Chapter 6
Application Mapping

Mapping of applications onto available hardware platforms is a key design step. We need to map applications both to processors and to particular execution times. This is feasible with appropriate scheduling techniques. Taking as many scheduling decisions as reasonable at design time enables us to provide timing guarantees. In this chapter, we will present a selected subset of the corresponding static scheduling techniques. They will be classified according to the triplet notation proposed by Pinedo and others. First of all, we will explain classical scheduling algorithms for single processors. We will cover algorithms for aperiodic as well as for periodic task systems, including the well-known *earliest deadline first* (EDF) and *rate monotonic scheduling* (RMS) algorithms. We will briefly explain the use of bin packing algorithms for homogeneous multiprocessor systems. This will be followed by a presentation of selected scheduling algorithms for heterogeneous multiprocessors. We will be presenting algorithms for independent and dependent jobs. For dependent jobs, the focus is on heuristics. Finally, we will be pointing toward issues in using dynamic scheduling.

6.1 Definition of Scheduling Problems

6.1.1 Elaboration on the Design Problem

The mentioned mapping to execution platforms is included in the simplified design flow, as shown in Fig. 6.1.

Selected scheduling algorithms should allow us to use systems with a certain combination of applications. For example, for a mobile phone, we expect being able to make a phone call while the Bluetooth stack is transmitting the audio signals to a headset and while we are looking up information in our "personal information manager" (PIM). At the same time, there may be a concurrent file transfer or even

P. Marwedel, *Embedded System Design*, Embedded Systems,
https://doi.org/10.1007/978-3-030-60910-8_6

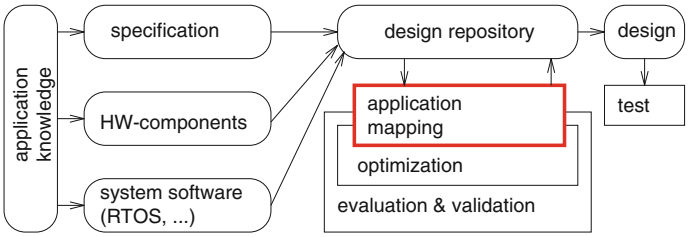


Fig. 6.1 Simplified design information flow

a video connection. We must make sure that these applications can be used together and that we are keeping the deadlines (no lost audio samples!). This is feasible through an analysis of the use cases.

It is a characteristic of embedded and cyber-physical systems that both hardware and software must be considered during their design. Therefore, this type of design is also called **hardware/software codesign**. The overall goal is to find the right combination of hardware and software resulting in the most efficient product meeting the specification. Therefore, embedded systems cannot be designed by a synthesis process taking only the behavioral specification into account. Rather, available components must be accounted for. There are also other reasons for this constraint: in order to cope with the increasing complexity of embedded systems and their stringent time-to-market requirements, reuse is essentially unavoidable. This led to the term **platform-based design:**

"*A platform is a family of architectures satisfying a set of constraints imposed to allow the reuse of hardware and software components. However, a hardware platform is not enough. Quick, reliable, derivative design requires using a platform application programming interface (API) to extend the platform toward application software. In general, a platform is an abstraction layer that covers many possible refinements to a lower level. Platform-based design is a meet-in-the-middle approach: in the top-down design flow, designers map an instance of the upper platform to an instance of the lower, and propagate design constraints*" [476].

The mapping is an iterative process in which performance evaluation tools guide the next assignment.

In this book, we focus on embedded system design based on available execution platforms. This reflects the fact that many modern systems are being built on top of some existing platform. Techniques other than the ones described in this book must be used when the execution platform needs to be designed as well. Due to our focus, the **mapping of applications to execution platforms** can be seen as the **main design problem**. In the general case, mapping will be performed onto multiprocessor systems.

Even for platform-based design, there may be a number of design options. We might be able to select between different variants of a platform, where each variant might have a different number of processors, different speeds of processors, or a

different communication architecture. Moreover, there may be different applicable scheduling policies. Appropriate options must be selected.

