5

PLANNING-BASED SCHEDULING

Many real-time applications are deployed in dynamic environments and hence require support for scheduling jobs as they arrive. Dynamic scheduling allows more flexibility in dealing with problems faced in practice, such as the need to alter scheduling decisions based on the occurrence of overloads, e.g., when

- the environment changes,
- there is a burst of job arrivals, or
- a part of the system fails.

If system overloads are assumed to be impossible, then schedulability analysis based on EDF can be used. If overloads do not occur, when a job is preempted there is an implicit guarantee that the remainder of the job will be completed before its deadline. Unfortunately, EDF can rapidly degrade system performance during overloads [23]. The arrival of a new job may result in all the previous jobs missing their deadlines. Such an undesirable phenomenon, called the *Domino Effect*, is depicted in Figure 5.1.

In particular, Figure 5.1a shows a feasible schedule of a job set executed under the EDF scheduling algorithm. However, if at time t_0 job J_0 is executed, all previous jobs miss their deadlines (see Figure 5.1b). In such a situation, EDF does not provide any type of guarantee on which jobs meet their timing constraints. This is a very undesirable behavior in those control applications in which a guarantee of minimum level of performance is necessary. In order to avoid domino effects, the operating system and the scheduling algorithm must be explicitly designed to handle overloads in a more controlled fashion, so that the damage due to a deadline miss can be minimized.

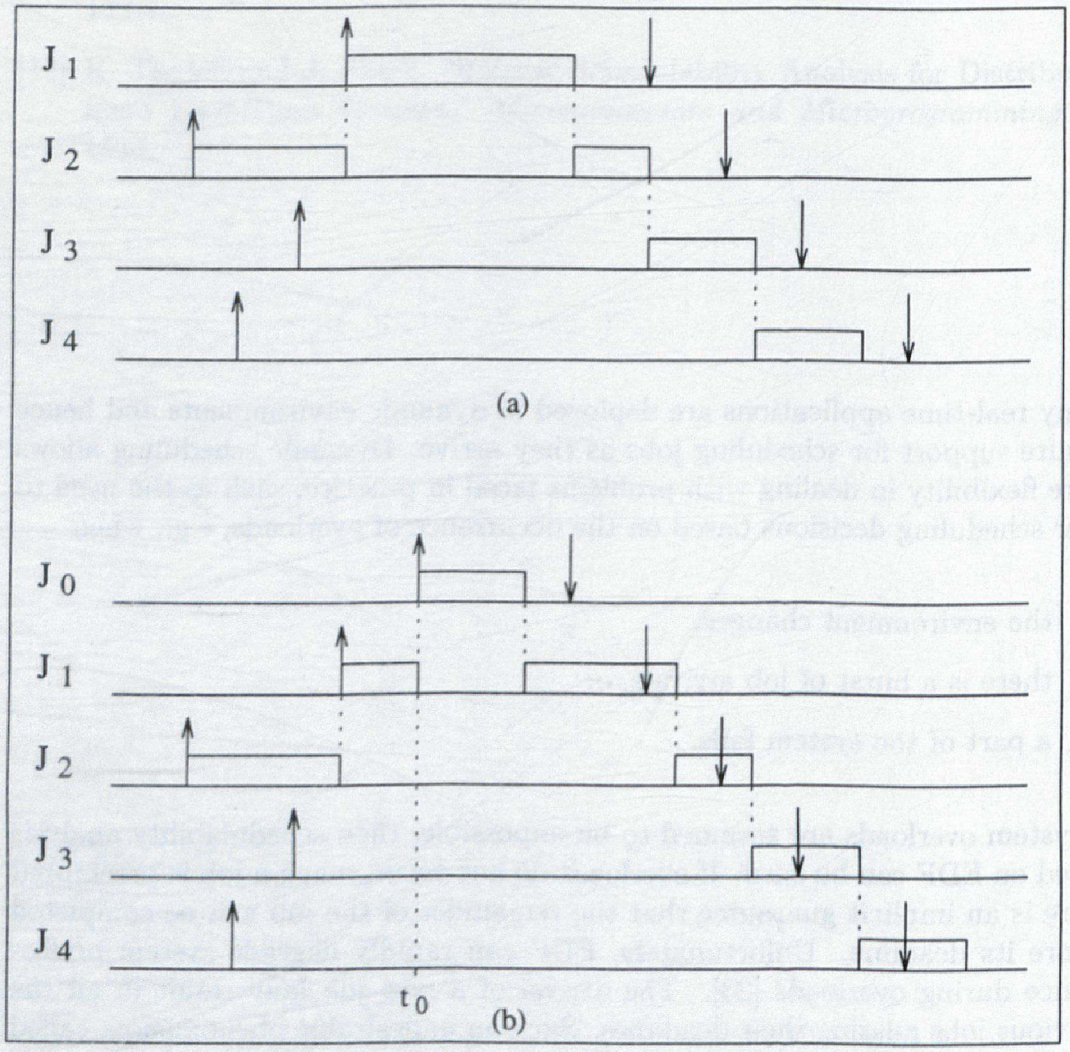

Figure 5.1 (a) Feasible schedule with Earliest Deadline First, in normal load condition. (b) Overload with domino effect due to the arrival of job J_0.

One possibility is to design a system for the worst case loads. Taking this approach it is usually assumed that worst case times are known and, therefore, overloads and failures never occur. Unfortunately, even with this assumption it is often inefficient to determine schedulability or to *a priori* construct a schedule for such a system. Also, in a dynamic system it is not possible to guarantee *a priori* that all job arrivals *will* be able to meet their deadlines: if the arrival times of jobs are not known, the schedulability of all the jobs cannot be guaranteed.

Dynamic planning algorithms are motivated by these practical considerations of dynamic real-time systems. What planning-based algorithms attempt to do is to give assurances to arriving jobs concerning the ability of the system to meet the time constraints associated with the jobs. Generally speaking, planning to determine schedulability is akin to admission control. Depending on the requested Quality of Service (QoS), a planning algorithm can be designed to work with different types of information, from worst case assumptions needed to provide "absolute guarantees" regarding the delivered QoS (even under the most pessimistic assumptions) to guarantees based on the satisfaction of specific conditions. For example, when jobs with different levels of importance are considered, the notion of "conditional guarantee" appears to be applicable whereby a job's guarantee is predicated upon the non-arrival of jobs with higher importance. This is in contrast with absolute guarantees whereby once a job is accepted, its schedulability remains intact under all circumstances.

Independent of the nature of guarantees, the construction of a plan may require assigning priorities to jobs; this raises the question of how priorities are assigned. A simple method is to assign priorities based on EDF: the closer a job's deadline, the higher its priority. For scheduling independent jobs with deadline constraints on single processors, EDF is optimal, so if any assignment of priorities can feasibly schedule such jobs, then so can EDF.

For a given job set, if jobs have the same arrival times but different deadlines, EDF generates a non-preemptive schedule. If both arrival times and deadlines are arbitrary, EDF schedules may require preemptions. EDF uses the timing characteristics of jobs and is suitable when the processor is the only resource needed and jobs are independent of each other.

Of more practical interest is the scheduling of jobs with timing constraints, precedence constraints, resource constraints and arbitrary importance on multiprocessors. Unfortunately, most instances of the scheduling problem for real-time systems are computationally intractable. Non-preemptive scheduling is desirable as it avoids context switching overheads, but determining such a

schedule is an NP-hard problem even on uni-processors when jobs can have arbitrary ready times [14]. The presence of other constraints exacerbates the situation.

This makes it clear that it serves no effective purpose to try to obtain an optimal schedule, especially when decisions are made dynamically. Dertouzos and Mok studied multi-processor on-line scheduling of real-time jobs [27, 12] noting that for most real-world circumstances, optimal dynamic algorithms do not exist [19, 8, 28]. With multi-processors, no dynamic scheduling algorithm is optimal and can guarantee all jobs without prior knowledge of job deadlines, computation times, and arrival times [27]. Such knowledge is not available in dynamic systems so it is necessary to resort to approximate algorithms or to use heuristics to construct the schedules.

Any real-time system must exhibit 'graceful degradation' under failures and overloads. To achieve this, not only must the fact that a job did not meet its deadline be detected, but the fact that this is going to occur must be detected as soon as possible and, by signaling this exception, make it possible for the job to be substituted by one or more contingency jobs. Thus on-line schedulability analysis must have an early warning feature which provides sufficient lead time for the timely invocation of contingency jobs, making it possible for the scheduler to take account of a continuously changing environment.

