these theorems is that the minimization of maximum lateness implies optimality even when all deadlines must be met, because the maximum lateness can be required to be less than or equal to zero. In fact, the very well-known paper by Liu and Layland [21] focussed on this aspect of EDF scheduling for a set of independent periodic processes, showing that a full processor utilization is always achievable and giving a very simple necessary and sufficient condition for the schedulability of the tasks:

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

where T_j is the period of the 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. Recently, variations of EDF are being used in multimedia applications, robotics, and real-time databases. Note, however, that in none of the above classical results for EDF is precedence constraints, shared resources, or overloads taken into account. We address these aspects in subsequent sections.

Another very important and common area for real-time scheduling is the scheduling of periodic tasks. Here the rate monotonic algorithm is often used. This algorithm assigns to each task a static priority inversely proportional to its period, i.e., tasks with the shortest periods get the highest priority. For a fixed set of independent periodic tasks with deadlines the same as the periods, we know:

Theorem 3.3 *(Liu and Layland [21]) A set of n independent periodic jobs can be scheduled by the rate monotonic policy if* $\sum_{i=1}^{n} p_i/T_i \leq n \cdot (2^{1/n} - 1)$ *where* T_i *and* p_i *are the period and worst case execution time, respectively.*

For large n we obtain the utilization bound of 69% meaning that as long as the CPU utilization is less than 69% all tasks will make their deadlines. This is often referred to as the schedulability test. If deadlines of periodic tasks can be less than the period the above rule is no longer optimal. Rather we must use a deadline monotonic policy [20] 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 of which deals

with shared resources (see Section 3.3), and schedulability tests have been formulated for the deadline monotonic algorithm [1].

The rate monotonic scheduling algorithm has been chosen for the Space Station Freedom Project, the FAA Advanced Automation System (AAS), and has influenced the specification of the IEEE Futurebus+. The DoD's 1991 Software Technology Strategy says that the Rate Monotonic Scheduling is 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 the real-time control of a simple experiment that might contain 20 sensors whose data must be processed periodically, or a chemical plant that has a large number of periodic tasks and a few alarms. These alarms can be treated as periodic tasks whose minimum interarrival time is equal to its period, and then static scheduling, using the rate monotonic algorithm, can be applied.

3.2 Precedence Constraints

In many systems of practical interest we do not expect tasks to be independent, but rather cooperate to achieve the goal of the application. Cooperation among tasks is achieved by various types of communication semantics. Depending on the chosen semantics, application tasks experience precedence constraints or blocking, or both, while accessing shared resources. A precedence relation among tasks makes the scheduling problem more complex. Since not all tasks are ready to be scheduled at the same time, the simple EDF rule is no longer optimal.

In the following, precedence constraints will be expressed with the notation $i \rightarrow j$, or with their associated digraph G(V,E) where V is the set of tasks and E the set of edges, an edge connecting tasks i,j if task i precedes task j.

The simple scheduling problem of a set of tasks with no-preemption, identical arrival time and a precedence relation among them, described as,

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

was solved by Lawler [17] with an EDF–like algorithm that works backwards, starting from the leaf tasks in the precedence graph.

The algorithm works as follows: the scheduling list is built starting from the bottom in reverse topological order, and adding to the list on each step, the task having the minimum value for the chosen metric and whose successors have been scheduled. Lawler's algorithm is optimal, and runs in $O(n^2)$.

Lawler's algorithm gives a solution for tasks having identical start time. Unfortunately, this is not sufficient for all systems of practical interest where periodic tasks or dynamically arising tasks do not have a common start time. The problem of non-preemptive scheduling of jobs with different release times and general precedence constraints is not a simple one, in fact, the problem

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

and the corresponding

$$1 \mid prec, nopmtn, r_i \mid L_{\max}$$

were proven to be NP-hard by Lenstra [19].

The NP hardness of the general precedence constrained problem is a major obstacle for non-preemptive scheduling, in spite of the fact that optimal results or polynomial algorithms exist for similar problems, where some of the general assumptions are constrained. For example, a polynomial algorithm was found for unit computation time tasks and arbitrary precedence constraints.

The most interesting results related to precedence constraints are those obtained working on sub-classes of the general precedence relation. Polynomial algorithms have been found for precedence relations in the form of intrees, that is when every task has no more than one predecessor, or outtrees, when tasks have no more than one successor, or when the precedence relation is a series-parallel graph. It is easy to show how the intree and outtree cases are included in the more general class of series-parallel graphs. The series-parallel graph is the most interesting subset of the general precedence relation for which optimality results have been found. A series-parallel graph is defined recursively this way:

- G({j,},0) is a series - parallel graph
- if $G_1(V_1, A_1)$ and $G_2(V_2, A_2)$ are series-parallel graphs than
  $G_1 \rightarrow G_2 = (V_1 \cup V_2, A_1 \cup A_2 \cup (V_1 \times V_2))$ and
  $G_1 \mid\mid G_2 = (V_1 \cup V_2, A_1 \cup A_2)$ are series - parallel graphs

or alternatively, a graph is a series-parallel graph only if its transitive closure does not contain the Z graph. A Z graph is a graph that contains as a subgraph 4 nodes { i,j,k,l } with only the following edges

$$i \rightarrow j, \qquad i \rightarrow k, \qquad l \rightarrow k.$$

Figure 2 graphically depicts intrees and outtrees (series-parallel graphs) and a Z graph (not a series parallel graph). Efficient solutions exist for series-parallel graphs, but they do not exist for a Z graph. Unfortunately, Z graphs arise in practice. Details on these results follow.