This leads us to the following definition of our mapping problem [535]:
Given:

- a set of applications,
- use cases describing how the applications will be used,
- a set of possible candidate architectures:
  - (possibly heterogeneous) processors,
  - (possibly heterogeneous) communication architectures,
  - possible scheduling policies.

Find:

- a mapping of applications to processors,
- appropriate scheduling techniques (if not fixed),
- a target architecture (if not fixed).

Objectives:

- Keeping deadlines and/or maximizing performance,
- minimizing cost, energy consumption, and possibly other objectives.

The exploration of possible architectural options is called **design space exploration** (DSE). As a special case, we may consider a completely fixed platform architecture.

Designing an AUTOSAR-based automotive system can be seen as an example: in AUTOSAR [28], we have a number of homogeneous execution units (called ECUs) and a number of software components. The question is: how do we map these software components to the ECUs such that all real-time constraints are met? We would like to use the minimum number of ECUs.

For embedded systems, we can assume that the set of applications comprises a number of tasks which are released (are ready for execution) repeatedly. The executed code can be associated with tasks. For example, there may be the need to execute certain code once for every input sample. We denote each task by τ_i and sets of tasks by $\tau = \{\tau_1, \ldots, \tau_n\}$.

Definition 6.1 Each execution of a task is called a **job** (cf. Definition 4.4). For each task τ_i, there is an associated set of jobs $J(\tau_i)$. Due to the repeated executions, the set of jobs of task τ_i is possibly not finite.

Definition 6.2 Tasks τ_i which are released once every T_i units of time are called **periodic tasks**, and T_i is called their **period**.

Definition 6.3 A task τ_i is called **sporadic** if there is a lower bound on the length of the interval between successive releases of this task. For each sporadic task τ_i, we call this interval length also T_i.

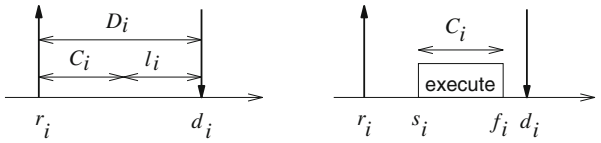


Fig. 6.2 Notation used for jobs

This minimum separation is important: without such a separation, arrival curves for any interval Δ could become unbounded. It would be impossible to find a schedule for a bounded set of resources.

Definition 6.4 Tasks which are neither periodic nor sporadic are called **aperiodic**.

For periodic and sporadic task systems, the concept of hyper-periods simplifies scheduling substantially:

Definition 6.5 Let τ be a periodic or sporadic task system. Its **hyper-period** is defined as the least common multiple of the periods of the individual tasks.

If tasks can be scheduled for one hyper-period, they can be scheduled for all hyper-periods, due to the repeating nature of the task structure.

6.1.2 Types of Scheduling Problems

The following notation is used in the remainder of this chapter for jobs. Let $J = \{J_i\}$ be a set of jobs. Let (see Fig. 6.2):

- r_i be the release time of J_i (the time at which it becomes available for execution),
- C_i be the **worst case execution time** (WCET) of J_i,
- d_i be the **(absolute) deadline** of J_i,
- D_i be the **relative deadline**, that is, the time between a job J_i becoming available and the time until which the same job J_i has to finish execution ($D_i = d_i - r_i$),
- l_i be the **laxity** or **slack**, defined as

$$l_i = D_i - C_i \tag{6.1}$$

  (if $l_i = 0$, then J_i has to be started immediately after it is released),
- s_i be the actual starting time of J_i,
- f_i be the actual finishing time of J_i.

In figures like Fig. 6.2, upward pointing vertical arrows indicate the release of jobs, and downward pointing arrows denote the deadline of jobs.