An advantage of dynamic scheduling is that fairly complex priority assignment policies can be used. This gives dynamic algorithms a lot of flexibility and aids in their ability to deal with a wide variety of job and resource characteristics. But a priority-based scheduler may incur substantial overheads in calculating the priorities of jobs and in selecting the job of highest priority. When dynamic priorities are used, the relative priorities of jobs can change as time progresses, as new jobs arrive, or as jobs execute. Whenever one of these events occurs, the priority of all the remaining jobs must be recomputed. This can make the use of dynamic priorities more expensive in terms of run-time overheads and in practice these overheads must be kept as small as possible.

Trying to minimize scheduling overheads conflicts with the goal of providing for the early warning feature. The earlier an arriving job is checked for feasibility, the sooner it can be known whether it will meet its deadline. However, if it is scheduled early but remains in the system for too long, say because its deadline is far away, then scheduling costs can be high for subsequent scheduling decisions since they must ensure the schedulability of this and other previously accepted jobs.

How practical planning-based approaches address these issues is discussed in subsequent sections. In preparation for this, the definitions of load in dynamic systems and the metrics relevant for dynamic real-time systems are discussed.

5.1 PRELIMINARIES: LOAD, METRICS, VALUE FUNCTIONS

5.1.1 Definition of Load in Dynamic Systems

In a real-time system, the definition of computational workload depends on the temporal characteristics of the computational activities. For non real-time or soft real-time jobs, a commonly accepted definition of workload refers to the standard queuing theory definition. Here the load ρ, also called *traffic intensity*, is:

$$\rho = \lambda C.$$

where C is the mean service time and λ is the average arrival rate of the jobs.

Notice that this definition does not take deadlines into account, hence it is not particularly useful to describe real-time workloads. In a hard real-time environment, a system is overloaded when, based on worst case assumptions, there exists an interval during which the work exceeds capacity, so one or more jobs *might* miss their deadline.

If the job set consists of n independent preemptable periodic tasks, whose relative deadlines are equal to their period, then the system load ρ is equivalent to the processor utilization factor:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i},$$

where C_i and T_i are the computation time and the period of job τ_i respectively. In this case, a load $\rho > 1$ means that the total computation time requested by the periodic activities in their *hyperperiod* $H = lcm(T_1, T_2, \ldots, T_n)$ exceeds the available time on the processor, therefore the job set cannot be scheduled by any algorithm.

A general method for calculating the load in an aperiodic real-time environment has been proposed in [6]. According to this method, the load is computed at each job activation time (r_i), and the number of intervals in which the computation of load is done is limited by the number of job deadlines (d_i). The method for computing the load is based on the consideration that, for a single job J_i, the load is given by the ratio of its computation time C_i and its relative deadline $D_i = d_i - r_i$. For example, if $C_i = D_i$, i.e., the job does not have slack time, the load in the interval $[r_i, d_i]$ is one. When a new job arrives, the load can be computed from the last request time, which is also the current time t, and the longest deadline, say d_n. In this case, the intervals that need to be considered for the computation are $[t, d_1]$, $[t, d_2]$, ... $[t, d_n]$. In general, the processor load in the interval $[t, d_i]$ is given by

$$\rho_i(t) = \frac{\sum_{d_k \leq d_i} c_k(t)}{(d_i - t)},$$

where $c_k(t)$ refers to the remaining execution time of job J_k at time t. Hence, the total load in the interval $[t, d_n]$ can be computed as the maximum among all $\rho_i(t)$, that is:

$$\rho = \max_{i=1,n} \rho_i(t).$$

5.1.2 Performance Metrics under Overloads

When a real-time system is underloaded and dynamic activation of jobs is not allowed, there is no need to consider job's importance in the scheduling policy, since there exist optimal scheduling algorithms that can guarantee a feasible schedule under a given set of assumptions. For example, Dertouzos [11] proved that EDF is an optimal algorithm for preemptive, independent jobs when there is no overload.

On the contrary, when jobs can be activated dynamically and an overload occurs there are no algorithms that can guarantee a feasible schedule of the job set. Since one or more jobs may miss their deadlines, it is preferable, from the viewpoint of achieving graceful degradation, that the less important jobs are the ones that get delayed. Hence, in overload conditions it is important to distinguish between time constraints and importance of jobs. In general, the importance of a job is not related to its deadline or its period, thus a job with a long deadline could be much more important than another one with an earlier deadline. For example, in a chemical process, monitoring the temperature every ten seconds is certainly more important than updating the clock picture on the user console every second. This means that, during an overload involving these

two tasks, it is better to skip one or more clock updates rather than missing the deadline of a temperature reading, since this could have a major impact on the controlled environment.

In order to specify importance, an additional parameter is usually associated with each job, its *value*, that can be used by the system to make scheduling decisions.

5.1.3 Value/Utility Functions

The value associated with a job reflects its importance with respect to the other jobs in the set. The specific assignment depends on the particular application. For instance, there are situations in which the value is set equal to the job computation time; in other cases it is an arbitrary integer number in a given range; in other applications it is set equal to the ratio of an arbitrary number (which reflects the importance of the job) and the job computation time (this ratio is called *value density*).

In a real-time system, however, the actual value of a job also depends on the time at which the job is completed, hence the job's importance can be better described by an utility function. Figure 5.2 illustrates some utility functions that can be associated with a job in order to describe its importance. According to this view, a non-real-time job, which has no time constraints, has a constant (low) value, since it always contributes to the system value whenever it completes its execution. On the contrary, a hard real-time job contributes to a value only if it completes within its deadline and, since a deadline miss would jeopardize the behavior of the whole system, the value after its deadline can be considered minus infinity in many situations. A job with a soft deadline can still give a value to the system if executed after its deadline, although this value may decrease with time. *Firm* real-time activities are those that do not unduly jeopardize the system, but give zero value if completed after their deadline.

Once the importance of each job has been defined, the performance of a scheduling algorithm can be measured by accumulating the values of the job utility functions computed at their completion time. Specifically, the *cumulative value* of a scheduling algorithm A is defined as the following quantity:

$$\Gamma_A = \sum_i v(f_i)$$

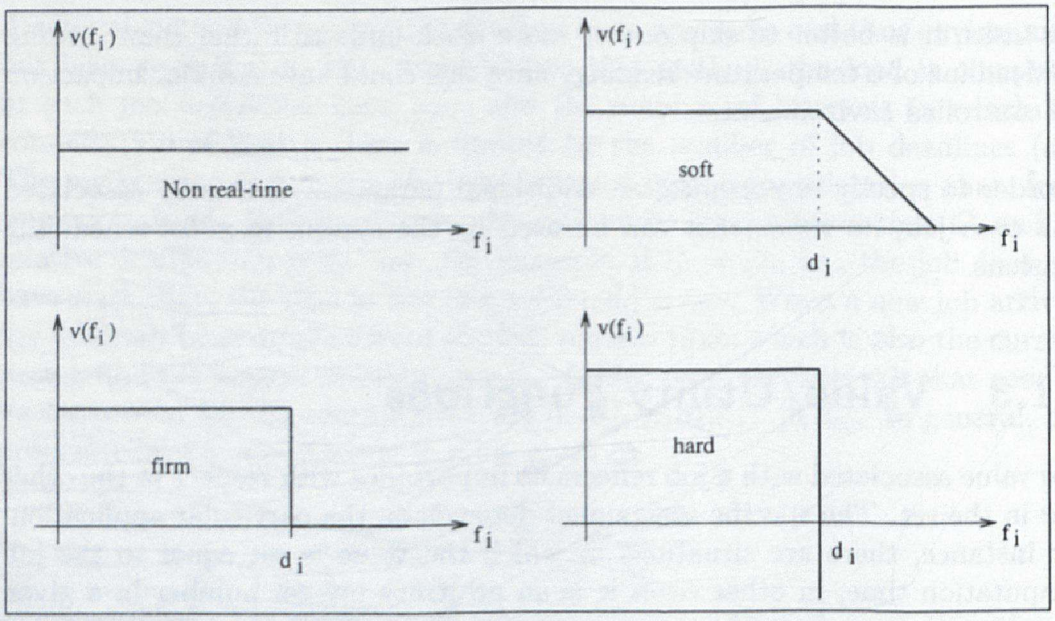

Figure 5.2 Utility functions that can be associated with a job to describe its importance.

Given this metric, a scheduling algorithm is optimal if it maximizes the cumulative value achievable for a job set.

