On Satisfying Timing Constraints in Hard-Real-Time Systems

Jia Xu *
Department of Computer Science
York University
North York, Ontario, Canada

David Lorge Parnas †
Department of Electrical and Computer Engineering
McMaster University
Hamilton, Ontario, Canada

Abstract

We explain why pre-run-time scheduling is essential if we wish to guarantee that timing constraints will be satisfied in a large complex hard-real-time system. We examine some of the major concerns in pre-run-time scheduling and consider what formulations of mathematical scheduling problems can be used to address those concerns. A purpose of this paper is to provide a guide to the available algorithms.

1 Introduction

There are many computer applications in which computations must satisfy stringent timing constraints, that is, one must guarantee that those computations must be completed before specified deadlines. Failure to meet the specified deadlines in such applications can lead to intolerable system degradation, and can, in some applications, result in catastrophic loss of life or property.

Such computer systems are called "hard-real-time" systems. Many safety-critical systems are hard-real-time systems. Examples of hard-real-time systems include embedded tactical systems for military applications, flight mission control, traffic control, production control, robotics, etc.

Currently many safety-critical hard-real-time systems are built using methods which do not provide any guarantee that critical timing constraints will be met. As a result, timing errors where computations miss their deadlines are the most unpredictable, most persistent, most costly, and most difficult to detect and correct type of errors in safety-critical hard-real-time software.

In this paper, we discuss a general technique for solving the problem of satisfying timing constraints in hard-real-time systems – pre-run-time scheduling.

First we point out the fundamental problems with the run-time scheduling approach with respect to satisfying timing constraints. We provide a critical analysis of current standard practices in hard-real-time systems design such as assigning static priorities to processes, the use of complex run-time mechanisms for process synchronization and mutual exclusion, allowing events to interrupt processes and occupy system resources at random times, and the reliance on stochastic simulations to "verify" system design, etc. In particular, we explain why they fail to guarantee that timing constraints will be satisfied. Second we examine the major concerns in pre-run-time scheduling and discuss what formulations of mathematical scheduling problems can be used to address those concerns. We then examine the characteristics of scheduling problems and algorithms that have been discussed in the literature and explain how they relate to those concerns.

Throughout this paper, it is assumed that the hard-real-time system includes a relatively large number of processes that have stringent timing constraints, that the processes have different timing characteristics and interdependencies, and that the processor utilization factor is not very low, i.e., there is not much spare CPU capacity available. The U.S. Navy's A-7E aircraft operational flight program is an example [FaPa88].

It is also assumed that in the hard-real-time system, one must guarantee *a priori* that *all* stringent timing constraints will be satisfied. We distinguish such systems from systems that may accept or reject a request for execution of a task, and only guarantee that timing constraints will be satisfied for those that are accepted.

2 Characteristics of processes and pre-run-time scheduling versus run-time scheduling

Tasks in hard-real-time systems can be designed as a set of cooperating sequential processes [Dijk68]. A *process* consists of operations that must be executed in a prescribed order. Processes can be divided into those that have *hard* deadlines, i.e., meeting their deadlines is critical to the system's operation; and those which have *soft* deadlines, i.e., although a shorter response time is desirable, occasionally missing a deadline can be tolerated. In this paper, we are mainly concerned with complex hard-real-time systems, those in which the bulk of the computation is performed by processes that have hard deadlines.

In hard-real-time systems there are two types of processes: *periodic processes* and *asynchronous processes.*

A periodic process consists of a computation that is executed repeatedly, once in each fixed period of time. A typical use of periodic processes is to read sensor data and update the current state internal variables and outputs.

An asynchronous process consists of a computation that responds to internal or external events. A typical use of an asynchronous process is to respond to operator requests.

*†This work was supported in part by Natural Sciences and Engineering Research Council of Canada grants to J. Xu and D. L. Parnas.

We have examined a number of hard-real-time applications and it appears that in most of them periodic processes constitute the bulk of the computation. Usually, periodic processes have hard deadlines and represent the most important computations, whereas asynchronous processes are relatively few in number and usually have very short computation times.

We believe that this is because most hard-real-time applications are concerned with simulating continuous behavior, and the simulation of continuous behavior requires many computationally intensive periodic actions. In addition, information about most external or internal events can be buffered until it can be handled by periodic processes.

A periodic process p can be described by a quadruple (r_p, c_p, d_p, prd_p). prd_p is the *period*, c_p is the worse case *computation time* required by process p, d_p is the *deadline*, i.e., the duration of the time interval between the beginning of a period and the time by which an execution of process p must be completed in each period. r_p is the *release time*, i.e., the duration of the time interval between the beginning of a period and the earliest time that an execution of process p can be started in each period. We assume that r_p, c_p, d_p, prd_p as well as any other parameters expressed in time have integer values. A periodic process p can have an infinite number of *periodic process executions* p_0, p_1, p_2, ..., with one process execution for each period. For the ith process execution p_i corresponding to the ith period, p_i's release time is $r_{p_i} = r_p + prd_p \times (i-1)$; and p_i's deadline is $d_{p_i} = d_p + prd_p \times (i-1)$; For an example of a periodic process see Fig. 1.

Although the precise request times for executions of an asynchronous process a are not known in advance, usually the *minimum amount of time between two consecutive requests* min_a is known in advance.

An asynchronous process a can be described by a triple (c_a, d_a, min_a). c_a is the worse case *computation time* required by process a, d_a is the *deadline*, i.e., the duration of the time interval between the time when a request is made for process a and the time by which an execution of process a must be completed. An asynchronous process a can have an infinite number of *asynchronous process executions* a_0, a_1, a_2, ..., with one process execution for each asynchronous request. For the ith asynchronous process execution a_i which corresponds to the ith request, if a_i's request time is r_{a_i}, then a_i's deadline is $d_{a_i} = r_{a_i} + d_a$; For an example of an asynchronous process see Fig. 2.

We introduce the notions *process execution unit* and *processor time unit.* If a periodic process p or an asynchronous process a has a computation time of c_p or c_a, then we assume that that process execution p_i or a_i is composed of c_p or c_a process execution units. We assume that each processor is associated with a *processor time axis.* Each processor time axis starts from 0 and is divided into a sequence of processor time units.

A *schedule* is a mapping from a possibly infinite set of process execution units to a possibly infinite set of processor time units on one or more processor time axes. The number of processor time units between 0 and the processor time unit that is mapped to by the first unit in a process execution is called the *start time* of that process execution, and the number of time units between 0 and the time unit subsequent to the processor time unit mapped to by the last unit in a process execution is called the *completion time* of that process execution. A *feasible schedule* is a schedule in which the start time of every process execution is greater than or equal to that process execution's release time or request time, and its completion time is less than or equal to that process execution's deadline. For an example of a feasible schedule see Fig. 3.

Each process p may consist of a finite sequence of *segments*[1] $p[0], p[1], ..., p[n[p]]$, where $p[0]$ is the first segment and $p[n[p]]$ is the last segment in process p. Given the release time r_p and deadline d_p of process p and the computation time of each segment $p[i]$ in process p, one can easily compute the release time and deadline for each segment. See Fig 8.

Various types of relations such as *precedence* relations and *exclusion* relations may exist between ordered pairs of processes segments. A process segment i is said to *precede* another process segment j if j can only start execution after i has completed its computation. Precedence relations may exist between process segments when some process segments require information that is produced by other process segments. A process segment i is said to *exclude* another process segment j if no execution of j can occur between the time that i starts its computation and the time that i completes its computation. Exclusion relations may exist between process segments when some process segments must prevent simultaneous access to shared resources such as data, I/O devices by other process segments.

Almost all mathematical scheduling problems previously studied in the literature assume one of two special cases of exclusion relations. The first special case is when no exclusion relations exist between any pair of process segments, that is, any process can be preempted by any other process at any time. When this assumption is made the schedule constructed by an algorithm that solves the problem is called a *preemptive* schedule. Fig. 3 shows an example of a preemptive schedule where each process consists of a single segment. The second special case is when there exists a mutual exclusion relation between every pair of process segments that execute on a same processor, that is, no process segment is allowed to be preempted at any time. When this assumption is made the schedule constructed by an algorithm that solves the problem is called a *nonpreemptive* schedule. Fig. 4 shows an example of a nonpreemptive schedule where each process consists of a single segment.

Another type of constraints studied in some mathematical scheduling problems in the literature is called *resource constraints.* With this type of constraints, one assumes that there exists a set of m *resources*: $R = \{r_1, r_2, \ldots, r_m\}$ in the system, and one can specify which resources are required by each process during its execution. Note that if all processes

[1]Parallel computations can be represented by several processes, with various types of relations defined between individual segments belonging to different processes, and processes can be executed concurrently; thus requiring each process to be a sequence of segments does not pose any significant restrictions on the amount of parallelism that can be expressed.

are preemptable then this type of constraints alone cannot be used to implement a critical section, since while a process is using a resource it can be preempted by another process which uses the same resource. With resource constraints, it is possible to enforce "allocation" constraints, by modelling processors as resources, and specifying in advance which process should be allocated to which processor. With this type of constraints, processor allocation is prespecified and fixed, unlike multiprocessor allocation strategies where a pool of processors are interchangeable.