In the following, we will be using the triplet classification for scheduling problems which was presented by Pinedo [455], based on an notation introduced

earlier by Graham et al. [190]. According to the notation, scheduling problems can be classified by a triplet:

$$(\alpha|\beta|\gamma). \tag{6.2}$$

The α Field

The α field describes the machine environment and consists of a single entry. Simple scheduling algorithms handle the case of single processors, whereas more complex algorithms also handle systems comprising multiple processors. In this book, we consider the following possible values of the α field:

- A value of 1 indicates a single processor.
- A value of Pm indicates m processors which can be used in parallel. Each job can be executed with the same speed on any of the m processors. In this case, processors are said to be **identical** (or homogeneous). The β field can be used to express constraints for the allocation of jobs to processors.
- A value of Qm denotes parallel processors with different performances. The performance is expressed as scaling factors relative to the performance of the slowest processor. Scaling factors can be represented by a vector $(s_1, .., s_m)$, where component s_k is the scaling factor of processor π_k. In this case, processors are called **uniform**. The uniform processor model is very much simplified; we will hardly refer to it.
- A value of Rm indicates m processors with unrelated processing speeds. The execution time of the job or task i on processor k is $C_{i,k}$. Processors are called **heterogeneous.** Heterogeneous processors can be optimized for particular objectives, e.g., for high performance or a small energy consumption. Hence, heterogeneous processors are very important for embedded systems. Hardware accelerators can be modeled as special-purpose processors.

The α field will always contain just a single element.

The β Field

The β field describes processing restrictions. This field may contain several components. In this book, we will consider the following possible values of this field:

- An entry r_i denotes existing release times that are depending on the job i to be allocated.
- An entry $prmp$ indicates that preemptions are allowed. Non-preemptive scheduling is assumed if this entry is missing. Non-preemptive schedulers are based on the assumption that jobs are executed until they are done. As a result, the response

time for external events[1] may be quite long if some jobs have a large execution time. Preemptive schedulers must be used if some jobs have long execution times or if the response time for external events is required to be short. However, preemption can result in unpredictable execution times of the preempted jobs. Therefore, restricting preemptions may be required in order to guarantee meeting the deadline of hard real-time jobs.

- Another possible entry would describe the type of timing constraints. We can distinguish between **soft** and **hard deadlines** (see Definition 1.8 on p. 10).

  Scheduling for soft deadlines is frequently based on extensions to standard operating systems. We will not discuss these systems further in this book. Therefore, the default assumption in this book is to have hard timing constraints.
- Entries *periodic* and *sporadic* may describe the type of task system considered.
- A value of *prec* expresses the fact that precedence constraints exist. Precedences among the jobs require jobs to be executed according to certain partial orders. They may be caused by communication between jobs. For embedded systems, precedences are the rule rather than an exception.
- For sporadic and periodic task sets, we are frequently differentiating scheduling problems with respect to their deadlines:

  The case $D_i = T_i$, for all i, is called the case of **implicit-deadline tasks**, or **Liu-and-Layland (L&L) tasks** [348]. This case is indicated by an entry $D_i = T_i$. Task sets which must satisfy $\forall i : D_i \leq T_i$ are called **constrained-deadline tasks**.

  Tasks whose deadlines do not need to meet any constraints regarding their period are called **arbitrary-deadline tasks**. These cases can also be indicated by corresponding entries.
- We could use this field also to describe the type of scheduling employed. For example, we could use entries *fixed-job-prio* and *fixed-task-prio* for jobs and tasks with a fixed priority.

  Furthermore, we could distinguish between static and dynamic scheduling. Dynamic schedulers take decisions at run-time. They are quite flexible but generate overhead at run-time. Also, they are usually not aware of global contexts such as resource requirements or precedences between jobs. For embedded systems, such global contexts are typically available at design time, and they should be exploited.

  Static schedulers take their decisions at design time. They are based on planning the start times of jobs and generate tables of start times forwarded to a simple dispatcher. The dispatcher does not take any decisions, but is just in charge of starting jobs at the times indicated in the table. The dispatcher can be controlled by a timer, causing the dispatcher to analyze the table.