Notice that if a job with a hard constraint misses its deadline, the cumulative value achieved by the algorithm is minus infinity, even though all other jobs are completed before their deadlines. For this reason, all activities with hard time constraints should be guaranteed *a priori* by assigning them dedicated resources (included processors). If all hard deadlines are guaranteed *a priori*, the objective of a real-time scheduling algorithm for soft and firm jobs should be to guarantee a feasible schedule in underload conditions and maximize the cumulative value during overloads. If more general utility functions are possible then it may also be a requirement for soft and firm jobs in underload conditions to maximize the cumulative value.

Given a set of n jobs $J(C_i, D_i, V_i)$, where C_i is the worst case computation time, D_i the relative deadline, and V_i the importance value gained by the system when the job completes within its deadline, the maximum cumulative value achievable on the job set is equal to the sum of all values V_i, i.e., $\Gamma_{max} = \sum_{i=1}^{n} V_i$. In overload conditions, this value cannot be achieved since one or more jobs will miss their deadlines. Hence, if Γ^* is the maximum cumulative value that can be achieved by any algorithm on a job set in overload conditions, the performance

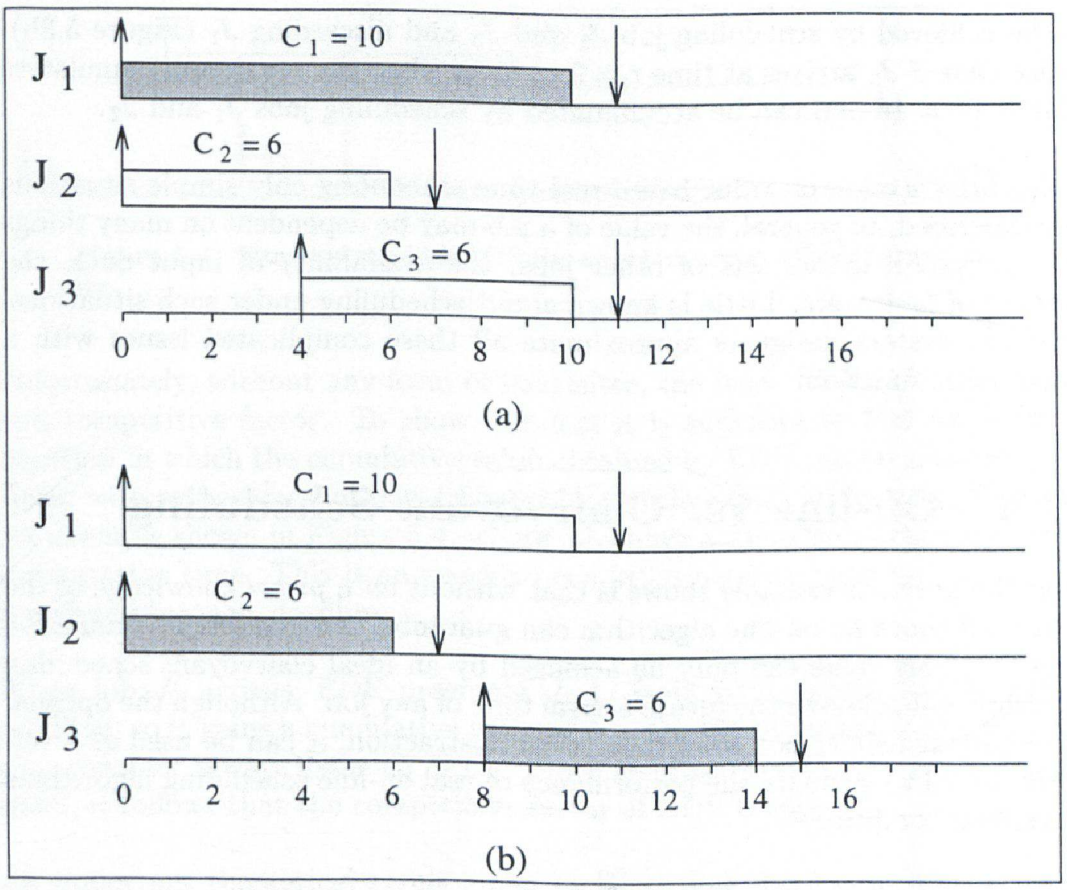

Figure 5.3 No optimal on-line optimal algorithms exist in overload conditions, since the schedule that maximizes Γ depends on the knowledge of future arrivals.

of a scheduling algorithm A can be measured by comparing the cumulative value Γ_A obtained by A with the maximum achievable value Γ^*.

Consider for example the job set shown in Figure 5.3, consisting of three jobs $J_1(10, 11, 10)$, $J_2(6, 7, 6)$, $J_3(6, 7, 6)$.

Without loss of generality, assume that the importance values associated with the jobs are equal to their execution times ($V_i = C_i$) and that jobs are firm, so no value is accumulated if a job completes after its deadline. If J_1 and J_2 simultaneously arrive at time $t_0 = 0$, there is no way to maximize the cumulative value without knowing the arrival time of J_3. In fact, if J_3 arrives at time $t = 4$ or before, the maximum cumulative value is $\Gamma^* = 10$ and can be achieved by scheduling job J_1 (Figure 5.3a). However, if J_3 arrives between time $t = 5$ and time $t = 8$, the maximum cumulative value is $\Gamma^* = 12$ and

can be achieved by scheduling job J_2 and J_3 and discarding J_1 (Figure 5.3b). Notice that if J_3 arrives at time $t = 9$ or later, then the maximum cumulative value is $\Gamma^* = 16$ and can be accumulated by scheduling jobs J_1 and J_3.

In this brief section on value based real-time scheduling only simple situations were described, In general, the value of a job may be dependent on many things such as system mode, sets of other jobs, the availability of input data, the presence of faults, etc. Little is known about scheduling under such situations. Normally, system designers approximate all these complicated issues with a simple value function.

5.1.4 On-line vs. Clairvoyant Scheduling

What the previous example shows is that without an *a priori* knowledge of the job arrival times no on-line algorithm can guarantee the maximum cumulative value Γ^*. This value can only be achieved by an ideal clairvoyant scheduling algorithm which knows the future arrival time of any job. Although the optimal clairvoyant scheduler is a pure theoretical abstraction, it can be used as a reference model to evaluate the performance of real on-line scheduling algorithms in overload conditions.

Definition 1 *A scheduling algorithm A has a* competitive factor φ_A *if and only if it can guarantee a cumulative value*

$$\Gamma_a \geq \varphi_A \Gamma^*$$

where Γ^* *is the cumulative value achieved by the optimal clairvoyant scheduler.*

From the above definition (given in [1]), it is noticed that the competitive factor is a real number $\varphi_A \in [0, 1]$. If an algorithm A has a competitive factor φ_A it means that A can achieve a cumulative value Γ_A at least φ_A times the cumulative value achievable by the optimal clairvoyant scheduler on any job set.

If the overload has an infinite duration, then no on-line algorithm can guarantee a competitive factor greater than zero. In real situations, however, overloads are intermittent and usually have a short duration, hence it is desirable to use scheduling algorithms with a high competitive factor.

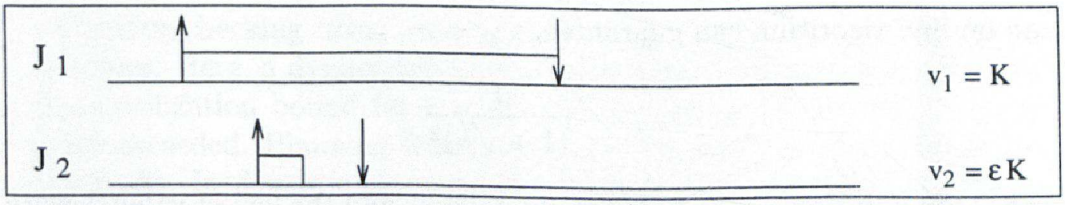

Figure 5.4 Situation in which EDF has an arbitrarily small competitive factor.

Unfortunately, without any form of guarantee, the basic EDF algorithm has a zero competitive factor. To show this fact it is sufficient to find an overload situation in which the cumulative value obtained by EDF can be arbitrarily reduced with respect to that one achieved by the clairvoyant scheduler. Consider the example shown in Figure 5.4, where jobs have a value proportional to their computation time. This is an overload condition because both jobs cannot be completed by their deadline.

When job J_2 arrives, EDF preempts J_1 in favor of J_2 which has an earlier deadline, so it gains a cumulative value of C_2. On the other hand, the clairvoyant scheduler always gains $C_1 > C_2$. Since the ratio C_2/C_1 can be arbitrarily small, it follows that the competitive factor of EDF is zero.

An important theoretical result found in [1] is that there is an upper bound on the competitive factor of any on-line algorithm. In particular the following theorem has been proved.