Theorem 3.4 *(Lawler [18]) Given any set of tasks related by a series-parallel precedence graph an optimal solution exists for every cost function that admits a string interchange relation.* □

Formally, a cost function has a string interchange relation if, given two strings of jobs α and β and a quasi total order $\leq$ among them, the following relation holds:

$$f(\alpha) \leq f(\beta) \Rightarrow f(a..\alpha\beta..b) \leq f(a..\beta\alpha..b)$$

Intuitively, this formula means that a cost function admits a string interchange relation when a lower value is obtained when individual tasks of lower value are scheduled first. The theorem says that if we are interested in minimizing or maximizing a cost function that admits a

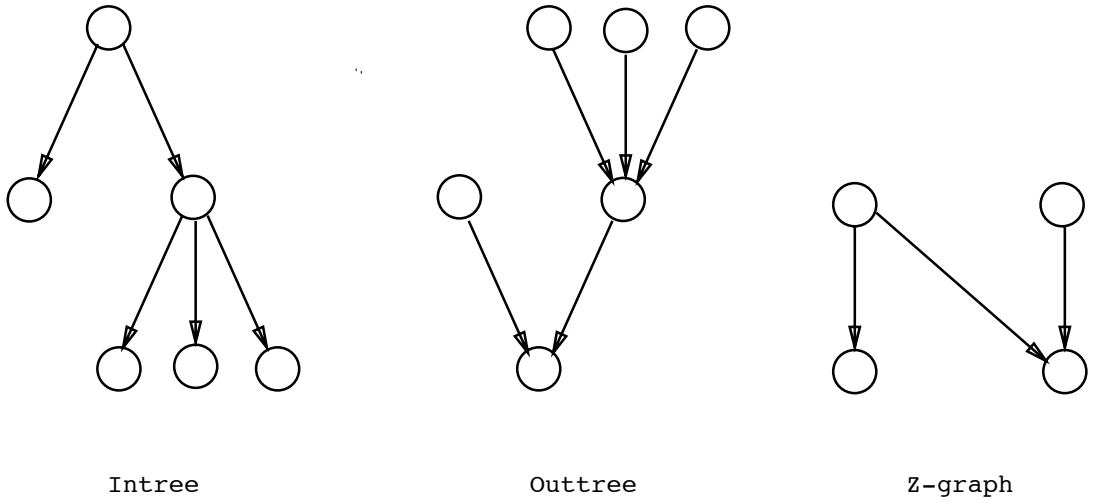

Figure 2: Precedence Relations

string interchange relation (e.g., minimizing lateness), it is possible to find an optimal schedule in polynomial time for every set of tasks related by a series-parallel precedence graph.

The algorithm which solves this problem works with a decomposition tree, that is the tree that shows how sub-graphs are connected by the series or parallel relation to form the global precedence graph. The decomposition tree can be found in $O(\mid N \mid + \mid A \mid)$ where N is the number of nodes and A the number of edges. The algorithm starts from the tasks having no successor in the decomposition tree, and, for every node, calculates a string sequence by combining the strings of jobs coming from the sons. The final node, representing the original graph, is reached when the whole optimal scheduling list has been computed.

A common feature of this algorithm, as is also found in other similar algorithms from the literature dealing with intrees or outtrees, is that they work on the precedence graph (or on the related decomposition tree), starting from jobs with no successors or no predecessors, and build a sequence of sub-optimal schedules. This technique can be useful in various scheduling heuristics.

To what extent can Lawler's optimal algorithm for series-parallel graphs, and even other optimal algorithms which work only on intrees or outtrees, help us in real-time systems? Unfortunately, some high level communication semantics found in programming languages, give rise to precedence constrained jobs with Z graphs, meaning that these optimal algorithms don't apply and heuristics need to be used. One example of how a Z graph arises is a simple pair of tasks linked by an asynchronous send with synchronous receive. See Figure 3. Note that remote procedure calls (RPC) do not give rise to Z graphs.

If preemption is allowed, classical results go further in providing solutions for general precedence constraints. Preemption reduces the complexity of the scheduling problem of precedence related tasks with different arrival times. The problem is, in fact, solvable in $O(n^2)$ by Baker's algorithm [2]