There are two distinct approaches to scheduling processes in hard-real-time systems. One is *run-time scheduling*, the other is *pre-run-time scheduling.*

In run-time scheduling, the schedule for processes is computed progressively on-line as processes arrive, and the scheduler does not assume any knowledge about the major characteristics of processes that have not yet arrived in the system. In the literature, computing the schedule at run-time is often called "on-line scheduling" or "dynamic scheduling". Perceived advantages of this approach include: it is unnecessary to know the major characteristics of the processes in advance, it is flexible and can easily adapt to changes in the environment. Often, the only stated disadvantage is its high run-time cost.

In pre-run-time scheduling, the schedule for processes is computed off-line; this approach requires that the major characteristics of the processes in the system be known in advance. It is possible to use pre-run-time scheduling to schedule periodic processes. This consists of computing off-line a schedule for the entire set of periodic processes occurring within a time period that is equal to the least common multiple of the periods of the given set of processes, then at run-time executing the periodic processes in accordance with the previously computed schedule. See Fig. 6 and Fig. 7 for an example.

In pre-run-time scheduling, often several alternative schedules may be computed off-line for a given time period; each such schedule corresponding to a different "mode" of operation. A small run-time scheduler can be used to select among the alternative schedules in response to external or internal events. This small run-time scheduler can also be used to allocate resources for a small number of asynchronous processes that have very short deadlines.

Some "mixed" strategies mix pre-run-time scheduling of hard-dead-line processes with run-time scheduling of *soft*-dead-line processes; Such strategies bring some of the benefits of pre-run-time scheduling, in situations where a "pure" strategy cannot be used.

It is possible to translate an asynchronous process into an equivalent periodic process. One technique for achieving this [Mok83], [Mok84], is to translate each asynchronous process (c_a, d_a, min_a) into a corresponding periodic process (r_p, c_p, d_p, prd_p) which satisfies the following conditions: $c_p = c_a$, $d_a \geq d_p \geq c_a$, $prd_p \leq min(d_a - d_p + 1, min_a)$, $r_p = 0$. For an example see Fig. 5. Thus it is also possible to schedule asynchronous processes using pre-run-time scheduling.

In the literature, pre-run-time scheduling is often called "off-line scheduling" or "static scheduling". Advantages of this approach include: it reduces significantly the amount of run-time resources required for scheduling and context switching. Pre-run-time scheduling has been perceived as being inflexible and not being capable of adapting to a changing environment or to an environment whose behavior is not completely predictable.

However, these often quoted advantages and disadvantages are really of secondary importance if the primary objective is to satisfy timing constraints. *For satisfying timing constraints in hard-real-time systems, predictability of the system's behaviour is the most important concern; pre-run-time scheduling is often the only practical means of providing predictability in a complex system.*

Even though the *exact* timing characteristics of processes and events in the computer system and environment may not be predictable, it is possible to use worse case estimates of the timing characteristics with a pre-run-time scheduling strategy to guarantee that timing constraints will be satisfied, and thus guarantee predictable behavior.

In the following sections we will explain further why pre-run-time scheduling can be essential for guaranteeing that timing constraints will be satisfied in a large complex hard-real-time systems, and why computing the schedule at run-time cannot, in general, provide such a guarantee.

3 Current standard practices in hard-real-time systems

There are two fundamental problems with the run-time scheduling approach to satisfying timing constraints in hard-real-time systems.

The first problem stems from the basic assumption that the scheduler does not have any knowledge about the major characteristics of processes that have not yet arrived in the system. *If the scheduler does not have such knowledge then it is impossible to guarantee that all timing constraints will be satisfied, because no matter how clever the scheduling algorithm is, there is always the possibility that a newly arrived process possesses characteristics that will make that process either miss its own deadline, or cause other processes to miss their deadlines. This is true even if the processor capacity was sufficient for the task at hand.*[2]

The second problem is the fact that the amount of time available for a run-time scheduler to compute schedules on-line is severely restricted, and the time complexity of most scheduling problems is very high. There are too many different possible combinations of times and orders in which processes may arrive and request use of system resources for the scheduler to compute at run-time, especially in complex systems where there is a large number of processes and resources. Being constrained to compute the schedule in a very limited amount of time at run-time reduces significantly the chances of finding a feasible schedule that takes into account all possible timing and resource constraints and meets every deadline for every possible combination, even if such a feasible schedule exists and can be computed easily off-line.

In the following, we shall take a look at current standard practices in hard-real-time systems design and discuss their shortcomings. In particular, we discuss why they fail

[2]For an example, see the adversary argument by Mok [Mok84].

to guarantee that timing constraints will be satisfied.

Current Standard Practice 1: assign static priorities to processes, then at each instant during run-time allocate system resources to the process that has the highest priority among all processes that have arrived and are requesting use of system resources.

Static priority-driven schemes are only capable of producing a very limited subset of the possible schedules for a given set of processes. This severely restricts the capability of priority-driven schemes to satisfy timing and resource sharing constraints at run-time. For example, there are situations where, in order to satisfy all given timing constraints, it is necessary to let the processor be idle for a certain interval in time, even though there are processes that have arrived and are requesting use of system resources. An example is shown in Fig. 9 where it is assumed that process segment B is not allowed to preeempt process segment A. The timing constraints then require that the processor must be left idle between time 0 and 11, even though A's release time is 0; if A starts it would cause B to miss its deadline. Note that in this example, some segments are preemptable, but others are not. In most real situations, there will be process segments that cannot be preempted because a critical section has been entered. Thus this example illustrates a very common situation. Neither static nor "dynamic" priority-driven schemes can deal properly with such situations.

In general, the smaller the subset of schedules that can be produced by a scheduling algorithm, the lower the level of processor utilization that can be achieved by that algorithm. For example, if one insists on using a priority-driven scheme to schedule the set of processes given in Fig. 9, then one would have to increase the processor capacity, and would achieve a much lower processor utilization than with a scheduling scheme capable of producing the schedule shown in Fig. 9. By using optimal algorithms that compute the schedule off-line, it is possible to achieve higher levels of resource utilization than those achievable by priority-driven schemes.[3] Hence, using priority-driven schemes may increase the cost of a system to non-competitive levels.

Because at run-time in a large complex hard-real-time system, there can be many different orders in which processes may arrive and request use of system resources, and there can be many relations between processes and resource sharing constraints that must be simultaneously satisfied, it is not generally possible to map the many different execution orderings of processes that are required by the different circumstances in a large complex system onto a rigid hierarchy of priorities.

For this reason, in practice the actual execution orderings of processes do not adhere to the original priorities of processes, but are changed on an ad-hoc basis to suit each particular circumstance (e.g., once a process enters a critical section, it may block any other process that also attempts to enter the critical section, regardless of the priorities assigned to them). This makes it more difficult to verify that all timing and resource constraints will be satisfied.

A few synchronization protocols that use priority-driven schemes for scheduling periodic processes provide some kind of "schedulability test", i.e., a set of sufficent conditions under which a set of processes are guaranteed to meet their deadlines [LiLa73] [ShRL90]. However, because of the difficulties mentioned above, most of the schedulability tests are restricted to special cases (e.g., independent processes; no synchronization constraints; no precedence relations between processes, etc); and, they tend to be complex. Furthermore, because priority-driven schemes can only produce a very limited subset of all possible schedules, in a large complex system, the level of processor utilization required to guarantee schedulability when complex relations between processes and resource sharing constraints are taken into account, may be too low to be really useful in practice.[3]

Often, priorities are assigned to processes based solely on how "important" each process is subjectively perceived to be, without a thorough analysis of how the assignment will affect the timing characteristics of other processes and of the whole system. When priorities are used in this way, they make it even more difficult to satisfy timing constraints of the whole system.

Current Standard Practice 2: rely on complex run-time mechanisms for process synchronization and mutual exclusion.

Such complex run-time synchronization mechanisms may include a variety of system calls or high level programming language constructs such as rendez-vous, monitors, etc.

One of the most serious problems with using such synchronization mechanisms, is that the timing of such synchronization mechanisms is often extremely difficult to predict with any certainty. For example, it is often impossible to put a reasonable upper bound (overly pessimistic upper bounds are of no practical use) on the amount of time that a process may stay in a software queue associated with a run-time synchronization mechanism, because that could depend on how many processes are currently in the queue, the runtime queueing policy (e.g., should it be FIFO which totally ignores timing constraints, or, should the position of processes at any time in the queue be determined by the timing constraints, which would result in an additional level of complexity?), etc.

In addition, the use of such run-time mechanisms by individual processes makes it possible for individual processes to make important scheduling decisions at run-time, even though individual processes normally do not possess global information about the system. This makes it practically impossible for the scheduler to schedule all processes in the system at the system level to satisfy timing constraints on a global basis.