Theorem 1 *If the job's value is proportional to its computation time, then no on-line algorithm can guarantee a competitive factor greater than* 0.25.

The proof is done by using an adversary argument, in which the on-line scheduling algorithm is identified as a player and the clairvoyant scheduler as the adversary. In order to propose worst case conditions, the adversary dynamically generates the jobs depending on the player decisions. At the end of the game, the adversary shows its schedule and the two cumulative values are computed. Since the player tries to do its best in worst case conditions, the ratio of the cumulative values gives the upper bound of the competitive factor for any on-line algorithm.

Baruah et. al. [1] also showed that, when using value density metrics (where the value density of a job is its value divided by its computation time), the best

that an on-line algorithm can guarantee is

$$\varphi_{max} = \frac{1}{(1+\sqrt{k})^2}$$

where k is the important ratio between the highest and the lowest value density job in the system.

Koren and Shasha [20] also found an on-line scheduling algorithm, called D^{over}, having the best possible competitive factor.

It is worth pointing out, however, that the above bounds are achieved under very restrictive assumptions, such as all jobs have zero laxity, the overload can have an arbitrary (but finite) duration, and job's execution time can be arbitrarily small. In most real world real-time applications, however, jobs' characteristics are much less restrictive and a lot is known about the actual job set. Therefore, the $1/4th$ bound has only a theoretical validity and more work is needed to derive other bounds based on more knowledge of the actual real-time job set of a given system.

5.2 STEPS IN A DYNAMIC PLANNING-BASED SCHEDULING APPROACH

In general, dynamic scheduling has three basic steps: feasibility checking, schedule construction, and dispatching. Dynamic planning based scheduling combines these steps in various ways. Depending on the kind of application for which the system is designed, the programming model adopted, and the scheduling algorithm used, all these steps may not be needed. Often, the boundaries between the steps may also not always be clearly delineated. The basic steps are described first and then how they are combined into planning based scheduling follows.

Feasibility Analysis

Feasibility, or schedulability, analysis is the process of determining whether the timing requirements of a set of jobs can be satisfied, usually under a given set of resource requirements and precedence constraints. Dynamic systems perform feasibility checking on-line, as jobs arrive.

Feasibility checking using schedulability formulae is most suited for periodic activities. Here, a dynamically arriving periodic task is accepted for execution if the utilization bound for the new job as well as the currently existing jobs is not exceeded. Planning-Based approaches provide similar support for aperiodic tasks. In *dynamic planning-based approaches*, execution of a job is begun only if it passes a feasibility test. The feasibility can be based on a model of executing jobs according to the EDF scheduling discipline. Often, a result of the feasibility analysis is a schedule or plan that determines when a job should begin execution.

Schedulability analysis is especially important for activities for which recovery following an abortion after partial execution can be complicated. Error handlers are complicated in general and abnormal termination may produce inconsistent system states. This is likely to be the case especially if the activity involves inter-process interaction. In such situations, it is better to allow an activity to take place only if it can be *guaranteed* to complete by its deadline. If such a guarantee cannot be provided, then the program can perform an alternative action. To provide sufficient time for executing the alternative action, a deadline may be imposed on the determination of schedulability. This can be generalized so that there are N versions of the activity and the algorithm attempts to guarantee the execution of the best possible version. 'Best' refers to the value of the results produced by a particular version; typically, the better the value of the result, the longer the execution time.

Schedule Construction

Schedule construction is the process of ordering the jobs to be executed and storing this in a form that can be used by the dispatching step. Whereas approaches that perform schedulability analysis by checking utilization bounds do not construct explicit schedules, for planning-based approaches, schedule construction is usually a direct consequence of feasibility checking. In the former case, priorities are assigned to jobs and at run time, the job in execution has the highest priority. Planning-Based approaches also can be considered to assign priorities to jobs. These are used to decide which job must be placed next in the plan being constructed. However, the resultant schedule may or may not rely on priorities. For example, it is possible that the final plan is a priority ordered list of jobs, or it may be that the final schedule is a list of start and finish times for each job without any explicit priority identified. In the rest of this chapter the terms *plan* and *schedule* are used interchangeably.

Dispatching

Dispatching is the process of deciding which job to execute next. The complexity and requirements for the dispatching step depend on

1. the scheduling algorithm used in the feasibility checking step;
2. whether a schedule is constructed as part of the schedulability analysis step;
3. the kinds of jobs, e.g., whether they are independent or with precedence constraints, and whether their execution is preemptive or non-preemptive; and
4. the nature of the execution platform, e.g., whether it has one processor or more and how communication takes place.

For example, with non-preemptive scheduling a job is dispatched exactly once; with preemptive scheduling, a job is dispatched once when it first begins execution and again whenever it is resumed.

These three steps are combined in a dynamic planning-based approach as follows. When a job arrives, an attempt is made to *guarantee* the job by constructing a plan for this new job execution to meet its timing constraints and where all previously guaranteed jobs continue to meet their timing constraints. A job is guaranteed subject to a set of assumptions, for example, about its worst case execution time and resource needs, overhead costs and the nature of the faults in the system. If these assumptions hold, once a job is guaranteed it *will* meet its timing requirements. Thus, feasibility is checked with each arrival.

If the attempt to guarantee fails, the job is not feasible and a *timing fault* is forecast. If this is known sufficiently ahead of the deadline, there may be time to take alternative actions. For example, it may be possible to trade off quality for timeliness, by attempting to schedule an alternative job which has a shorter computation time or less resource needs. In a distributed system, it may be possible to transfer the job (or an alternative job) to a less-loaded node. The alternative job must itself be guaranteed to avoid its impacting previously guaranteed work.

In the rest of this chapter, issues underlying planning-based scheduling are discussed followed by the details of plan construction. Since the run-time cost of a dynamic approach is an important practical consideration, several techniques are discussed for efficient dynamic scheduling.

5.3 ALGORITHMS FOR DYNAMIC PLANNING

Most of the scheduling algorithms proposed in the literature use one of the following scheduling schemes, also illustrated in Figure 5.5. Out of these, the second and third schemes predict and handle overloads by controlling the entry of new jobs and are the subject of this section.

1. **Best Effort Scheme.** This scheme includes those algorithms with no prediction for overload conditions. At its arrival, a new job is always accepted into the ready queue, so the system performance can only be controlled through a proper priority assignment.
2. **Admission Control Scheme.** This scheme includes those algorithms in which the load on the processor is controlled by an acceptance test executed at each job arrival. Typically, whenever a new job enters the system, a guarantee routine verifies the schedulability of the job set based on worst-case assumptions. If the job set is found schedulable, the new job is accepted in the ready queue; otherwise, it is rejected.
3. **Robust Scheme.** This scheme includes those algorithms that separate timing constraints and importance by considering two different policies: one for job acceptance and one for job rejection. Typically, whenever a new job enters the system, an acceptance test verifies the schedulability of the new job set based on worst-case assumptions. If the job set is found schedulable, the new job is accepted in the ready queue; otherwise, one or more jobs are rejected based on a different policy.

The admission control scheme is able to avoid domino effects by sacrificing the execution of newly arriving jobs. Basically, the acceptance test acts as a filter that controls the load on the system. Once a job is accepted, the algorithm guarantees that it will complete by its deadline (assuming that no job will exceed its estimated worst-case computation time). Admission control schemes, however, do not take job's importance into account and, during overloads, always reject the newly arrived job, regardless of its value. In certain conditions (such as when jobs have very different importance levels), this scheduling strategy may exhibit poor performance in terms of cumulative value, whereas a robust algorithm can be much more effective.