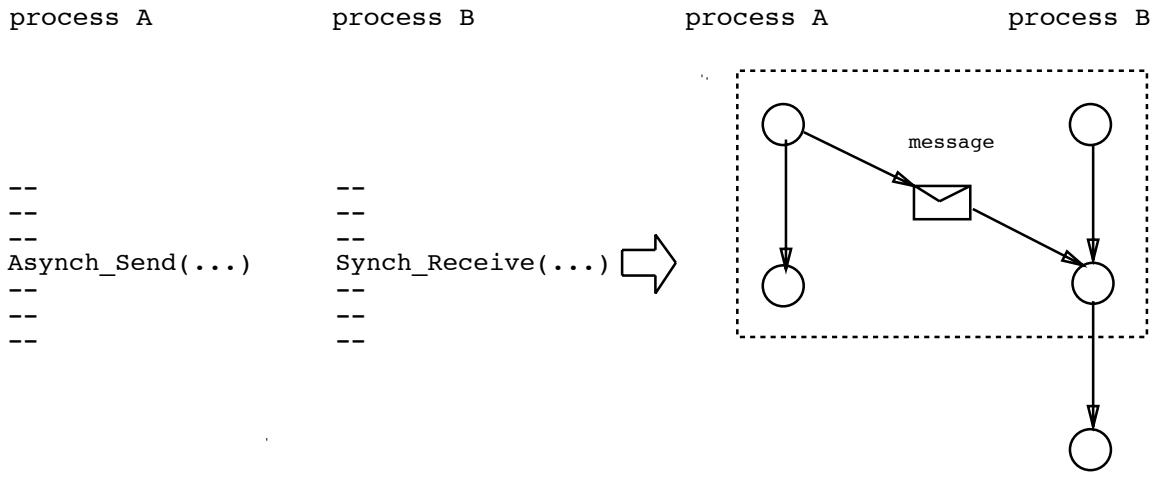

Figure 3: Program Example That Gives Rise to Z Graph

$$1 \mid prec, pmtn, r_i \mid L_{\max}.$$

Baker's procedure is recursive and because of its computational complexity it seems suited for off-line scheduling. Due to the difficulty of describing the algorithm and space limitations, we do not describe the algorithm here. However, an important feature of the algorithm is that the number of preemptions is limited to n-1 where n is the number of jobs, thus making the preemption overhead bounded. In all practical situations the scheduling and preemption overheads must be bounded and taken into account. We rarely see this issue addressed in classical scheduling theory.

In the above solutions, a scheduling list is explicitly created. Another technique is to encode the precedence relations into the parameters used by the scheduling algorithm, for example, into deadlines and release times.

Blazewicz [4] shows how to adjust deadlines so that precedence constraints are encoded in the deadlines *a priori*, and at run time you simply use EDF scheduling. His result comes from the fact that task deadlines depend on their deadlines and successors' deadlines, while task start times depend on their own start time and predecessors' start times. The theorem assumes no shared resources among tasks.

Theorem 3.5 *(Blazewicz [4]) EDF is optimal for tasks that have a general precedence relation and different release dates if deadlines and start times are revised according to the following formulas:*

$$d_i^* = \min\{d_i, \min(d_j^* - p_j; S_i \rightarrow S_j)\}$$

starting from the tasks having no successor and processing on every step those tasks whose successors have been processed

$$r_i^* = \max\{r_i, \max(r_j^* + p_j; S_j \rightarrow S_i)\}$$

starting from the tasks having no predecessor. □

This result allows us to transform a set of dependent tasks into a set of independent ones obtaining an equivalent problem under the EDF policy. The optimality of the technique of the revised deadlines and arrival dates has been used in both on-line [7] and off-line algorithms [24].

Unfortunately, the optimality of this technique is again lost if tasks with precedence constraints also share resources in an exclusive way. Moreover, if arbitrary protocols are used to access shared resources, the revision of tasks' deadlines and release times is no longer sufficient to guarantee the correct ordering of jobs without additional constraints. The general problem of scheduling a set of tasks with precedence constraints and arbitrary resource conflicts is NP-hard.

Some off-line algorithms face the NP hardness of the general problem trying to find acceptable solutions by means of heuristics, branch and bound techniques and so on. An example is given by the algorithm by Xu and Parnas [32] where on every step a sub-optimal schedule is obtained. There are even examples of on-line systems driven by heuristics as the Spring system [26] where the scheduling list is built on-line.

3.3 Shared Resources

Shared resources are commonly used in multitasking applications. While in general purpose systems this is a well-known problem solved, for example, with mutual exclusion primitives, in real-time systems a straightforward application of this solution does not hold. Defining a run-time scheduler as *totally on-line* if it has no knowledge about the future arrival times of the tasks, the following has been proven:

Theorem 3.6 *(Mok [24]). When there are mutual exclusion constraints, it is impossible to find a totally on-line optimal run-time scheduler.* □

The proof is simply given by an adversary argument. Furthermore, the same author showed a much more negative result:

Theorem 3.7 *(Mok [24]). The problem of deciding whether it is possible to schedule a set of periodic processes which use semaphores only to enforce mutual exclusion is NP-hard.* □

A transformation of the 3-partition problem to this scheduling problem is shown to prove the theorem.

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." A confirmation of this point of view is that the problem of minimizing the maximum lateness of n independent unit-time jobs with arbitrary release times, that is,

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

is easy [18]. Moreover, if we add precedence constraints and we want to minimize the maximum completion time (*makespan*), that is, we want to solve

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

the problem is still easy [11]. The algorithm that solves it makes use of *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 late task to miss its deadline.