Other difficulties include the amount of overhead involved, and the possibility of deadlocks or "starvation"

[3]Even under the assumption that all processes are completely independent of each other, and no critical sections exist, i.e., all processes can be preempted at any time, there still exist sets of processes with a processor utilization factor of approximately 70% for which no static priority assignment can meet the timing constraints, whereas 100% processor utilization is always achievable for such processes by alternative methods (e.g., the earliest-deadline-first scheduling algorithm) [LiLa73] [Mok84]. The processor utilization would be even lower when run-time synchronization on critical sections is required [ShRL90].

caused by the use of such run-time mechanisms. In general, deadlock avoidance at run-time requires that the run-time synchronization mechanism be conservative, resulting in situations where a process is blocked by the run-time synchronization mechanism, even though it could have proceeded without causing deadlock. This reduces further the level of processor utilization.

In contrast, in a pre-run-time scheduling approach, it is possible to avoid using sophisticated run-time synchronization mechanisms by directly defining precedence relations and exclusion relations on pairs of process segments to achieve process synchronization and prevent simultaneous access to shared resources. Compared with the complex schedulability analysis required when run-time synchronization mechanisms are used, it is straightforward to verify that all processes will meet their deadlines in an off-line computed schedule.

Current Standard Practice 3: allow events to interrupt processes and occupy system resources at any random time

a) This further increases the unpredictability of the timing behaviour of processes in the system and makes it even more difficult to devise a scheduling strategy that can take into account all possible timing and resource constraints and meet every deadline under all possible circumstances.

b) This also requires a large amount of context switching.

In contrast, pre-run-time scheduling reduces substantially the amount of run-time resources required for scheduling and for context switching [FaPa88]. Asynchronous process requests are buffered until the corresponding periodic processes are activated to serve them. So most interrupts can be eliminated, which not only reduces context switching, but also allows greater flexibility in meeting timing constraints of the whole system. Any remaining context switching can be optimized, since it is known in advance from the predetermined schedule which process preempts which other process and exactly where the preeemption occurs. Resource utilization can also be optimized, and it should be much easier to make future extensions to the system [ShGa90], since the precise resource utilization pattern is known in advance. For example, one can "fill up" the idle gaps in the predetermined schedule with processes that do not have strict timing constraints, or use the gaps for adding new processes or increasing the computation time of existing processes.

Current Standard Practice 4: rely on stochastic simulations to "verify" system design.

Since the above practices make it impossible to predict the timing characteristics of processes in the system, testing becomes the only means by which the designer can get information about how the system will behave at run-time. Dijkstra's observation that "testing shows the presence not the absence of bugs" [Dijk72] points out the fundamental flaw in this practice. The problem of satisfying timing constraints in a hard-real-time system is inherently a deterministic problem. Stochastic studies can only show the average timing behavior of the system, they cannot provide any guarantee regarding the worst case timing behavior of the system which is really what we are concerned about.

4 Major concerns in pre-run-time scheduling

In this section we present major concerns in pre-run-time scheduling and discuss formulations of mathematical scheduling problems that can be used to address those concerns. In many cases, the mathematical scheduling problems that researchers formulate and try to solve with various algorithms are only approximations of the real pre-run-time scheduling problems that need to be solved in reality. Below we refer to problems encountered in practice as "real problems" to distinguish them from the mathematical problems under discussion. We are mostly interested in mathematical scheduling problems/algorithms in the literature that can be used to provide a "valid" and "reasonable" approximation to the real pre-run-time scheduling problem we have on hand. By *valid* we mean that an algorithm that solves the mathematical scheduling problem must guarantee that all timing constraints in the real pre-run-time scheduling problem will be satisfied. By *reasonable* we mean that the approximation should not be so crude that the chances of finding a feasible schedule where all timing constraints are met are significantly reduced. An example of an approximation that is valid but not reasonable is a mathematical scheduling problem which assumes that all process computation times are equal to the greatest process computation time when the actual computation times of processes differ significantly.

When we discuss scheduling algorithms, we shall distinguish between *optimal algorithms* and *heuristics*. An optimal scheduling algorithm is capable of always finding a feasible schedule for a given mathematical scheduling problem whenever one exists. In contrast, for a given mathematical scheduling problem, a heuristic attempts to find a feasible schedule, but it is possible that it may not find one even if one exists[4].

We would like to emphasize again that we are assuming that in the real pre-run-time scheduling problem there exists a relatively large number of processes that have stringent timing constraints, that the processes have different timing characteristics and interdependencies, and that the processor utilization factor is not very low, i.e., there is not much spare CPU capacity available. In other words, we assume that one cannot use ad hoc manual methods to satisfy the timing constraints easily. If the system is fairly simple, has few processes or most processes have uniform characteristics, or there is a very large amount of spare CPU capacity available, then it may be feasible to solve the problem manually with methods such as the *cyclic executive* method [BaSh89], [BaSh89], [MeSc82], [Carl84], or, it may be feasible to compute the schedule at run-time and rely on worse-case run-time delay bounds or "schedulability tests" to guarantee that timing constraints will be satisfied [Lein80] [LiLa73] [ShRL90].

In the following, we discuss some of the major concerns

[4]Some optimal algorithms use techniques called "heuristic search" that guide an exhaustive search, trying the paths that are most likely to lead to a feasible solution first, but eventually covering all paths if necessary. In comparison, heuristic methods may consider only a subset of the possible solutions and may miss some solutions.

in pre-run-time scheduling.

Concern 1: Satisfy relevant timing constraints.

As was mentioned in the previous section, it is possible to satisfy the timing constraints of a set of periodic processes and asynchronous processes if we translate asynchronous processes into periodic processes and a feasible pre-run-time schedule can be found in which every process starts after its release time and completes its computation before its deadline within a time period that is equal to the least common multiple of their periods.

Thus the minimum set of requirements for a mathematical scheduling problem to be able to provide a valid approximation to a real pre-run-time scheduling problem are the following:

a) The mathematical scheduling problem should consider arbitrary release times and arbitrary deadlines.

In the literature some mathematical scheduling problems only consider special cases where either the release times or the deadlines are the same for all processes. (See Table 1.) In real pre-run-time scheduling problems such special cases are rare.

b) Processes in general have arbitrary computation times in real pre-run-time scheduling problems, so it is preferable that mathematical scheduling problems consider arbitrary computation times.

In the literature some mathematical scheduling problems only consider special cases where all processes are nonpreemptable and have identical (or unit) computation times [GJST81], [GaJo76], [GaJo77], [Simo83], [Blaz79]. (See Table 1.) If in the real pre-run-time scheduling problem on hand all processes have similar computation times, then it may be possible to use such mathematical problems and their associated algorithms to obtain a valid and reasonable approximation by considering the mathematical problem where all processes have a length that is equal to the maximal length among all processes in the real problem.

If the mathematical problems and their associated algorithms also consider precedence constraints besides considering nonpreemptable processes of the same length, then they can be used to obtain a valid approximation by breaking up each original process into a set of new processes with equal length, and adding precedence constraints to ensure that the ordering of the new processes is consistent with the original processes. One of the disadvantages of using such an approximation approach is that it would create a very large number of new processes. This increases the time it takes to find a feasible schedule. Later, we will also discuss other difficulties that may prevent one from using this approach.

c) The objective of the mathematical scheduling problem should be to obtain a feasible schedule, i.e., a schedule in which every process starts after its release time and completes its computation before its deadline.

Certain mathematical scheduling problems and their associated algorithms in the literature achieve this objective by minimizing the lateness of the schedule, i.e., the maximal difference between the completion time and deadline for all processes in the schedule. Using this method, they can find a feasible schedule whenever one exists.

Some scheduling algorithms in the literature, e.g. [BlDW86], have the main objective of minimizing the schedule length, that is, the maximum finishing time among all processes. See [Coff76], [Gonz77] and [LaLR81] for surveys of algorithms that use this performance measure. This performance measure does not consider individual process deadlines, thus such algorithms are not really applicable to pre-run-time scheduling.

Another type of mathematical problems assume that the total number of processors can vary instead of being fixed and try to minimize the number of processors under the condition that a feasible schedule exists for that number of processors [DhLi78], [DaDh86], [GoSo76]. However, in most real pre-run-time scheduling problems the number of processors is fixed.

Concern 2: controlling the execution order of processes.

When we design a hard-real-time system, we normally break up large tasks into sets of smaller activities, or processes. Often, there is a particular order in which some of those processes must be executed relative to each other. Such execution orders can be specified with *precedence relations* that are defined on ordered pairs of processes,[5] and require algorithms that consider precedence relations to enforce them in a pre-run-time schedule.

Concern 3: prevent simultaneous access to shared resources such as data, I/O devices, etc.

On a single processor, this could be achieved by prohibiting preemptions by other processes who may access the same shared resource. The section of code where a process should prohibit preemptions while accessing a shared resource is often called a *critical section.*

Many mathematical scheduling problems in the literature make the assumption that all processes can be preempted at any time by any other process. (See Table 1.) The main disadvantage of this assumption is that it makes it impossible to prevent simultaneous access to shared resources.

In order to prevent simultaneous access to shared resources on a single processor it is preferable if one can specify exactly which portion of which process cannot be preempted by which portion of which other process, while allowing other portions of the same process to be preemptable by other processes. This is possible if the mathematical scheduling problem and associated algorithm explicitly considers exclusion constraints. Fig. 9 shows a feasible schedule constructed by the algorithm in [XuPa90] that satisfies a given set of exclusion and precedence relations defined on ordered pairs of process segments in addition to release time and deadline constraints.