In this section, two algorithms based on a dynamic planning approach are described. The first deals with jobs having deadline constraints and where

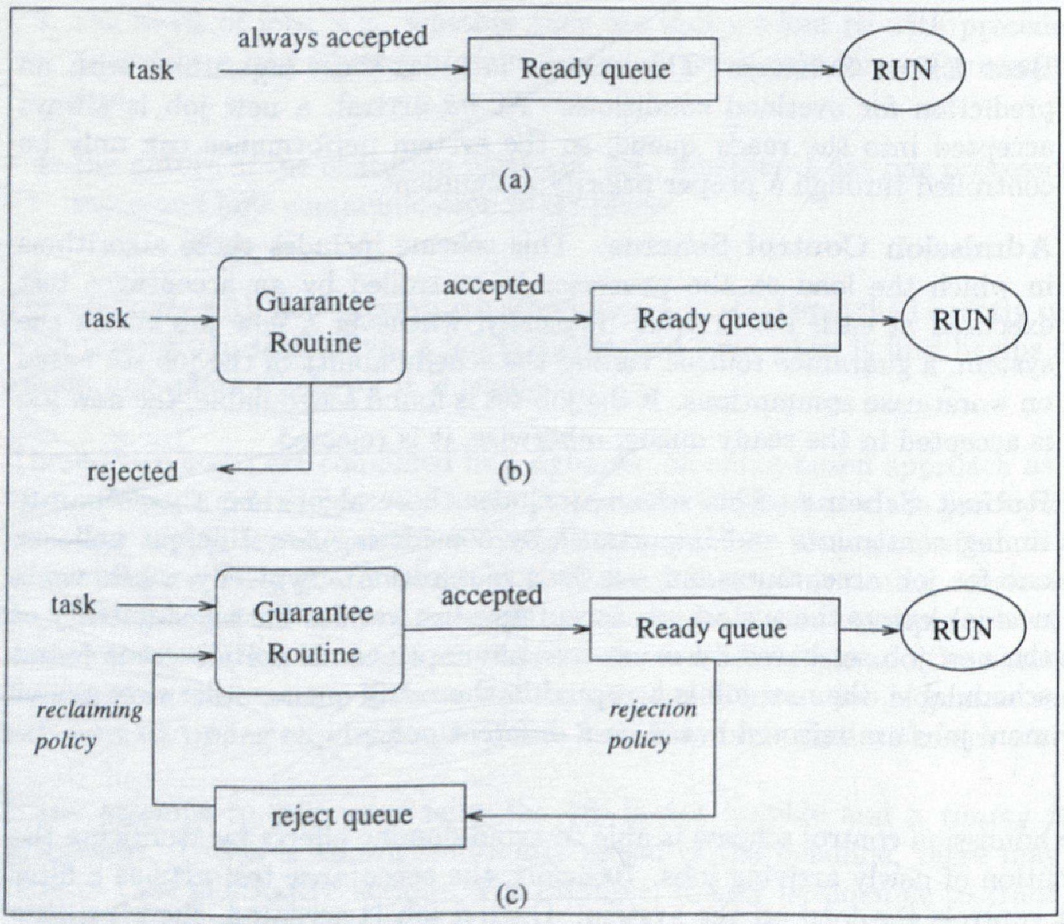

Figure 5.5 Scheduling schemes for handling overload situations. **a.** Best Effort. **b.** Admission Control. **c.** Robust.

each job also has a deadline tolerance. This algorithm incorporates features to select and reject less important tasks when an important job arrives but cannot be admitted because of current load conditions. The second algorithm deals with jobs having deadline and resource constraints. Other related algorithms together with their simulation results and performance comparisons can be found in [5].

5.3.1 The RED Algorithm

RED (Robust Earliest Deadline) [6] is a robust scheduling algorithm for dealing with firm aperiodic tasks in overloaded environments. The algorithm synergistically combines many features including graceful degradation in overloads, deadline tolerance, and resource reclaiming. It operates in normal and overload conditions and is able to predict not only deadline misses, but also the size of the overload, its duration, and its overall impact on the system.

In RED, each job instance $J_i(C_i, D_i, M_i, V_i)$ is characterized by four parameters: a worst case execution time (C_i), a relative deadline (D_i), a deadline tolerance (M_i), and an importance value (V_i). The deadline tolerance is the amount of time by which a job is permitted to be late, i.e., the amount of time that a job may execute after its deadline and still produce a valid result. This parameter can be useful in many real applications, such as robotics and multimedia systems, where the deadline timing semantics is more flexible than scheduling theory generally permits.

Deadline tolerances also provide a sort of compensation for the pessimistic evaluation of the worst case execution time. For example, without tolerance, it could be that a job set is not feasibly schedulable, and hence the job is rejected. But, in reality, the jobs could have been scheduled within the tolerance levels. Another positive effect of tolerance is that various jobs could actually finish before their worst case times, so a resource reclaiming mechanism could then compensate and the jobs with tolerance could actually finish on time.

In RED, the primary deadline plus the deadline tolerance (which provides a sort of secondary deadline), are used to run the acceptance test in overload conditions. Notice that having a tolerance greater than zero is different than having a longer deadline. In fact, jobs are scheduled based on their primary deadline, but accepted based on their secondary deadline. In this framework, a schedule is said to be *strictly feasible* if all jobs complete before their primary

deadline, while is said to be *tolerant* if there exists some job that executes after its primary deadline, but completes within its secondary deadline.

The acceptance test performed in RED is formulated in terms of residual laxity. The residual laxity L_i of a job is defined as the interval between its estimated finishing time (f_i) and its primary (absolute) deadline (d_i). Each residual laxity can be efficiently computed using the result of the following lemma.

Lemma 5.1 *Given a set $J = \{J_1, J_2, \ldots, J_n\}$ of active aperiodic tasks ordered by increasing primary (absolute) deadline, the residual laxity L_i of each job J_i at time t can be computed as:*

$$L_i = L_{i-1} + (d_i - d_{i-1}) - c_i(t). \tag{5.1}$$

where $L_0 = 0$, $d_0 = t$ (i.e., the current time), and $c_i(t)$ is the remaining worst case computation time of job J_i at time t.

Proof. By definition, a residual laxity is $L_i = d_i - f_i$. Since jobs in the set J are ordered by increasing deadlines, job J_1 is executing at time t, and its estimated finishing time is given by the current time plus its remaining execution time ($f_1 = t + c_1$). As a consequence, L_1 is given by:

$$L_1 = d_1 - f_1 = d_1 - t - c_1.$$

Any other job J_i, with $i > 1$, starts as soon as J_{i-1} completes and finishes after c_i units of time from its start ($f_i = f_{i-1} + c_i$). Hence,

$$\begin{aligned} L_i &= d_i - f_i = d_i - f_{i-1} - c_i = d_i - (d_{i-1} - L_{i-1}) - c_i = \\ &= L_{i-1} + (d_i - d_{i-1}) - c_i \end{aligned}$$

and the lemma follows. □

Notice that if the current job set J is schedulable and a new job J_a arrives at time t, the acceptance test of the new job set $J' = J \cup \{J_a\}$ requires the computation of only the residual laxity of job J_a and one of the jobs J_i such that $d_i > d_a$. This is because the execution of J_a does not influence those jobs having deadlines less than or equal to d_a, which are scheduled before J_a. It follows that the acceptance test has $O(n)$ complexity in the worst case.

To simplify the description of the RED acceptance test, the *Exceeding Time* E_i is defined as the time that job J_i executes after its secondary deadline[1]:

$$E_i = \max(0, -(L_i + M_i)). \tag{5.2}$$

[1] If $M_i = 0$, the *Exceeding Time* is also called *Tardiness*.

```
Algorithm RED_acceptance_test(J, J_new)
{
    E = 0;                          /* Maximum Exceeding Time */
    L_0 = 0;
    d_0 = current_time();

    J' = J ∪ {J_new};
    k = <position of J_new in the job set J'>;

    for each job J'_i such that i ≥ k do {
        /* compute the maximum exceeding time */
        L_i = L_{i-1} + (d_i − d_{i-1}) − c_i;
        if (L_i + M_i < −E) then E = −(L_i + M_i);
    }

    if (E > 0) {
        <select a set J* of least value jobs to be rejected>;
        <reject all job in J*>;
    }
}
```

Figure 5.6 The RED acceptance test.

The *Maximum Exceeding Time* E_{max} is defined as the maximum among all E_i's in the jobs set, that is: $E_{max} = \max_i(E_i)$. Clearly, a schedule is strictly feasible if and only if $L_i \geq 0$ for all jobs in the set, while it is tolerant if and only if there exists some $L_i < 0$, but $E_{max} = 0$.

The RED algorithm uses the acceptance test (shown in Figure 5.6) to determine if a job J_{new} is likely to miss its deadline. If so, it computes the amount of processing time required above the capacity of the system – the maximum exceeding time. Otherwise, the job is accepted and executes according to the EDF policy.

The global view provided by the maximum exceeding time allows the planning of an action to recover from the overload condition that would occur if the job is accepted. Many recovering strategies can be used to solve this problem [2]. The simplest one is to reject the least value job that can remove the overload situation. In general it is assumed that, whenever an overload is detected, some

rejection policy searches for a subset J^* of least value jobs that are rejected to maximize the cumulative value of the remaining subset.