At this point several choices are possible. One of them, followed by Mok, is to enforce the use of mutually exclusive scheduling blocks having the same computation time, and another, followed, for example, by Sha *et al.* [27] and Baker [2], is to efficiently find a suboptimal solution with a clever allocation policy, guaranteeing at the same time a minimum level of performance.

The former solution is called *Kernelized Monitor*. The key idea 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 i-th critical section. In other words the grain of the system is made coarser. Furthermore, the ready times and the deadlines of the tasks can be previously modified according to some partial order on the tasks. Adjusting the EDF scheduler with the technique of the forbidden regions mentioned above, the following theorem can be proven:

Theorem 3.8 *(Mok [24]). 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.* □

In [27] Sha *et al.* introduce the *Priority Ceiling Protocol* (PCP), an allocation policy for shared resources which works with a Rate Monotonic scheduler. Successively Chen and Lin [5] extend the utilization of the protocol to an EDF scheduler.

The main goal of this, as other similar protocols, is to bound the usually uncontrolled priority inversion, a situation in which a higher priority job is blocked by lower priority jobs for an indefinite period of time (recall that a block can occur if a job tries to enter a critical section already locked by some other job). Finding a bound to priority inversion allows to evaluate the worst case blocking times eventually experienced by the jobs, so that they can be accounted for in the schedulability guaranteeing formulas. In other words this means to evaluate the worst case loss of performance.

The key ideas behind the PCP is to prevent multiple priority inversions by means of early blocking of tasks that could cause priority inversion, and to minimize as much as possible the length of the same priority inversion allowing a temporary rise of the priority of the blocking task. This is done in the following way: define the *ceiling* of a critical section as the priority of the highest priority task that currently locks or could lock the critical section, and allow the locking of a critical section only if the priority of the requesting task is higher than the ceiling of all critical sections currently locked. In case of blocking, the task that holds the lock inherits the priority of the requesting task until it leaves the critical section.

The following properties have been shown:

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

Of course, the former property is used to evaluate the worst case blocking times of the jobs.

In [2] Baker describes a similar protocol, the Stack Resource Policy (SRP), that handles a more general situation in which multiunit resources, both static and dynamic priority schemes, and sharing of runtime stacks are all allowed. The protocol relies on the following 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 inversion, 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 key idea behind this protocol is that when a job needs a resource not available, it is blocked at the time it attempts to preempt, rather than later, when it actually may need the shared resource. The main advantages of this earlier blocking are to save unnecessary context switches and the possibility of a simple and efficient implementation of the SRP by means of a stack.

In summary, dealing with shared resources in a real-time system is of utmost importance. The classical results given in this section provide a good means for handling resources in a uniprocessor. Many researchers feel that these techniques do not work well in multiprocessors nor in distributed systems. For such systems shared resources are typically addressed by on-line planning algorithms [26, 28, 33], or by static schedules developed with off-line heuristics. Both of these alternative approaches *avoid* blocking over shared resources by scheduling competing tasks at different points in time.

3.4 Overload and Value

EDF and LLF algorithms have been shown to be optimal with respect to different metrics. However, in overload conditions, these algorithms perform very poorly. Experiments carried out by Locke [22] and others have shown that both EDF and LLF rapidly degrade their performance during overload intervals. This is due to the fact that such algorithms give the highest priority to those 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.

In order to gain control over the tardy tasks in overload conditions, a value is usually associated with each task, reflecting the importance of that task within the set. When dealing with task sets with values, tasks can be scheduled by the Smith' rule.

Theorem 3.9 *(Smith's rule [29]) Finding an optimal schedule for*

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

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

Smith's rule resembles the common shortest processing time first (SPT) rule and is equivalent to it 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 [19] 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 had been found for particular kind of precedence relations, in fact, optimal polynomial algorithm had been found for the problems

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

$$1 \mid series - parallel \mid \sum C_j$$

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

Unfortunately, in real-time systems the precedence constraints imposed on tasks are often more general. A heuristic was proposed in the Spring project, where deadline and cost driven algorithms are combined together with rules to dynamically revise values and deadlines in accordance with the precedence relations [6].

A number of heuristic algorithms have also been proposed to deal with overloads [30] [13] which improve the performance of EDF.

Baruah, et al. [3] have shown that there exists an upper bound on the performance of any on-line (preemptive) algorithm working in overload conditions. The "goodness" of an on-line algorithm is measured 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. According to this metric, they proved the following theorem:

Theorem 3.10 *(Baruah, et. al. [3]) There does not exist an on-line scheduling algorithm with a competitive factor greater than 0.25.*

What the theorem says is that no on–line scheduling algorithm can guarantee a cumulative value greater than $1/4th$ 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 then the bound is 1, as the load just surpasses 1 then the bound drops immediately to .385, for loads from greater than 1 up to 2 the bound gradually drops from .385 to .25, and then for all loads greater than 2 the bound is .25.

It is worth pointing out that the above bound is achieved under very restrictive assumptions, such as all tasks in the set have zero laxity, the overload can have an arbitrary (but finite) duration, task's execution time can be arbitrarily small, and task value is equal to computation time. Since in most real world applications tasks characteristics are much less restrictive, the $1/4th$ bound has only a theoretical validity and more work is needed to derive other bounds based on more knowledge of the task set.