[5]Using priorities alone is inadequate for controlling the execution order of processes. Suppose that it is required that process A precede process B. If B happens to arrive and request execution before A, and B is the only process to request execution at that time, then B will be executed before A, even if A had been assigned a priority that is higher than the priority of B. But the precedence relation A *precedes* B, if properly enforced by the scheduling algorithm, guarantees that A will always be executed before B, regardless of which one arrived first.

Mathematical scheduling problems and associated algorithms that make the assumption that all processes are nonpreemptable can be used to provide a valid approximation to the problem of preventing simultaneous access to shared resources on a single processor, although such an assumption would significantly reduce the chances of finding a feasible schedule.

With mathematical scheduling problems and their associated algorithms that make the assumption that all processes are nonpreemptable and have the same computation time, and the problem includes precedence constraints, a valid approximation that prevents simultaneous access to shared resources can be made by making the computation time of all the new processes equal to or greater than the length of the longest critical section among all the original processes, although this will also reduce the chances of finding a feasible schedule unless all processes have very short critical sections. Such a strategy has been used by Mok for a scheduling model called the Kernelized Monitor Model [Mok83] [Mok84].

We note that conventional resource constraints (i.e., process i requires resource $r_m, \ldots, r_n$) that are assumed in some mathematical scheduling problems and their associated algorithms [ZhRS87], [Blaz79] alone are not sufficient to prevent simultaneous access to shared resources unless one can also specify that preemption by other processes is prohibited during critical sections. This observation applies to both single and multiprocessor cases.

On multiprocessors, prohibiting preemption alone is not sufficient to prevent simultaneous access to shared resources. It is preferable to prevent simultaneous access to shared resources on multiprocessors by using explicit exclusion constraints [Xu90].

Mathematical scheduling problems and their associated algorithms which make the assumption that all processes are nonpreemptable and which also consider resource constraints [ZhRS87a] can be used to obtain a valid approximation which prevents simultaneous access to shared resources. However, each process would then have to occupy resources for the entire duration of the process, which decreases the chances of finding a feasible schedule. In such cases, it would be useful if the mathematical scheduling problem considers precedence constraints in addition to resource constraints when all processes are nonpreemptable, because then one may split the processes into smaller process segments and use precedence constraints to ensure the proper ordering of the segments which belong to the same process.

Concern 4: take into account and reduce the cost of context switching overhead caused by preemptions.

In mathematical scheduling problems in the literature, reducing the number of preemptions is seldom explicitly specified as an objective, even though it is of significant practical importance. In order to address this concern, one may select algorithms which do not result in a large number of preemptions and which also provide a predictable and reasonable upperbound on the number of preemptions. Algorithms that are known to have this property include algorithms that use variations of the earliest deadline first strategy, [LiLa73] [Serl72] [Horn74] [MoDe78] [McFl75] [Guns84], [Xu90] [XuPa90].

In comparison, algorithms which use a pure least laxity strategy [MoDe78], or use techniques that solve network flow problems [BrFR71], [Mart82], or various other strategies [GoSa78], [SaCh79], [SaCh80], may result in a significantly larger number of preemptions.

Concern 5: increase the chances that a feasible schedule can be found.

One should maximize the chances that a feasible schedule can be found under the condition that each of the concerns above is adequately addressed. In general the following can be observed:

1) if preemptions are allowed whenever possible (e.g., when explicit exclusion relations are used instead of assuming that all processes are nonpreemptable), the scheduling algorithm should have a greater chance of finding a feasible schedule; For example, in Fig. 9, a feasible schedule exists because the process segments C and E were allowed to preempt process segment A, whereas no feasible schedule would have existed if all process segments were nonpreemptable.

2) if the input data of the mathematical scheduling problem to be solved corresponds exactly to the input data of the real pre-run-time scheduling problem (e.g., arbitrary computation times) instead of an approximation (e.g., assuming all computation times are the same), then the scheduling algorithm that solves the mathematical scheduling problem should have a greater chance of finding a feasible schedule.

3) for a given problem, if an optimal algorithm is used instead of a heuristic, the scheduling algorithm should have a greater chance of finding a feasible schedule, because an optimal algorithm is guaranteed in theory to find a feasible schedule whenever one exists, while a heuristic may fail to find a feasible schedule even if one exists.

5 Characteristics of mathematical scheduling problems and algorithms in the literature

In the following, we examine the characteristics of scheduling problems and algorithms that have been discussed in the literature (See Table 1) and explain how they relate to the concerns discussed above.

1) **Process type.**

a) *Static processes*

The majority of scheduling algorithms in the literature are designed to schedule a static set of processes for which the release times, deadlines, and computation times for each process are given. Such algorithms can be used to perform pre-run-time scheduling for both periodic and asynchronous processes as explained below.

b) *Periodic processes.*

Some algorithms explicitly consider a set of periodic processes, e.g., [DaDh86], [GoSo76], [DhLi78], [LiLa73], etc.

However, for pre-run-time scheduling of periodic processes, it is possible to use algorithms that were originally designed for scheduling a static set of processes. This is because, as mentioned earlier, if we can find a feasible schedule for the instances of a given set of periodic processes within a time interval between 0 and the least common multiple

(LCM) of the process periods, then the given set of periodic processes can be scheduled by repeating that feasible schedule [LeMe80], [LaMa81].

If the periods are relatively prime, then one may have to compute a fairly long schedule. But in practice, there is often flexibility in adjusting periods so that the least common multiple of all periods is of reasonable length.

c) *Asynchronous processes.*

Scheduling algorithms in the literature that are designed to be used "on-line" assume that the release times of the asynchronous processes are not known in advance, and attempt to schedule them as they arrive at run-time. For the reasons described earlier, unless the system is simple enough to allow worse case delay bounds or "schedulability tests" to be used, a run-time scheduling approach will not provide any guarantee that all deadlines will be met.

In order to provide such a guarantee, one may first translate asynchronous processes into equivalent periodic processes [Mok84] (see Fig. 5). Then one can use algorithms for static processes to compute a feasible schedule for the periodic processes before run-time.

d) *Both periodic and asynchronous processes.*

The same method above can be used to guarantee in advance that both periodic and asynchronous processes will meet their deadlines. This can be achieved by first translating asynchronous processes into equivalent new periodic processes; then use algorithms for static processes to find a feasible schedule for both the original and new periodic processes within a time interval between 0 and the LCM of the original and new process periods.

2) Release times.

a) *Release times not known in advance*

As mentioned above, algorithms that are designed to be used "on-line", that is, perform scheduling at run-time, assume that release times are not known in advance. For simple problems where all processes are preemptable and only a single processor is used, there exist algorithms designed for on-line scheduling that can make optimal scheduling decisions for processes in an incremental fashion, i.e., they do do not need to know any information about processes that have release times greater than the release times of the processes currently being scheduled [Horn74], [MoDe78], [Blaz76].

Algorithms designed for on-line scheduling can also be used for pre-run-time scheduling where the release times are known in advance. However, for all but the simplest pre-run-time scheduling problems, if an algorithm is constrained to make scheduling decisions for processes in an incremental fashion, the algorithm's ability to find a feasible schedule would be significantly reduced.

Some algorithms designed primarily for on-line scheduling also have extensions for use in an off-line mode by allowing limited backtracking, in which previous scheduling decisions can be reversed, thus increasing the possibility that a feasible schedule will be found [ZhRS87], [ZhRS87a].

b) *Same release time for all processes.*

Some algorithms assume that the release time is the same for all processes, e.g., [GoSa78], [SaCh80], [Blaz76]. However, such cases are extremely rare in hard-real-time applications.

c) *Start of period.*

Most algorithms in the literature that explicitly assume that each process is periodic also assume that the release times of processes are the same as the start times of their periods.

d) *Arbitrary release times.*

The method mentioned earlier for translating asynchronous processes into periodic processes will result in a set of new periodic processes with identical periods, but with different release times that are fixed points within the identical period. Thus it is often useful to consider arbitrary release times.

3) Deadlines

a) *Deadlines not known in advance*

If the deadlines are not known in advance, and the algorithm is constrained to make scheduling decisions for processes in an incremental fashion, then the algorithm's ability to find a feasible schedule would be significantly reduced.

b) *Same deadline for all processes.*

Some algorithms assume that the deadline is the same for all processes, e.g., [SaCh79]. However, again such cases are extremely rare in hard-real-time applications.

c) *End of period.*

Most algorithms in the literature that explicitly assume that each process is periodic also assume that the deadlines of processes are the same as the end times of their periods.

d) *Arbitrary deadlines.*

The method mentioned earlier for translating asynchronous processes into periodic processes will result in a set of new periodic processes with identical periods, but with different deadlines that are fixed points within the identical period. Thus it is also useful to consider arbitrary deadlines.