A resource reclaiming mechanism can [35] be used to take advantage of those jobs that complete before their worst case finishing time. To reclaim the spare time, rejected jobs are not removed, but temporarily parked in a queue, called the *Reject Queue*, ordered by decreasing values. Whenever a running job completes its execution before its worst case finishing time, the algorithm tries to reaccept the highest value jobs in the Reject Queue having positive laxity. Jobs with negative laxity are removed from the system.

5.3.2 The Spring Algorithm

This section describes an admission control algorithm that can also accommodate resource requirements of jobs beyond CPU resources. Admission is granted if a schedule can be constructed, that is, execution can be planned for a given set of jobs [32, 3] such that they meet their deadlines.

Schedule construction can be viewed as a search for a feasible schedule in a tree in which the leaves represent schedules, some of which may be feasible. The root is the empty schedule. An internal node is a partial schedule for a job set with one more job than that represented by its parent. Given the NP-completeness of the scheduling problem, it would serve little purpose to search exhaustively for a feasible schedule. So heuristics are used to direct the search. A planning algorithm [32, 41] starts at the root of the search tree and repeatedly tries to extend the current partial plan (with one more job) by moving to one of the vertices at the next level in the search tree until a full feasible schedule is derived.

The rest of the section examines the details of this planning-based admission control algorithm, as implemented in the Spring Kernel [38], given the arrival or release time r, deadline d, and worst case computation time C of jobs. Jobs require one CPU. For generality we assume a multi-processor system with m processors and for simplicity, first consider the provision of absolute guarantees. In subsequent chapters when precedence and resource constraints are examined, the basic search algorithm is extended to deal with these additional considerations. Also, jobs are scheduled to execute non-preemptively.

The algorithm computes the earliest start time, est_i, at which job J_i can begin execution after accounting for processor availability given jobs scheduled thus

far. Given a partial schedule, the earliest available time for a resource (which is cpu_j in this case) is given as, $erat_j$. This time can be determined after each job is assigned to a resource for its worst case duration. Then the earliest time that a job J_i that is yet to be scheduled can begin execution is

$$est_i = max(r_i, min_{j=1..m} erat_j)$$

Even though for jobs which need just the cpu, EDF is a good priority assignment policy, later when resources beyond the cpu are considered, a more sophisticated priority assignment policy becomes necessary. Hence, for generality it is assumes that each job J_i has a priority computed dynamically and denoted by $Pr(J_i)$.

At each level of the search, Pr is computed for all the jobs that remain to be scheduled. The job with the highest priority is selected to extend the current partial schedule.

While extending the partial schedule at each level of search, the algorithm determines if the current partial schedule is *strongly-feasible* or not. A partial feasible schedule is said to be *strongly-feasible* if *all* the schedules obtained by extending this current schedule with any one of the remaining jobs are also feasible. Thus, if a partial feasible schedule is found not to be *strongly-feasible* because, say, job J misses its deadline when the current partial schedule is extended by J, then it is appropriate to stop the search since none of the future extensions involving job J will meet its deadline. In this case, a set of jobs cannot be scheduled given the current partial schedule. (In the terminology of branch-and-bound techniques, the search path represented by the current partial schedule is *bounded* or pruned since it does not lead to a feasible complete schedule.)

However, it is possible to backtrack to continue the search even after a non-strongly-feasible schedule is found. Backtracking is done by discarding the current partial schedule, returning to the previous partial schedule, and extending it with a different job, e.g., the job with the *second* highest priority. When backtracking is used, the overheads can be restricted either by restricting the maximum number of possible backtracks or the total number of re-evaluations of priorities.

The fact that priority is computed for all remaining jobs at each level makes it a

$$n + (n-1) + ... + 2 \;=\; O(n^2)$$

search algorithm where n is the number of jobs in the set. The complexity can be reduced to $O(n)$ if only a maximum of k jobs that remain to be scheduled at each level of search are considered [32]. These k jobs can be selected by taking the k jobs with the earliest deadlines. k is a constant (in practice it is small when compared to n). In both cases, the job with the highest priority is selected to extend the current schedule.

Here is a description of the complete algorithm (see Fig. 5.7). Besides Pr, introduced earlier, the following variables are useful in precisely describing the algorithm's steps.

- TR, the jobs that remain to be scheduled, in order of increasing deadline;
- $\mathcal{N}(TR)$, the number of jobs in TR;
- $\mathcal{M}(TR)$, the maximum number of jobs considered by each step of scheduling;
- N_{TR}, the actual number of jobs in TR considered at each step of scheduling, where

  $N_{TR} = \mathcal{M}(TR)$, if $\mathcal{N}(TR) \geq \mathcal{M}(TR)$

  $N_{TR} = \mathcal{N}(TR)$, otherwise

  and
- TC, the first N_{TR} jobs in TR.

The algorithm starts with an empty partial schedule. At each step, the *est* for each job is first computed. To determine the job with the highest priority the priority value for each job is computed next. As a prerequisite for extending the partial schedule with the job with the highest priority, strong-feasibility is determined with respect to all the jobs in TC. After a job J is selected to extend the current partial schedule, its scheduled start time *sst* is set equal to *est* of J and resource availability vectors are updated.

So far, the jobs were assumed to be independent and had just deadline and release time specifications. The extensions necessary to deal with periodic tasks are considered now. Extensions to the basic planning-based approach needed to deal with resources other than CPUs are discussed in Chapter 6, precedence constraints are considered in Chapter 7, and for distributed systems in Chapter 10.

```
TR := job set to be scheduled;
partial schedule := empty;
Result := Success;

while TR ≠ empty ∧ Result ≠ Failure do
TC := first N_TR jobs in TR;
Given a partial schedule
   est calculation:
      for each job J in TR Compute est;
   Priority value generation:
      for each job J in TR Compute Pr(J);
   Job selection:
      find job min_J with highest priority in TC;
   Update partial schedule or backtrack:
      if (partial schedule ⊕ min_J) is feasible and strongly feasible
         partial schedule := (partial schedule ⊕ min_J);
         TR := TR ⊖ min_T;
         Update resource availability vector;
      else if backtracking is allowed and possible
         backtrack to a previous partial schedule;
         choose a job not yet chosen;
      else Result:=Failure
```

Figure 5.7 Basic guarantee algorithm.

There are several ways of guaranteeing periodic tasks when they are executed together with aperiodic tasks. Assume that when a periodic job is guaranteed, every instance of the task is guaranteed.

Consider a system with only periodic tasks. A schedule can be constructed using the basic planning algorithm; given n periodic tasks with periods $T_1 \ldots T_n$,

$$length\ of\ the\ schedule\ =\ LCM(T_1, ..., T_n).$$

The earliest start time of the j^{th} release of the i^{th} job is $(j-1) \times T_i$ and its deadline is $j \times T_i$. In other words, all instances of the periodic tasks are created with release times and deadlines and the entire set is handed to the planning algorithm at once.

If a periodic task arrives dynamically, an attempt can be made to construct a new template. The new task is guaranteed if the attempt succeeds. This new template begins execution at the end of the current LCM of the previous periodic tasks. If this is too long to wait, it is possible to modify this algorithm to handle a quicker mode change between the two templates. This is not discussed further in this book.

Suppose there are periodic *and* aperiodic tasks in the system. If the resources needed by the two sets of tasks are disjoint then the processors in the system can be partitioned, with one set used for the periodic tasks. The remaining processors are used for aperiodic tasks guaranteed using the dynamic planning algorithm.

If however, periodic and aperiodic tasks need common resources, a more complicated scheme is needed. If a periodic task arrives in a system consisting of previously guaranteed periodic and aperiodic tasks, an attempt is made to construct a new schedule. If the attempt fails, the new task is not guaranteed and its introduction has to be delayed until either the guaranteed aperiodic tasks complete or its introduction does not affect the remaining guaranteed jobs.

Suppose a new aperiodic task arrives. Given a schedule for periodic tasks, the new task can be guaranteed if there is sufficient time in the idle slots of the template. Alternatively, applying the dynamic guarantee scheme, an aperiodic task can be guaranteed if all releases of the periodic tasks and all previously guaranteed aperiodic tasks can also be guaranteed.

So far all jobs had been assumed to have the same level of importance. In Biyabani et. al. [2] the planning-based algorithm was extended to deal with jobs having different values, and various policies were studied to decide which

jobs should be dropped when a newly arriving job could not be guaranteed. This referenced work extends the algorithm described thus far to become more *robust.*

5.4 TIMING OF THE PLANNING

As the number of jobs increases, so does the cost of planning and there is less time available for planning. Needless to say, planning-based schemes must be cognizant of time available for planning. So when a system overload is anticipated, use of a method that controls scheduling overheads is essential. Thus, it is important to address the issue of *when* to plan the execution of a newly arrived job. Two simple approaches are:

1. when a job arrives, attempt to plan its execution along with previously scheduled jobs: this is *scheduling-at-arrival-time* and all jobs that have not yet executed are considered for planning when a new job arrives;
2. postpone the feasibility check until a job is chosen for execution: this is *scheduling-at-dispatch* time and can be done very quickly for non-preemptive job execution by checking whether the new job will finish by its deadline.