3.5 Summary of Uni-processor Results

Many basic algorithms and theoretical results have been developed for scheduling on uni-processors. Many of these are based on earliest deadline scheduling or rate monotonic scheduling. Extensions of these results to handle precedence and resource sharing have occurred. Because of this work, designers of real-time systems have a wealth of information concerning uni-processor scheduling. What is still required are more results on scheduling in overload and for fault tolerance (although fault tolerance usually requires multiple processors as well). It is also necessary to develop a more integrated and comprehensive scheduling approach that addresses periodic and aperiodic tasks, preemptive and non-preemptive tasks in the same system, tasks with values, and combined CPU and I/O scheduling, to name a few issues. As an example, the operational flight program of the A-7E aircraft has 75 periodic and 172 aperiodic processes with significant synchronization requirements. Extensions to rate monotonic that integrate periodic and aperiodic tasks could be used for such an application.

4 Multi-processor Real-Time Scheduling

More and more real-time systems are relying on multiprocessors. Unfortunately, less is known about how to schedule multiprocessor based real-time systems than for uni-processors. This is partly due to the fact that complexity results show that almost all real-time multiprocessing scheduling is NP-hard, and partly due to the minimal actual experience that exists with such systems so even the number of heuristics that exist is relatively low. In spite of the negative implications that complexity analysis provides, it is important to understand these complexity results because

- 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 upon which to base such heuristics,

- fundamental limitations of on-line algorithms must be understood to better create robust systems and to avoid operating under misconceptions, and
- serious scheduling anomalies can be avoided.

In this section we present multiprocessing scheduling results for deterministic (static) scheduling both with and without preemption, for dynamic on-line scheduling with and without preemption, identify various anomalies, and briefly discuss the similarity of this problem to bin packing. Important implications of the theory are stressed throughout the section and a summary of the global picture of multiprocessor real-time scheduling is given.

4.1 Deterministic (Static) Scheduling

4.1.1 Non-preemptive Multiprocessing Results

Let our model of multiprocessing be that there are a set of P processors, T tasks, and R resources. The processors are identical. Each task has a worst case execution time of τ, is non-preemptive, and tasks may be related by a partial order indicating that, e.g., task T(i) must complete before task T(j). It is important to note that in most of the scheduling theory results, tasks are considered to have constant execution time. For most computer applications tasks never have constant execution time so we must understand the implication of this fact. For example, this fact gives rise to one of the interesting multiprocessing anomalies of real-time scheduling (see section 4.3). For each resource R(k) there is a number which indicates how much of it exists. 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% of the resource. The complexity results from deterministic scheduling theory for multiprocessing where tasks are non-preemptive, have a partial order among themselves, have resource constraints (even a single resource constraint), and have a single deadline show that almost all 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 following theorems is the amount of computation time required for determining a schedule which satisfies the partial order and resource constraints, and completes all required processing before a given fixed deadline.

Theorem 4.1 *(Coffman and Graham [8]). The multiprocessor scheduling problem with 2 processors, no resources, arbitrary partial order relations, and every task has unit computation time is polynomial.* □

Theorem 4.2 *(Garey and Johnson [10]). The multiprocessor scheduling problem with 2 processors, no resources, independent tasks, and arbitrary computation times is NP-complete.* □

Theorem 4.3 *(Garey and Johnson [10]). The multiprocessor scheduling problem with 2 processors, no resources, arbitrary partial order, and task computation times are either 1 or 2 units of time is NP-complete.* □

Proc.	Res.	Ordering	Comp T.	Complexity
2	0	Arbitrary	Unit	Polynomial
2	0	Independ.	Arbitrary	NP-Comp
2	0	Arbitrary	1 or 2 Units	NP-Comp
2	1	Forest	Unit	NP-Comp
3	1	Independ.	Unit	NP-Comp
N	0	Forest	Unit	Polynomial
N	0	Arbitrary	Unit	NP-Comp

Table 1: Summary of Basic Multiprocessor Scheduling Theorems

Theorem 4.4 *(Garey and Johnson [10]). The multiprocessor scheduling problem with 2 processors, 1 resource, a forest partial order, and each computation time of every task equal to 1 is NP-complete.* □

Theorem 4.5 *(Garey and Johnson [10]). The multiprocessor scheduling problem with 3 or more processors, one resource, all independent tasks, and each tasks computation time equal to 1 is NP-complete.* □

Theorem 4.6 *(Hu [15]). The multiprocessor scheduling problem with n processors, no resources, a forest partial order, and each task having a unit computation time is polynomial.* □

Theorem 4.7 *(Ullman [31]). 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 non-preemptive multiprocessing scheduling almost all problems are NP-complete implying that heuristics must be used for such problems. Basically, we see that non-uniform task computation time and resource requirements cause NP-completeness immediately. An implication of these results is that designs which use only local resources (such as object based systems and functional language based systems) and schedule based on a unit time slot 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 up a process into unit times creates difficult maintenance problems and possibly wasted processing cycles when tasks consume less than the allocated unit of time. Note that the above results refer to a single deadline for all tasks. If each task has a deadline the problem is exacerbated.