4) Process synchronization.

a) *None: all processes are preemptable by any other process at any point during their executions.*

Many scheduling algorithms in the literature make this assumption. Unfortunately, this assumption makes it difficult to prevent processes from simultaneously accessing shared resources.

b) *Resource constraints.*

Most algorithms that deal with resource constraints specify that a process should use one or more resources exclusively at any time that process is being executed, e.g. [ZhRS87], [ZhRS87a]. This kind of resource constraint does not prevent simultaneous access to shared resources if any process can preempt any other process at any time, because when a process is preempted by another process, it relinquishes all resources that it was using, and continues to use those resources when it resumes execution. But in the mean time, the preempting process may access some of those resources, causing inconsistencies.

Resource constraints also require that each process must hold every resource that it needs for the entire duration of its computation rather than for particular portions of its computation.

c) *No process can be preempted by any other process at any point during its execution.*

Most scheduling algorithms in the literature that do not assume that all processes are preemptable assume this condition, e.g., [BrFr71], [BaSu74], [ZhRS87a], etc.

If only one processor is used, then prohibiting all preemptions is sufficient to prevent simultaneous access to shared resources.

If more than one processor is used, then prohibiting all preemptions alone is not sufficient to prevent simultaneous access to shared resources. In order to prevent simultaneous access to shared resources, one can either combine this condition with resource constraints [ZhRS87a], or, use general exclusion relations as discussed below.

A significant disadvantage of prohibiting *all* preemptions is that it reduces the chances of finding a feasible schedule, i.e., the system's ability to meet deadlines.

d) *General exclusion relations.*

One may define exclusion relations on ordered pairs of process segments that access shared resources, which we call *critical sections.* If process segment X excludes process segment Y, if no execution of Y can occur between the time that X starts its computation and the time that X completes its computation.

With general exclusion relations, one may specify which process segments are preemptable by which other process segments, and which process segments are not preemptable by which other process segments.

This is most preferable, if we want to prevent simultaneous access to shared resources, while at the same time maximizing the chances of finding a feasible schedule.

Many types of process synchronization do not require the symmetry of mutual exclusion: when a process segment X excludes process segment Y, process segment Y may not always have to exclude process segment X. Various versions of the readers and writers problem [CoHP71] [BeWi73] are examples. In such cases, it would be better to use nonsymmetric exclusion relations [FaPa88] [XuPa90]. Note that one can always define a mutual exclusion relation using a pair of nonsymmetric exclusion relations [BeWi73].

In practice, it is often the case that various segments within processes that access shared resources are not disjoint, i.e., they overlap, or nest within each other. If one is only allowed to use exclusion relations that define non-overlapping and non-nested critical sections, it would result in larger than necessary critical sections, and restrict the scheduler's ability to find a feasible schedule. In such cases, it would be better to use exclusion relations that can be defined on overlapping or nested critical sections [XuPa91].

5) Precedence relations.

a) *No precedence relations.*

If no precedence relations can be specified, then the only possible way of controlling the execution order of interdependent processes, is to lump them together into a larger single process in which their execution order is determined by the static code. But this approach can be unsatisfactory for many reasons, including the fact that it is in general much easier to design the system as a set of smaller cooperating processes or modules; and the fact that it gives less flexibility to the pre-run-time scheduling algorithm for satisfying timing constraints, etc.

A much better way of controlling the execution order of interdependent processes is to specify precedence relations on ordered pairs of processes (or process segments), and use algorithms that solve mathematical scheduling problems with precedence relations to schedule them.

b) *Tree precedence relations.*

Some algorithms restrict the precedence relations that are enforced to be in the form of in-trees/out-trees. That is, each process can have at most one other process that is constrained to directly follow/precede it. This puts rather awkward constraints on how the processes are structured.

c) *General precedence relations.*

The most general case of precedence relations, is the case where precedence relations can be defined by an arbitrary acyclic directed graph, where each node in the graph represents a process, and a directed edge between two nodes represents a precedence relation between the two corresponding processes.

6) Computation times.

a) *Uniform computation times.*

In the literature some scheduling algorithms only consider special cases where all processes are nonpreemptable and have uniform (or unit) computation times. As discussed earlier, if the algorithm also allows precedence relations to be specified, then one can apply such an algorithm by breaking up the original set of processes into a new set of smaller processes of equal length and using the precedence relations to guarantee that they are scheduled in an order that is consistent with the original processes.

b) *Arbitrary integer computation times.*

In most cases, computation time can be specified using integer time units that each correspond to one CPU clock cycle. Thus, algorithms that assume that processes have arbitrary integer computer times should be satisfactory for most applications.

c) *Arbitrary real computation times.*

Some algorithms assume that processes have arbitrary real computation times, e.g., [GJST81], [Mart82], [Simo83].

7) Number of processors.

a) *1 processor.*

The majority of scheduling algorithms in the literature assume that there is only one processor. In general, single processor algorithms cannot be directly applied to cases where there are 2 or more processors.

b) *2 processors.*

Some algorithms deal with the special case where 2 processors are used, e.g., [GaJo76], [GaJo77]. Such algorithms normally do not allow more than 2 processors to be used.

c) *n pre-assigned processors.*
In some algorithms, CPU's are modelled as resources [ZhRS87], [ZhRS87a]. For every resource R_j and every process p_i, whether p_i must use R_j is pre-specified and fixed, unlike multiprocessor allocation strategies where a pool of processors are interchangeable.

d) *n processors.*
Normally, multiprocessor algorithms can also be applied to the special case where there is only one processor. However, single processor algorithms are normally more efficient than multiprocessor algorithms when solving single processor cases.

8) **Processor speeds.**

a) *Identical processor speeds.*
Most multiprocessor algorithms assume that all processors are identical, that is, they run processes at the same speed.

b) *Arbitrary processor speeds.*
In some multiprocessor applications, processors with different speeds are used and it would be desirable if corresponding algorithms are designed. Some algorithms consider preemptive scheduling of independent processes on processors with different speeds, e.g., [Mart82], [SaCh79], [SaCh80].

9) **Measures of performance.**

a) *Minimizing schedule length.*
The main objective of some algorithms in the literature is to minimize the schedule length, that is, the maximum finishing time. See [Coff76], [Gonz77] and [LaLR81] for a survey of such algorithms. This performance measure does not consider individual process deadlines. Since this performance measure is not really applicable to pre-run-time scheduling, we did not include such algorithms in Table 1.

b) *Minimizing the number of processors.*
Some algorithms assume that the total number of processors can vary instead of being fixed and try to minimize the number of processors under the condition that a feasible preemptive schedule exists for that number of processors [DhLi78], [DaDh86], [GoSo76]. All of the algorithms of this type referred to in Table 1 make the assumption that all processes are completely preemptable, which makes it difficult to prevent simultaneous access to shared resources.

c) *Finding a feasible schedule.*
This is the most useful performance objective in pre-run-time scheduling.
Certain scheduling algorithms in the literature, e.g. [McFl75], achieve this objective by minimizing the lateness of the schedule, i.e., the maximal difference between the completion time and deadline for all processes in the schedule. Using this method, they can find a feasible schedule whenever one exists.

10) **Algorithm optimality**

a) *Optimal algorithm.*
In the case where the performance measure is lateness, an *optimal algorithm* is an algorithm that finds a feasible schedule such that all processes meet their deadlines whenever such a schedule exists for any given problem.
Similarly, in the case where the objective is to minimize the number of processors, or schedule length, an optimal algorithm is an algorithm that always finds the minimum number of processors or minimum schedule length for any given problem.

b) *Heuristic.*
In the case where the performance measure is lateness, a *heuristic* may not find a feasible schedule such that all processes meet their deadlines even if such a schedule exists for the given problem.
Similarly, in the case where the objective is to minimize the number of processors, or schedule length, a heuristic may not always be able to find the minimum number of processors or the minimum schedule length for a given problem.

6 Concluding Comments

In this paper, we explained why pre-run-time scheduling is essential if the objective is to guarantee in advance that stringent timing constraints will be satisfied in a large complex hard-real-time system.

We presented some of the major concerns in pre-run-time scheduling and discussed what formulations of mathematical scheduling problems can be used to address those concerns.

We also examined the characteristics of mathematical scheduling problems and algorithms discussed in the literature and explained how they relate to those concerns.

Appropriate algorithms for solving mathematical scheduling problems that address those concerns can be used to automate pre-run-time scheduling.

In the past, practitioners working on safety-critical hard-real-time systems, such as submarine data processing and aircraft weapon delivery programs, have been observed doing such problems by hand. The result of this "hand scheduling" is "spaghetti code" that is very hard to verify and maintain. Except for very simple problems, ad hoc and manual methods are prone to errors, time consuming, and they often fail to find a feasible schedule even when one exists.

Appropriate scheduling algorithms can greatly facilitate the task of pre-run-time scheduling. Their application would virtually eliminate any possibility of errors in the computation of schedules. Not only would they be capable of finding a feasible schedule whenever one exists, they would also be capable of informing the user whenever no feasible schedule exists for a given set of parameters much faster and reliably than any ad hoc or manual method.

If the task of computing schedules is completely automated, it would be very easy to modify the system and re-compute new schedules in case changes are required by applications. This can greatly reduce the cost of maintaining the system.

There is still much work that needs to be done in pre-run-time scheduling. A major problem is that most of the mathematical scheduling problems that have been studied so far rely on too many assumptions which do not hold in reality. As shown earlier, few mathematical scheduling problems studied in the past address all of the major concerns in pre-run-time scheduling, and even fewer have optimal algorithms that address all of those concerns.