The second approach is less flexible and announces job rejection very late. Consequently, it does not provide sufficient lead time for considering alternative actions when a job cannot meet its timing constraints. Both avoid resource wastage since a job does not begin execution unless it is known that it will complete before its deadline.

To minimize scheduling overhead while giving enough lead time to choose alternatives, instead of scheduling jobs when they arrive or when they are dispatched, they should be scheduled somewhere in between – at the most opportune time. They can be scheduled at some *punctual point* which limits the number of jobs to be considered for scheduling and avoids unnecessary scheduling (or rescheduling) of jobs that have no effect on the order of jobs early in the schedule.

Choice of the punctual point must consider the fact that the larger the mean laxity and the higher the load, the more jobs are ready to run. The increasing number of jobs imposes growing scheduling overhead for all except a scheduler with constant overheads. The punctual point is the minimum laxity value,

i.e., the value to which a job's laxity must drop before it becomes eligible for scheduling. In other words, the guarantee of a job with laxity larger than the punctual point is postponed *at most* until its laxity reaches the punctual point. Of course, if the system is empty a job becomes eligible for scheduling by default. By postponing scheduling decisions, the number of jobs scheduled at any time is kept under control, reducing the scheduling overhead and potentially improving the overall performance.

The main benefit of scheduling using punctual points is the reduced scheduling overhead when compared to scheduling at arrival time. This is due to the smaller number of relevant jobs (the jobs with laxities smaller than or equal to the punctual point) that are scheduled at any given time. Clearly, when the computational complexity of a scheduling algorithm is higher than the complexity of maintaining the list of relevant jobs, the separation into relevant/irrelevant jobs reduces the overall scheduling cost; that is, the scheduling becomes more efficient.

Consider the following scheme for jobs with deadlines that are held on a dispatch queue, $Q_1(n)$, maintained in minimum laxity order, and a variant of the $FCFS$ queue. When a job arrives, its laxity is compared with that of the n jobs in the queue $Q_1(n)$ and the job with the largest laxity among the $n+1$ jobs is placed at the end of the $FCFS$ queue. When a job in Q_1 is executed, the first job on the $FCFS$ queue is transferred to Q_1. Analysis [15, 18, 29, 30] shows that performance to within 5% of the optimal LLF algorithm is achieved for even small values of n.

A more experimental way to limit the number of scheduled jobs is to have a HIT queue and a $MISS$ queue [17]: the number of scheduled jobs in the HIT queue is continuously adjusted according to the the ratio of jobs that complete on time (the 'hit' ratio). This method is adaptive, handles deadlines and values, and is easy to implement. However, it does not define a punctual point.

The weakness of both of these approaches is the lack of analytical methods to adjust the number of scheduled jobs. The parameters that control the number of schedulable jobs must be obtained through simulation and a newly arrived job can miss its deadline before it gets considered for execution. In contrast, if the punctual point is derived analytically, then it can be ensured that every arrived job will be considered for execution [42].

The number of schedulable jobs must be controlled using timing constraints, rather than by explicitly limiting the number of schedulable jobs; this ensures that every job is considered for scheduling when its laxity reaches the most

opportune moment, the punctual point. The approach is especially beneficial for systems where jobs have widely different values and rejecting a job without considering it for scheduling might result in a large value loss, something that can happen easily when the number of schedulable jobs is fixed.

The features of a 'well-timed scheduling framework' are summarized below.

- Newly arrived jobs are classified as *relevant* or *irrelevant*, depending on their laxity.
- Irrelevant jobs are stored in a D-queue (the delay queue), where they are delayed until their laxity becomes equal to the punctual point, at which time they become relevant.
- Relevant jobs are stored in an S-pool (the scheduling pool) as jobs eligible for immediate scheduling.
- When a job is put into the S-pool, a feasibility check is performed; if this is satisfied, it is transferred into the current feasible schedule. Otherwise, it can be placed in the reject queue to await possible resource reclamation.

It is important to observe that apart from reducing the scheduling cost, the separation of relevant and irrelevant jobs also contributes to the reducing scheduling overhead due to queue handling operations. A simple analytical model is developed in [42], but a formally derived punctual point awaits further work.

5.5 IMPLEMENTING PLANNING-BASED SCHEDULING

In implementing planning-based scheduling, there are two main considerations: feasibility checking and schedule construction. In a multi-processor system, feasibility checking and dispatching can be done independently, allowing these system functions to run in parallel. The dispatcher works with a set of jobs that have been previously guaranteed to meet their deadlines and feasibility checking is done on the set of currently guaranteed jobs plus any newly invoked jobs. See the Spring kernel [38] for a discussion on how to implement this parallelism in a predictable manner and to avoid race conditions.

One of the crucial issues in dynamic scheduling is the cost of scheduling: the more time that is spent on scheduling the less there is for job executions.

In a single processor system, feasibility checking and job executions compete for processing time. If feasibility checking is delayed, there is less benefit from the early warning feature. However, if feasibility checking cannot be performed immediately after a job arrives it may lead to guaranteed jobs missing their deadlines. Thus, when jobs are guaranteed, some time must be set aside for scheduling-related work and a good balance must be struck depending on job arrival rates and job characteristics such as computation times.

One way is to provide for the periodic execution of the scheduling activity. Whenever invoked, the scheduler attempts to guarantee all pending jobs. In addition, if needed, the scheduler could be invoked sporadically whenever these extra invocations affect neither guaranteed jobs nor the minimum guaranteed periodic rate of other system jobs.

Another alternative, applicable to multi-processor systems, is to designate a scheduling processor whose sole responsibility is to deal with feasibility checking and schedule construction. Guaranteed jobs are executed on the remaining 'application' processors. In this case, feasibility checking can be done concurrently with job execution. Recall that a job is guaranteed as long as it can be executed to meet its deadline and the deadlines of previously guaranteed jobs remain guaranteed. Guaranteeing a new job might require re-scheduling of previously guaranteed jobs and so care must be taken to ensure that currently running jobs nor jobs that might execute prior to the guarantee algorithm completing are not re-scheduled.

These considerations suggests that scheduling costs should be computed based on the total number of jobs in the schedule plus the newly arrived jobs, the complexity of the scheduling algorithm and the cost of scheduling one job. Jobs with scheduled start times before the current time plus the scheduling cost are not considered for rescheduling; the remaining jobs are candidates for re-scheduling to accommodate new jobs.

5.6 DISPATCHING JOBS IN A PLANNING-BASED SCHEDULE

Planning-based schedulers typically use non-preemptive schedules. Dispatching depends on whether the jobs are independent and whether there are resource constraints.

If the jobs are independent and have no resource constraints, dispatching can be extremely simple: the job to be executed next is the next job in the schedule, and this job can always be executed immediately even if its scheduled start time has not arrived. Note that a scheduled start time (when the job is actually scheduled to run) is not the same as a job release time (which is the earliest time a job is eligible to run).

On the other hand, precedence constraints and resource constraints may increase the complexity of dispatching. If jobs have resource constraints and/or precedence constraints, the dispatching process must take these into account. When the actual computation time of a job differs from its worst case computation time in a non-preemptive multi-processor schedule with resource constraints, run time anomalies [13, 14] may occur, causing some of the scheduled jobs to miss their deadlines. There are two possible kinds of dispatchers.

1. Dispatch jobs exactly according to the given schedule. In this case, upon the completion of one job, the dispatcher may not be able to immediately dispatch another job because idle time intervals may have been inserted by the scheduler to conform to the precedence constraints, release times, or resource constraints. One way to construct a correct dispatcher is to use a hardware (count down) timer in order to enforce the start time constraint.
2. Dispatch jobs taking into consideration the fact that, given the variance in jobs' execution times, some jobs complete earlier than expected. The dispatcher tries to reclaim the time left by early completion and uses it to execute other jobs.