4.1.2 Preemptive Multiprocessing Real-Time Scheduling

It is generally true that if the tasks to be scheduled are preemptable, then the scheduling problem is easier, but in certain situations there is no advantage to preemption. The following classical results pertain to multiprocessing scheduling where tasks are preemptable, i.e.,

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

Theorem 4.8 *(McNaughton [23]). 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.* □

So here we see an example, for a given metric, that 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, then the shortest processing time first greedy approach solves the problem and is not NP. Here again, there is no advantage to preemption. This result can have an important implication when creating a static schedule; we certainly prefer to minimize preemption for practical reasons at run time, so knowing that there is no advantage to preemption, a designer would not create a static schedule with any preemptions.

Theorem 4.9 *(Lawler [18]). The multiprocessing problem of scheduling P processors, with task preemption allowed and where we try to minimize the number of late tasks is NP-hard.* □

This theorem indicates that one of the most common forms of real-time multiprocessing scheduling, i.e.,

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

where U_j are the late tasks, requires heuristics.

4.2 Dynamic Multiprocessor Scheduling

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

First, consider that under certain conditions in a uni-processor, dynamic earliest deadline scheduling is optimal. Is this algorithm optimal in a multiprocessor? The answer is no.

Theorem 4.10 *(Mok [24]). Earliest deadline scheduling is not optimal in the multiprocessor case.* □

To illustrate why this is true consider the following example. We have 3 tasks to execute on 2 processors. The task characteristics are given by task-number(computation time, deadline): $T_1(1,1)$, $T_2(1,2)$, and $T_3(3,3.5)$. Scheduling by earliest deadline would execute T_1 on P1 and T_2 on P2 and then T_3 misses its deadline. However, if we schedule T_3 first, on P1, 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 [14], but this algorithm is too complicated to present here.

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 4.11 *(Mok [24]). For two or more processors, no deadline scheduling algorithm can be optimal without complete a priori knowledge of 1) deadlines, 2) computation times, and 3) start times of the tasks.* □

This implies that any of the classical scheduling theory algorithms which requires knowledge of start times can not be optimal if used on-line. This also points out that we cannot hope to develop an optimal on-line algorithm for the general case. 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 run time even if the arrival times are different because those later arrivals can't make conditions any worse. When such a worst case analysis approach is not possible for a given system, usually because such sufficient conditions cannot be developed or because ensuring such conditions are too costly, more probabilistic approaches are needed. A number of good heuristics exist for dynamic multiprocessor scheduling and we are beginning to see much needed 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 non-optimally, it might be able to provide a minimum level of guaranteed performance.

As mentioned, various heuristics exist for real-time multiprocessor scheduling with resource constraints [26]. However, in general, these heuristics use a non-preemptive model. The advantages of a non-preemptive model are few context switches, higher understandability and easier testing than for the preemptive model, and avoidance of blocking is possible. The main disadvantage of the non-preemptive model is the (usually) less efficient utilization of the processor. Heuristics also exist for a preemptive model [33]. The advantages of a preemptive model are high utilizations and low latency at reacting to newly invoked work. The disadvantages are many context switches, difficulty in understanding the run time execution and its testing, and blocking is common. All these heuristics, whether for the preemptive or non-preemptive cases, are fairly expensive in terms of absolute on-line computation time compared to very simple algorithms such as EDF, so this 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, and if the task meets its deadline then a value equal to the execution time is obtained, else no value is obtained. Let the system operate in both normal and overload conditions. Let there be 2 processors.

Theorem 4.12 *(Baruah, et. al. [3]). No on-line scheduling algorithm can guarantee a cumulative value greater than one-half for the dual processor case.* □

As for the bounds results for the uni-processor case (presented in Section 3.4), the implications of this theorem are very pessimistic. As before, some of the pessimism arises because of the assumptions made concerning the lack of knowledge of the task set. In reality, we do have significant knowledge (such as 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 know the minimum laxity of any task in the system is greater than some value). If we can exploit this knowledge, then the bounds may not be so pessimistic. We require more algorithms that directly address the performance of a multiprocessing system in overload conditions.

4.3 Multiprocessing Anomalies

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

Theorem 4.13 *(Graham [12]). 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.* □

An implication of this result means that if tasks have deadlines, then the accompanying increase in schedule length due to the anomaly can cause a previously valid schedule to become invalid, i.e., tasks can now miss deadlines. It is initially counter intuitive to think that adding resources such as an extra processor, or relaxing constraints such as less precedence among tasks, or less execution time requirements can make things worse. But, this is 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 $T1$ on the first processor. This means that the second task $T2$ 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, where had $T2$ 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 4.