There seems to be a tendency to believe that if a scheduling problem has a high computational complexity, i.e., if it has been proved to be NP-hard [GaJo79], then one should always use a heuristic rather than an optimal algorithm to solve the problem. However we think that this belief is mistaken for the following reasons.

1) Although it is possible to construct pathological problem instances of an NP-hard scheduling problem such that in order to find a feasible solution to those problem instances an optimal algorithm would require an amount of computation time that is exponentially related to the problem size, such pathological problem instances occurred seldom in our (limited) experience. More experience is needed.

2) Optimal algorithms can be designed in a way such that at each intermediate stage of the algorithm a complete schedule is constructed and the algorithm starts with an initial schedule that is obtained by a good heuristic, and then systematically improves on that initial schedule until one among the following occurs: a) a feasible schedule is found; or b) it is determined that no feasible schedule exists; or c) a pre-specified time or space limit has been exceeded. In this way, even if the algorithm has to be terminated prematurely, it would still deliver a complete schedule that is at least as good as any schedule obtained by using a heuristic [XuPa90], [Xu90].

3) When schedules are computed off-line, time efficiency is not of the greatest concern. Of greater concern is the ability to find a feasible schedule when one exists or determine that a feasible schedule does not exist. In the latter case, it is preferable if the algorithm can give clear indication as to which problem parameters should be changed in order to find a feasible schedule. (In [XuPa90], and [Xu90], this is achieved by allowing the system designer to restrict his attention to the latest process segment and the subset of process segments that precede and include the latest process segment in a continuous utilization of the processor in the schedule).

While we have listed some of the major concerns in pre-run-time scheduling, we have by no means exhausted all the important concerns. For example, an important concern not discussed so far is how to schedule processes on processors with different speeds. In [Mart82] the problem of scheduling processes on processors with different speeds has been considered under the assumption that all processes are preemptable and independent of each other, but this assumption and the lack of capability to address the other major concerns discussed earlier makes it difficult to apply the results in practice.

Another important concern is how does one incorporate fault detection, error recovery, and fault tolerance in pre-run-time scheduling. Intuitively, it should be much easier to address this concern with pre-run-time scheduling as compared with conventional run-time scheduling, although more work needs to be done to address this issue.

Also more work needs to be done on how to guarantee in advance that timing constraints will be satisfied in a distributed computer system where the progress of processes on each computer depend on timely communication with processes on other computers through a computer network [ZhSR90], [Stan88]. Another problem that has been investigated recently is the problem of scheduling real-time periodic jobs that allow imprecise results [ChLL90].

In general, much work needs to be done on solving scheduling problems that address simultaneously and effectively all the major concerns we discussed earlier. Such scheduling problems would have to deal with many different input data and requirements, they would seem "messy" and would most certainly be NP-hard and difficult to solve. They would represent a major departure from the simple and clean mathematical scheduling problems studied in the past for which simple polynomial time solutions exist, but which are seldom useful in practice. They would most likely inspire optimal algorithms that employ new implicit enumeration techniques, and new heuristics that are more systematic in their search for a feasible schedule than existing heuristics.

Finally, we note that the ability to determine a reasonably accurate upper-bound on the computation times of processes and process segments is crucial to satisfying timing constraints in hard-real-time systems. Although some work related to this problem has been done [KlSt86] [PuKo89] [Shaw89], much more work remains to be done in this area.

ACKNOWLEDGEMENTS

The authors are grateful to the referees for many helpful comments and suggestions that have led to significant improvements to this paper.

REFERENCES

[BaSh89] T. P. Baker, and A. Shaw, "The cyclic executive model and Ada," *Journal of Real-Time Systems*, vol. 1, June 1989.

[BaSu74] K. R. Baker, and Zaw-Sing Su, "Sequencing with due-dates and early start times to minimize maximum tardiness," *Nav. Res. Log. Quart.*, vol. 21, Mar. 1974.

[BeWi73] G. Belpaire, and J. P. Wilmotte, "A semantic approach to the theory of parallel processes," *Proceedings of the 1973 European ACM Symposium* (Davos, Switzerland), ACM, New York, 1973.

[Blaz76] J. Blazewicz, "Scheduling dependent tasks with different arrival times to meet deadlines," *Modelling and Performance Evaluation of Computer Systems*, E. Gelenbe ed. North-Holland, 1976.

[Blaz79] J. Blazewicz, "Deadline scheduling of tasks with ready times and resource constraints," *Inform. Proc. Lett.*, vol. 8, Feb. 1979.

[BlDW86] J. Blazewicz, M. Drabowski, and J. Weglarz, "Scheduling multiprocessor tasks to minimize schedule length," *IEEE Trans. on Computers*, C-35(5), 1986.

[BrFR71] P. Bratley, M. Florian, and P. Robillard, "Scheduling with earliest start and due date constraints," *Nav. Res. Log. Quart.*, vol. 18, Dec. 1971.

[BrFR75] P. Bratley, M. Florian, and P. Robillard, "Scheduling with earliest start and due date constraints on multiple machines," *Nav. Res. Log. Quart.*, vol. 22, 1975.

[Carl80] J. Carlier, "Probleme a une machine", Manu–script, Institute de Programmation, Universite Paris VI, 1980.

[Carl84] G. D. Carlow, "Architecture of the space shuttle primary avionics software system," *Commun. ACM,* Sept 1984.

[ChLL90] J.-Y. Chung, J. W. S. Liu, and K.-J. Lin, "Scheduling periodic jobs that allow imprecise results," *IEEE Trans. on Computers*, vol. 39, Sept 1990.

[Coff76] E. G. Coffman, Jr., "Computer and Jobshop Scheduling Theory", N.Y.: Wiley -Interscience, 1976.

[CoHP71] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with readers and writers," *Commun. ACM,* vol. 10, Oct. 1971.

[DaDh86] S. Davari, and S. K. Dhall, "An on-line algorithm for real-time tasks allocation," *IEEE Real-Time Systems Symposium*, Dec. 1986.

[DhLi78] S. K. Dhall, and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, 1978.

[Dijk68] E. W. Dijkstra, "Cooperating sequential processes," in *Programming Languages* (Genuys F., ed.), London: Academic Press, 1968.

[Dijk72] E. W. Dijkstra, "Notes on structured programming," in *Structured Programming* (O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare), London: Academic Press, 1972.

[FaPa88] S. R. Faulk and D. L. Parnas, "On synchronization in hard-real-time systems", *Commun. ACM.* vol. 31, Mar. 1988.

[GaJo76] M. R. Garey and D. S. Johnson, "Scheduling tasks with non-uniform deadlines on two-processors", *J.ACM*, vol. 23, July 1976.

[GaJo77] M. R. Garey and D. S. Johnson, "Two-processor scheduling with start- times and deadlines", *SIAM J. Comput.*, vol. 6, Sept. 1977.

[GaJo79] M. R. Garey and D. S. Johnson, "Computers and Intractability : A Guide to the Theory of NP-Completeness", Freeman, San Francisco, 1979.

[GJST81] M. R. Garey, D. S. Johnson, B. B. Simons, and R. E. Tarjan, "Scheduling unit-time tasks with arbitrary release times and deadlines", *SIAM J. Comput.*, vol. 10, May 1981.

[Gonz77] M. J. Gonzalez, Jr., "Deterministic processor scheduling", *Computing Surveys*, vol. 9, Sep. 1977.

[GoSa78] T. Gonzalez, and S. Sahni, "Preemptive scheduling of uniform processor systems," *J. ACM*, vol. 25, Jan. 1978.

[GoSo76] M. J. Gonzalez, and J. W. Soh, "Periodic job scheduling in a distributed processor system," *IEEE Trans. Aerospace and Electronic Systems*, AES-12, 5, Sep. 1976.

[Guns84] D. Gunsfield, "Bounds for naive multiple machine scheduling with release times and deadlines", *Journal of Algorithms*, vol. 5, 1984.

[Horn74] W. A. Horn, "Some simple scheduling problems," *Nav. Res. Log. Quart.*, vol. 21, Mar. 1974.

[KlSt86] E. Kligerman and A. Stoyenko, "Real-Time Euclid: a language for reliable real-time systems," *IEEE Trans. on Software Engineering,* SE-12, Sep. 1986.

[LaLR81] E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Recent developments in deterministic sequencing and scheduling: a survey", in *Proc. NATO Advanced Study and Research Institute on Theoretical Approaches to Scheduling Problems*, Durham, England, July 1981, in "Deterministic and Stochastic Scheduling. M. A. H. Dempster et al ed., D. Reidal Publishing Company.

[Lein80] D. W. Leinbaugh, "Guaranteed response time in a hard real-time environment," *IEEE Trans. Software Eng.*, vol SE-6, Jan. 1980.

[LaMa81] E. L. Lawler, and C. U. Martel, "Scheduling Periodically occurring tasks on multiple processors," *Information Processing Letters*, vol. 12, Feb. 1981.