Clearly, non-real-time jobs which do not use resources needed by the real-time jobs can be executed in idle time slots. More valuable is an approach that improves the guarantees of jobs that have time constraints. Complete rescheduling of all remaining jobs is an available option, but given the complexity of scheduling, it is usually expensive and ineffective. Resource reclaiming algorithms used in systems that do dynamic planning-based scheduling must maintain the feasibility of guaranteed jobs, must have low overheads as a resource reclaiming algorithm is invoked whenever a job finishes, and must have costs that are independent of the number of jobs in the schedule. They must also be effective in improving the performance of the system. Simple but effective resource reclaiming algorithms are described in [35] for independent jobs having resource requirements and in [25] for jobs having, in addition, precedence constraints.

5.7 SUMMARY

While utilization bounds analyses support predictable job executions, their applicability is restricted to jobs whose release times are known *a priori*. Best-effort approaches are applicable to jobs with arbitrary needs, but do not offer predictability. Planning-based scheduling offers the best of both worlds. It must be mentioned that the predictability they offer is on a per-job basis. What is needed is a definition and determination of a global system-wide formal schedulability notion based on individual guarantees.

This Chapter presented a planning-based paradigm for scheduling jobs and presented two different algorithms: the RED algorithm and the Spring algorithm. Since overload handling is such an important issue, this Chapter ends with a brief look at other work related to overloads.

In 1986, Locke [23] developed an algorithm which makes a best effort at scheduling jobs based on earliest deadline with a rejection policy based on removing jobs with the minimum value density. He also suggested that removed jobs remain in the system until their deadline has passed. The algorithm computes the variance of the total slack time in order to find the probability that the available slack time is less than zero. The calculated probability is used to detect a system overload. If it is less than the user prespecified threshold, the algorithm removes the jobs in increasing value density order. Real-time Mach [40] uses an approach similar to this: jobs are ordered by EDF and overload was predicted using a statistical guess. If overload is predicted, jobs with least value are dropped.

In other related work, Sha and his colleagues [37] showed that the rate monotonic algorithm has poor properties under overload. Thambidurai and Trivedi [39] studied overloads in fault tolerant real-time systems, building and analyzing a stochastic model for such a system. However, they provided no details on the scheduling algorithm itself.

Finally, Haritsa, Livny and Carey [17] presented the use of a feedback controlled EDF algorithm for use in real-time database systems. The purpose of their work was to obtain good average performance for transactions even in overload. Since they were working in a database environment they assumed no knowledge of transaction characteristics and they considered jobs with soft deadlines that are not guaranteed.

REFERENCES

[1] 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," *Real-Time Systems*, 4(2), June 1992.

[2] S. Biyabani, J. Stankovic and K. Ramamritham, "The Integration of Deadline and Criticalness in Hard Real-Time Scheduling," *Proceedings of the Real-Time Systems Symposium*, December 1988.

[3] B. A. Blake and K. Schwan, "Experimental Evaluation of a Real-Time Scheduler for a Multiprocessor System," *IEEE Transactions on Software Engineering*, 17(1), January 1991.

[4] A. Burns, "Scheduling Hard Real-Time Systems: A Review," *Software Engineering Journal*, May 1991.

[5] G. Buttazzo, M. Spuri, and F. Sensini, "Value vs. Deadline Scheduling in Overload Conditions," *Proceedings of the Real-Time Systems Symposium*, December 1995.

[6] G. Buttazzo and J. Stankovic, "Adding Robustness in Dynamic Preemptive Scheduling," in *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*, Edited by D.S. Fussell and M. Malek, Kluwer Academic Publishers, Boston, 1995.

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

[8] H. Chetto and M. Chetto. "Some Results of the Earliest Deadline Scheduling Algorithm," *IEEE Transactions on Software Engineering*, 15(10), October 1989.

[9] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic Scheduling of Real-Time Tasks under Precedence Constraints," *Real-Time Systems Journal*, 2(3), September 1990.

[10] E. G. Coffman, Jr., editor, *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, 1976.

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

[12] M. L. Dertouzos and A. K-L. Mok, "Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks," *IEEE Transactions on Software Engineering*, 15(12), December 1989.

[13] M.R. Gareyand D.S. Johnson, "Complexity Results for Multiprocessor Scheduling Under Resource Constraints," *SIAM Journal of Computing* 4, 1975.,

[14] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[15] P. Goli, J. Kurose, and D. Towsley, "Approximate Minimum Laxity Scheduling Algorithms for Real-Time Systems," Technical Report COINS 90-88, University of Massachusetts, Amherst, Department of Computer and Information Science, 1990.

[16] N. Gehani and K. Ramamritham, "Real-Time Concurrent C: A Language for Programming Dynamic Real-Time Systems," *Real-Time Systems Journal*, 3, pp. 377-405, 1991.

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

[18] J. Hong, X.Tan, and D. Towsley, "A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time Systems," *IEEE Transactions on Computers*, C-38(12), December 1989.

[19] K.S. Hong and J.Y-T. Leung, "On-Line Scheduling of Real-Time Tasks," *Proceedings of the Real-Time Systems Symposium*, December 1988.

[20] G. Koren and D. Shasha, "D-over: An Optimal On-Line Scheduling Algorithm for Overloaded Real-Time Systems," *Proceedings of the Real-Time Systems Symposium*, December 1992.

[21] G. Koren and D. Shasha, "Skip-Over: Algorithms and Complexity for Overloaded Systems that Allow Skips," *Proceedings of the Real-Time Systems Symposium*, December 1995.

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

[23] C.D. Locke, Best-Effort Decision Making for Real-Time Scheduling, *Ph.D. Thesis* Carnegie Mellon University, Pittsburgh, PA., May 1985.

[24] G. Manimaran, S. R. Murthy and K. Ramamritham, "A New Algorithm for Dynamic Scheduling of Parallelizable Tasks in Real-Time Multiprocessor Systems," *Real-Time Systems Journal*, Vol. 15, 1998, pp. 39-60.

[25] G. Manimaran, S. R. Murthy and K. Ramamritham, "New Algorithms for Resource Reclaiming from Precedence Constrained Tasks in Multiprocessor Real-Time Systems," *Journal of Parallel and Distributed Computing*, Vol.44, No.2, Aug. 1997, pp.123-132.

[26] R. McNaughton, "Scheduling With Deadlines and Loss Functions," *Management Science* **6**, 1959.

[27] A. K. Mok and M. L. Dertouzos, "Multiprocessor Scheduling in a Hard Real-Time Environment," In *Proceedings of the Seventh Texas Conference on Computing System*, 1978.

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

[29] S. S. Panwar and D. Towsley, "On the Optimality of the Ste Rule for Multiple Server Queues that Serve Customers with Deadlines," Technical Report COINS 88-81, University of Massachusetts, Amherst, Department of Computer and Information Science, July 1988.

[30] S. S. Panwar, D. Towsley, and J. K. Wolf, "Optimal Scheduling Policies for a Class of Queues with Customer Deadlines until the Beginning of Service," *Journal of the Association for Computing Machinery*, 35(4), October 1988.

[31] K. Ramamritham and J. Stankovic, "Dynamic Task Scheduling in Hard Real-Time Distributed Systems," *IEEE Software*, pp. 65-75, July 1984.

[32] K. Ramamritham, J.A. Stankovic and P. Shiah, "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184-94, April 1990.

[33] K. Ramamritham and J. A. Stankovic, "Scheduling Algorithms and Operating Systems Support for Real-Time Systems," *Proceedings of the IEEE*, pp. 55-67, January 1994.

[34] K. Ramamritham, "Dynamic Priority Scheduling," in *Real-Time Systems - Specification, Verification and Analysis*, Edited by Mathai Joseph, Prentice-Hall, 1996.

[35] C. Shen, K. Ramamritham and J.A. Stankovic, "Resource Reclaiming in Multiprocessor Real-Time Systems," *Transactions on Parallel and Distributed Systems*, Vol. 4, No. 4, pp. 382-397, April 1993.

[36] K. Schwan and H. Zhou, "Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads," *IEEE Transactions on Software Engineering*, Vol. 18, No. 8, pp. 736-748, August 1992.

[37] L. Sha, J. Lehoczky, and R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *Proceedings of Real-Time Systems Symposium*, December 1986.

[38] J.A. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Hard Real-Time Operating Systems," *IEEE Software*, 8(3):62-72, May 1991.

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

[40] H. Tokuda, J. Wendorf, and H. Wang, "Implementation of a Time-Driven Scheduler for Real-Time Operating Systems," *Proceedings of the Real-Time Systems Symposium*, December 1987.

[41] W. Zhao and K. Ramamritham, "Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints," *Journal of Systems and Software*, 7:195-205, 1987.

[42] G. Zlokapa, Real-time Systems: Well-timed Scheduling and Scheduling with Precedence Constraints, Ph.D. Thesis, University of Massachusetts, February 1993.