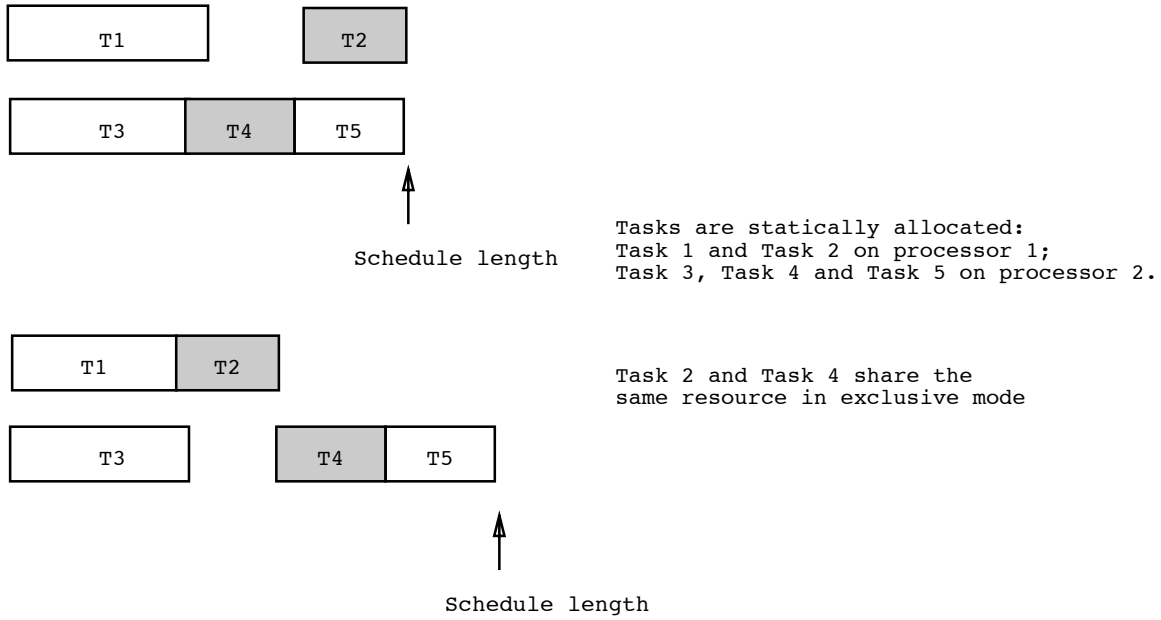

Figure 4: One Example of Richard's Anomalies

It is especially important to 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 often be very inefficient. However, algorithms such as [28] strive to reclaim this idle time, but carefully address the anomalies so that they will not occur.

4.4 Similarity to Bin Packing

Another tremendously active area of scheduling research is in bin packing algorithms. Each bin (processor) has a maximum capacity and boxes (jobs or tasks) placed in the bins require some percentage of the capacity. The goal is either, given a fixed number of bins, pack them with jobs so as to minimize the maximum length of any bin, or rather fill the bins to capacity minimizing the number of bins required. The bins are the computers of a multiprocessor which provide a computing capacity up to the deadline of the set of jobs. Jobs require some amount of processing time. In real-time scheduling it is usually assumed that memory requirements are implicitly met. The common algorithms are best fit (BF), first fit (FF), first fit decreasing (FFD), and best fit decreasing (BFD). The latter two algorithms arrange the list of jobs into a nonincreasing list with respect to capacity requirements, and then apply first fit or best fit, respectively. Theoretical bounds exist to describe, e.g., the minimum number of bins required. The worst case bounds for FF and BF for large task sets are $(17/10)L*$ where $L*$ is the optimal (minimum) number of bins [12]. For FFD the bound is $(11/9)L*$ and it is known that the bound of BFD is less than or equal to the FFD bound [12]. This work is of limited value for real-time systems since we have only a single deadline and other issues such as precedence constraints and other real considerations are not taken into account. However, some useful implications are

- we can know about the worst case and avoid it by design,
- we can obtain an estimate on the number of processors required for our application, and
- since average behavior is also important and since we are doing this analysis off-line, if good packing is not achieved then we can permute the packing using average case information, put constraints on job sizes, etc. Bin packing results should be extended and incorporated into real-time design tools.

4.5 Summary of Multiprocessor Results

Most multiprocessor scheduling problems are NP, but for deterministic scheduling this is not a major problem because either the specific problem is not NP-complete and we can use a polynomial algorithm and develop an optimal schedule, or we can use off-line heuristic search techniques based on what classical theory implies. These off-line techniques usually only have to find 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. We must point out, however, good tools with implemented heuristics are still necessary and many extensions that treat more sophisticated sets of task 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 [28]. Better results for operation in overloads, better bounds which account for typical *a priori* knowledge found in real-time systems, and algorithms which can guarantee various levels of performance are required. Dynamic multiprocessing scheduling is in its infancy.

5 Conclusion

Classical scheduling theory provides a basic set of results of use to real-time systems designers. Many results are known for uni-processors and very few for multi-processors. Complexity, fundamental limits, and performance bounds for important scheduling problems are known. Anomalies that must be avoided have been identified. It is still necessary for real-time designers to take these basic facts and apply them to their problem – a difficult engineering problem in many cases. Many new results are needed that deal more directly with metrics of interest to real-time applications and with more realistic task set characteristics than is typical for much of the theory presented here.

Many issues are outside the scope of this paper including distributed scheduling, integration of cpu scheduling with communication scheduling, with I/O scheduling, groups of tasks with a single deadline, placement constraints and the impact of this placement on the run time scheduling, fault tolerance needs, other kinds of timing requirements besides simple deadlines and periods, integration of critical and non-critical tasks, and the interaction of scheduling algorithms with the system design and implementation including run time overhead. Most of these areas are wide open areas for research.