[Lein80] D. W. Leinbaugh, "Guaranteed response time in a hard real-time environment," *IEEE Trans. Software Eng.*, vol SE-6, Jan. 1980.

[LeMe80] J. Y.-T. Leung, and M. L. Merrill, "A note on preemptive scheduling of periodic, real-time tasks," *Information Processing Letters*, vol. 11, Nov. 1980.

[LiLa73] C. L. Liu, and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment", *J.ACM*, vol. 20, Jan. 1973.

[Mart82] C. Martel, "Preemptive scheduling with release times, deadlines, and due dates", *J.ACM*, vol. 29, July 1982.

[McFl75] G. McMahon and M. Florian, "On scheduling with ready times and due dates to minimize maximum lateness", *Operations Research*, vol. 23, 1975.

[MeSc82] M. P. Melliar-Smith, and R. L. Schwartz, "Formal specification and mechanical verification of SIFT: A fault-tolerant flight control system," *IEEE Trans. Comput.*, vol C-31, July 1982.

[MoDe78] A. K. Mok and M. L. Detouzos, "Multiprocessor scheduling in a hard real-time environment", in *Proc. 7th IEEE Texas Conference on Computing Systems*, Nov. 1978. (Also in *IEEE Transactions on Software Engineering*, vol. 15, Dec. 1989).

[Mok83] 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.

[Mok84] A. K. Mok, "The design of real-time programming systems based on process models", in *Proc. IEEE Real-Time Systems Symposium*, Dec. 1984.

[PuKo89] P. Puschner and Ch. Koza, "Calculating the maximum execution time of real-time programs," *Real-Time Systems*, vol. 1, Sep. 1989.

[SaCh79] S. Sahni, and Y. Cho, "Nearly on-line scheduling of a uniform processor system with release times," *SIAM J. Comput.*, vol. 8, 1979.

[SaCh80] S. Sahni, and Y. Cho, "Scheduling independent tasks with due dates on a uniform processor system," *J. ACM*, vol. 27, July, 1980.

[Serl72] O. Serlin, "Scheduling of time critical processes," *Proc. Spring Joint Computer Conference, 1982.*

[Shaw89] A. C. Shaw, "Reasoning about time in higher-level language software," *IEEE Trans. on Software Eng.*, SE-15, July 1989.

[ShGa90] T. Shepard and M. Gagne, "A model of the F18 mission computer software for pre-run-time scheduling," *Proc. 10th Int. Conf. on Distributed Computing Systems*, 1990.

[ShRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Trans. on Computers*, vol. C-39, Sep. 1990.

[Simo83] B. Simons, "Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines", *SIAM J. Comput.*, vol. 12, May 1983.

[Stan88] J. Stankovic, "Real-time computing systems: the next generation," in IEEE Computer Society Tutorial "Hard Real-Time Systems," J. Stankovic Ed., 1988.

[XuPa90] J. Xu and D. L. Parnas, "Scheduling processes with release times, deadlines, precedence, and exclusion relations," *IEEE Trans. on Software Engineering*, Mar. 1990.

[Xu90] J. Xu, "Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations," Technical Report No. CS-90-13, Department of Computer Science, York University, Nov. 1990.

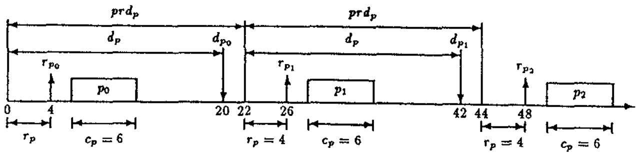
[XuPa91] J. Xu, and D. L. Parnas, "Pre-run-time scheduling of processes with exclusion relations on nested or overlapping critical sections," Technical Report No. CS-91-03, Department of Computer Science, York University, April 1991.

[ZhRS87] W. Zhao, K. Ramamrithan, and J. Stankovic, "Scheduling tasks with resource requirements in hard real-time systems," *IEEE Trans. on Software Engineering*, vol. SE-13, May 1987.

[ZhRS87a] W. Zhao, K. Ramamrithan, and J. Stankovic, "Preemptive scheduling under time and resource constraints", *IEEE Trans. on Computers*, Aug. 1987.

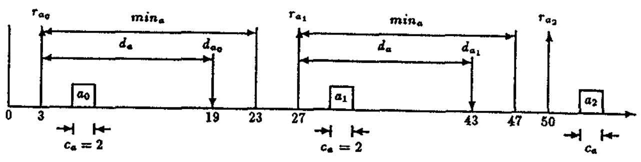
[ZhSR90] W. Zhao, J. Stankovic, and K. Ramamrithan, "A window protocol for time constrained messages," *IEEE Trans. on Computers* vol. 39, Sept. 1990.

Fig. 1.



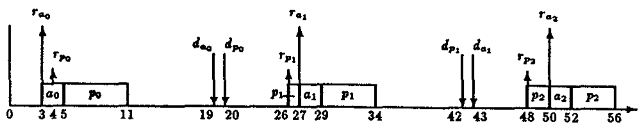
Periodic process (r_p, c_p, d_p, prd_p) where $r_p = 5, c_p = 6, d_p = 20, prd_p = 22$.

Fig. 2.



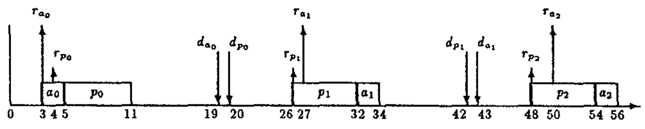
Asynchronous process (c_a, d_a, min_a) where $c_a = 2, d_a = 16, min_a = 20$.
The asynchronous request times are: $r_{a_0} = 3, r_{a_1} = 27, r_{a_2} = 50$.

Fig. 3.



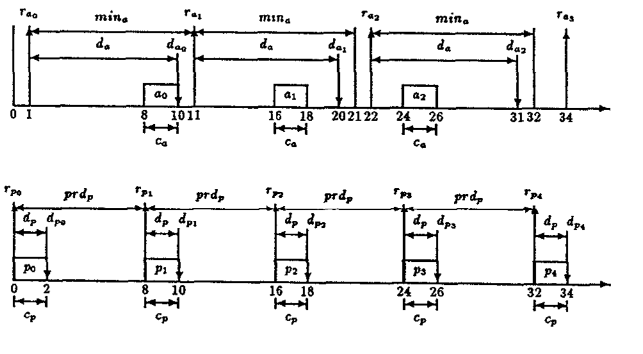
A preemptive schedule for the periodic process executions p_0, p_1, p_2, and the asynchronous process executions a_0, a_1, a_2 in Fig.1 and Fig. 2. within the finite time interval [0,56].
a_1 preempts p_1 at time 27, and a_2 preempts p_2 at time 50.

Fig. 4.



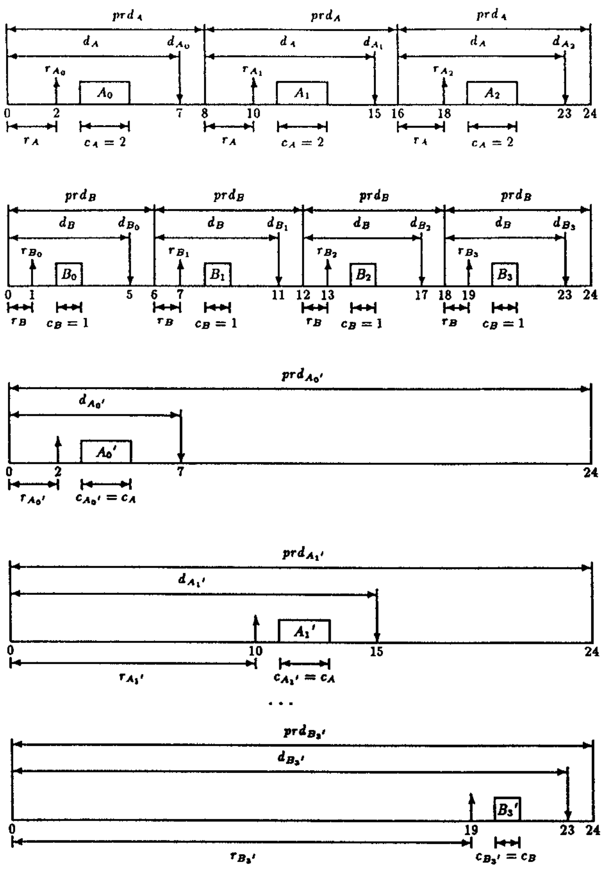
A non-preemptive schedule for the periodic process executions p_0, p_1, p_2 and the asynchronous process executions a_0, a_1, a_2 in Fig 1 and Fig 2 within the finite time interval [0,56]

Fig. 5.


r_{a_0} min_a r_{a_1} min_a r_{a_2} min_a r_{a_3}
d_a d_{a_0} d_a d_{a_1} d_a d_{a_2}
a_0 a_1 a_2
0 1 8 10 11 16 18 20 21 22 24 26 31 32 34
c_a c_a c_a
r_{p_0} prd_p r_{p_1} prd_p r_{p_2} prd_p r_{p_3} prd_p r_{p_4}
d_p d_{p_0} d_p d_{p_1} d_p d_{p_2} d_p d_{p_3} d_p d_{p_4}
p_0 p_1 p_2 p_3 p_4
0 2 8 10 16 18 24 26 32 34
c_p c_p c_p c_p c_p