References

[1] N. Audsley, A. Burns, M. Richardson, and A. Wellings, "Hard Real-Time Scheduling: The Deadline Monotonic Approach," *IEEE Workshop on Real-Time Operating Systems*, 1992.

[2] T.P. Baker, "Stack-Based Scheduling of Real-time Processes," *Journal of Real-Time Systems*, 3, 1991.

[3] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang, "On the Competitiveness of On-Line Real-Time Task Scheduling," *Proceedings of Real-Time Systems Symposium*, December 1991.

[4] J. Blazewicz, "Scheduling Dependent Tasks with Different Arrival Times to Meet Deadlines," In E. Gelenbe, H. Beilner (eds), Modeling and Performance Evaluation of Computer Systems, Amsterdam, North-Holland, 1976.

[5] M. Chen and K. Lin, "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems," *Journal of Real-Time Systems*, 2, 1990.

[6] S. Cheng, J. Stankovic, and K. Ramamritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems, " *Real-Time Systems Symposium*, December 1986.

[7] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic Scheduling of Real-Time Tasks Under Precedence Constraints," *Real-Time Systems Journal*, 2, 1990.

[8] E.G. Coffman and R. Graham, "Optimal Scheduling for Two-Processor Systems," *ACTA Informat.*, 1, 1972.

[9] M.L. Dertouzos, "Control Robotics: the Procedural Control of Physical Processes," *Information Processing 74*, North-Holland Publishing Company, 1974.

[10] R. Garey and D. Johnson, "Complexity Results for Multiprocessor Scheduling Under Resource Constraints," *SIAM Journal of Computing*, 1975.

[11] M.R. Garey, D.S. Johnson, B.B. Simons, and R.E. Tarjan, "Scheduling Unit-Time Tasks with Arbitrary Release Times and Deadlines," *SIAM Journal Comput.*, 10(2), May 1981.

[12] R. Graham, Bounds on the Performance of Scheduling Algorithms, chapter in *Computer and Job Shop Scheduling Theory*, John Wiley and Sons, pp. 165-227, 1976.

[13] J. R. Haritsa, M. Livny, and M. J. Carey, "Earliest Deadline Scheduling for Real-Time Database Systems," *Proceedings of Real-Time Systems Symposium*, December 1991.

[14] W. Horn, "Some Simple Scheduling Algorithms," *Naval Research Logistics Quarterly*, 21, pp. 177-185, 1974.

[15] T. C. Hu, "Parallel Scheduling and Assembly Line Problems," *Operations Research*, 9, 1961.

[16] J.R. Jackson, "Scheduling a Production Line to Minimize Maximum Tardiness," Research Report 43, Management Science Research Project, University of California, Los Angeles, 1955.

[17] E.L. Lawler, "Optimal Sequencing of a Single Machine Subject to Precedence Constraints," Management Science, 19, 1973.

[18] 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.

[19] J.K. Lenstra and A.H.G. Rinnooy Kan "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," Ann. Discrete Math. 5, pp. 287-326, 1977.

[20] J. Leung and J. Whitehead, "On the Complexity of Fixed Priority Scheduling of Periodic, Real-Time Tasks," *Performance Evaluation*, 2(4), pp. 237-250, 1982.

[21] C.L. Liu, J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, 20(1), 1973.

[22] C. D. Locke, "Best-effort Decision Making for Real-Time Scheduling," PhD thesis, Computer Science Department, Carnegie-Mellon University, 1986.

[23] R. McNaughton, "Scheduling With Deadlines and Loss Functions," *Management Science*, 6, pp. 1-12, 1959.

[24] A.K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment," Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1983.

[25] J. Moore, "An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs," *Management Science*, Vol. 15, No. 1, pp. 102-109, September 1968.

[26] K. Ramamritham, J. Stankovic, and P. Shiah, "Efficient Scheduling Algorithms For Real-Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Computing*, Vol. 1, No. 2, pp. 184-194, April 1990.

[27] L. Sha, R. Rajkumar, J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, 39(9), 1990.

[28] C. Shen, K. Ramamritham, and J. Stankovic, "Resource Reclaiming in Multiprocessor Real-Time Systems," *IEEE Transactions on Parallel and Distributed Computing*, Vol. 4, No. 4, April 1993.

[29] W. Smith, "Various Optimizers for Single Stage Production," *Naval Research Logistics Quarterly*, 3, pp. 59-66, 1956.

[30] P. Thambidurai and K. S. Trivedi, "Transient Overloads in Fault-Tolerant Real-Time Systems," *Proceedings of Real-Time Systems Symposium*, December 1989.

[31] J. D. Ullman, "Polynomial Complete Scheduling Problems," *Proc. 4th Symp. on Operating System Principles*, 1973.

[32] J. Xu and D. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, pp.360-369, March 1990.

[33] W. Zhao, K. Ramamritham, and J. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints," Special Issue of *IEEE Transactions on Computers* on Real-Time Systems, Vol. C-36, No. 8, pp. 949-960, August 1987.