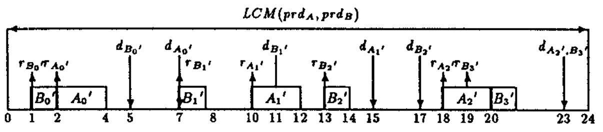
Asynchronous process (c_a, d_a, min_a) where $c_a = 2, d_a = 9, min_a = 10$.
Periodic process (r_p, c_p, d_p, prd_p) translated from the asynchronous process $(c_a, d_a, min_a) = (2, 9, 10)$ where $r_p = 0, c_p = c_a = 2, d_p = c_a = 2, prd_p = min(d_a - d_p + 1, min_a) = min(9 - 2 + 1, 9) = 8$.
If periodic process executions $p_0, p_1, p_2, p_3, p_4, \ldots$ are scheduled to start at time 0, 8, 16, 24, 32, . .,
and if the asynchronous request times $r_{a_0}, r_{a_1}, r_{a_2}$ are 1, 11, 22, then the start times of the asynchronous process executions a_0, a_1, a_0 are 8, 16, 24. a_0 executes in the time slot of p_1,
a_1 executes in the time slot of p_2, a_2 executes in the time slot of p_3.

Fig. 6.



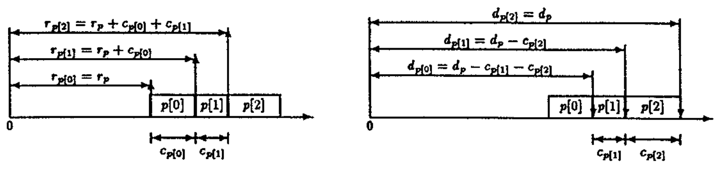
The 2 periodic processes $(r_A, c_A, d_A, prd_A) = (2,2,7,8)$ and $(r_B, c_B, d_B, prd_B) = (1,1,5,6)$ are replaced by 7 new periodic processes with identical periods equal to $LCM(prd_A, prd_B) = 24$: $(r_{A_0'}, c_{A_0'}, d_{A_0'}, prd_{A_0'}) = (2,2,7,24)$, $(r_{A_1'}, c_{A_1'}, d_{A_1'}, prd_{A_1'}) = (10,2,15,24)$, , $(r_{B_3'}, c_{B_3'}, d_{B_3'}, prd_{B_3'}) = (19,1,23,24)$.

Fig. 7.



A feasible schedule in which the timing constraints of all 7 new periodic process executions within one LCM period in Fig. 7 are satisfied also gives a feasible schedule for all the process executions $A_0, A_1, \ldots, B_3$ of the original two periodic processes A and B within one LCM period.

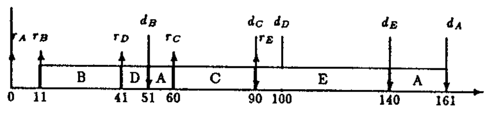
Fig. 8.



$r_{p[2]} = r_p + c_{p[0]} + c_{p[1]}$
$r_{p[1]} = r_p + c_{p[0]}$
$r_{p[0]} = r_p$
p[0]
p[1]
p[2]
$c_{p[0]}$
$c_{p[1]}$
$d_{p[2]} = d_p$
$d_{p[1]} = d_p - c_{p[2]}$
$d_{p[0]} = d_p - c_{p[1]} - c_{p[2]}$
$c_{p[2]}$

Suppose process p consists of 3 segments $p[0]$, $p[1]$ and $p[2]$. Given the release time r_p and deadline d_p of process p and computation time of each segment, i.e., $c_{p[0]}$, $c_{p[1]}$, and $c_{p[2]}$, one can easily compute the release time and deadline for each segment, i.e., $r_{p[0]}$, $r_{p[1]}$, $r_{p[2]}$, $d_{p[0]}$, $d_{p[1]}$, and $d_{p[2]}$

Fig. 9.



A feasible schedule for the 5 process segments A, B, C, D, E, that satisfies the timing constraints. $r_A = 0, c_A = 30, d_A = 161; r_B = 11, c_B = 30, d_B = 51$; $r_C = 60, c_C = 10, d_C = 90, r_D = 41, c_D = 10$, $d_D = 100; r_E = 90, c_E = 50, d_E = 140$, and satisfies the precedence relations: B PRECEDES D, and satisfies the exclusion relations: A EXCLUDES D, A EXCLUDES B

In the first part of Table 1 on the next page, the first three columns refer to algorithms which assume that all processes either have the same release time, or have the same deadline. The fourth and fifth columns refer to algorithms which assume that the number of processors can vary and the algorithms' objective is to minimize the number of processors. The last six columns refer to algorithms that do not have the restrictions mentioned above, but assume that *all* processes are preemptable.

In the second part of the Table 1, the first four columns refer to algorithms which assume that all processes have the same computation time. The last seven columns refer to algorithms that do not have the constraints mentioned above.

Footnotes for Table 1:

[1] Although the algorithm in [ZhRS87] was designed primarily for on-line scheduling of asynchronous processes, an extension to the algorithm to deal with periodic processes is also discussed.
[2] In [Blaz79] each process can have one or zero resources.
[3,4] In [Xu90], [XuPa90] and [XuPa91], asynchronous processes are first translated into equivalent new periodic processes, then the main algorithm is applied to compute a feasible schedule for the instances of all the original and new periodic processes within a time interval between 0 and the least common multiple of the process periods.

Table 1.

Characteristics of scheduling problem		Algorithms					Algorithms					
1) Process Type	a) static	x	x	x				x			x	
	b) periodic				x	x	x					
	c) asynchronous								x	x		
	d) both											x[1]
2) Release Times	a) unknown in adv.								x	x		x
	b) same	x	x									
	c) start of period				x	x	x					
	d) arbitrary			x				x	x	x	x	x
3) Deadlines	a) unknown in adv.								x	x		
	b) same			x								
	c) end of period				x	x	x					
	d) arbitrary	x	x					x	x	x	x	x
4) Process Synchr.	a) preemptive	x		x		x	x	x	x	x	x	x
	b) resource constr.											x
	c) non-preemptive		x		x							
	d) gen. exclusion											
5) Precedence Relations	a) none	x		x	x	x	x	x	x		x	x
	b) tree											
	c) general		x							x		
6) Computation Times	a) uniform											
	b) arbitrary integer							x				
	c) arbitrary real	x	x	x	x	x	x		x	x	x	x
7) Number of Processors	a) 1		x				x	x	x	x		
	b) 2											
	c) n (pre-assigned)											x
	d) n	x		x	x	x					x	
8) Processor Speeds	a) identical				x	x						x
	b) arbitrary	x		x							x	
9) Measures of Performance	a) sched. length											
	b) # of processors				x	x						
	c) lateness	x	x	x			x	x	x	x	x	x
10) Optimality	a) heuristic				x	x						x
	b) optimal	x	x	x			x	x	x	x	x	
		[GoSa78] [SaCh80]	[Blaz76]	[SaCh79]	[GoSo76]	[DhLi78] [DaDh86]	[LiLa73] [Serl72]	[BrFR71]	[Horn74] [MoDe78]	[Blaz76]	[Mart82]	[ZhRS87]

Characteristics of scheduling problem		Algorithms				Algorithms						
1) Process Type	a) static	x	x	x	x	x	x		x		x	x
	b) periodic											
	c) asynchronous							x		x		
	d) both										x[3]	x[4]
2) Release Times	a) unknown in adv.							x		x		
	b) same											
	c) start of period											
	d) arbitrary	x	x	x	x	x	x	x	x	x	x	x
3) Deadlines	a) unknown in adv.							x				
	b) same											
	c) end of period											
	d) arbitrary	x	x	x	x	x	x	x	x	x	x	x
4) Process Synchr.	a) preemptive											
	b) resource constr.				x[2]					x		
	c) non-preemptive	x	x	x	x	x	x	x	x	x	x	
	d) gen. exclusion										x	x
5) Precedence Relations	a) none			x	x	x	x	x	x	x		
	b) tree											
	c) general	x	x								x	x
6) Computation Times	a) uniform	x	x	x	x							
	b) arbitrary integer						x				x	x
	c) arbitrary real					x		x	x	x		
7) Number of Processors	a) 1	x				x	x					x
	b) 2		x									
	c) n (pre-assigned)									x		
	d) n			x	x			x	x		x	
8) Processor Speeds	a) identical			x	x			x	x	x	x	
	b) arbitrary											
9) Measures of Performance	a) sched. length											
	b) # of processors											
	c) lateness	x	x	x	x	x	x	x	x	x	x	x
10) Optimality	a) heuristic							x		x		
	b) optimal	x	x	x	x	x	x		x		x	x
		[GJST81]	[GaJo77] [GaJo76]	[Simo83]	[Blaz79]	[BrFR71] [BaSu74]	[McFl75] [Carl80]	[Guns84]	[BrFR75]	[ZhRS87a]	[Xu90]	[XuPa90] [XuPa91]