[1] This is the time from the occurrence of an external event until the completion of the reaction required for the event.

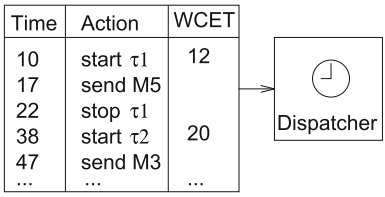


Fig. 6.3 TDL in a time-triggered system

Systems which are totally controlled by a timer are said to be **entirely time-triggered** (TT systems). Such systems are explained in detail in the book by Kopetz [303]:

"*In an entirely time-triggered system, the temporal control structure of all tasks is established* ***a priori*** *by off-line support-tools. This temporal control structure is encoded in a* ***Task-Descriptor List (TDL)*** *that contains the cyclic schedule for all activities of the node*[2] *(Fig. 6.3). This schedule considers the required precedence and mutual exclusion relationships among the tasks such that an explicit coordination of the tasks by the operating system at run time is not necessary. . . . The dispatcher is activated by the synchronized clock tick. It looks at the TDL, and then performs the action that has been planned for this instant . . . .*"

The main advantage of static scheduling is that it can be easily checked if timing constraints are met: *"For satisfying timing constraints in hard real-time systems, predictability of the system behavior is the most important concern; pre-run-time scheduling is often the only practical means of providing predictability in a complex system"* [604]. The main disadvantage is that the response to events may be quite poor.

- Multiprocessor scheduling algorithms either can be executed locally on one processor or can be distributed among a set of processors. Hence, we can also distinguish between **centralized** and **distributed scheduling**. This distinction could also be expressed in the β field.

The γ Field

The γ field describes the objective function. In this book, we consider the following possible values of this field:

- An entry of L_{max} means that the maximum lateness is to be minimized.

  Definition 6.6 Maximum lateness is defined as the difference between the completion time and the deadline, maximized over all jobs.

  Maximum lateness is negative if all tasks complete before their deadline.

[2] This term refers to a processor in this case.

- An entry of MS_{max} denotes the case of minimizing the makespan (the time at which the last job finishes).

  Definition 6.7 The **makespan** is defined as[3]

$$MS_{max} = max_i(f_i) \tag{6.3}$$

- In addition to the entries considered by Pinedo, other entries are relevant for embedded systems. For example, we might want to minimize the energy consumption, or we might even consider trade-offs between several objectives.

A huge amount of scheduling algorithms is available, and comprehensive coverage of existing algorithms would be infeasible even if an entire book or course were available. In a standard undergraduate curriculum, there is typically not enough headroom for a dedicated course on scheduling (but this may be different for courses for graduate students). Therefore, we provide only a brief introduction to scheduling in this book. Many scheduling problems are known to be very complex [41, 455]. In many cases, only approximately optimal mappings can be guaranteed. We will provide an overview of scheduling algorithms frequently considered in embedded systems. Table 6.1 comprises an overview of the techniques in this chapter. From left to right, columns refer to the processor model, asynchronous arrival times, preemptiveness, precedences, periodic/sporadic tasks vs. aperiodic jobs, the deadline model (for periodic/sporadic tasks), job- vs. task-based priorities (for periodic/sporadic tasks), global vs. local scheduling (for multiprocessors), the objective, the subsection, and the name of algorithm(s). Algorithms like earliest deadline first are designed for nonperiodic systems but can be applied in periodic/sporadic systems as well. Note that only the last three lines correspond to full support for heterogeneous processors, as can be seen in column one. Uniform processors will be mentioned only as a possible use of the 0/1 multi-knapsack model. If all jobs arrive at the same time (indicated by an entry of "–" for the second column), preemption is useless, and hence, the third column is not marked by an X. Entries for column D_i are relevant only for periodic/sporadic tasks. Regarding the objectives, we observe that lateness is the relevant objective in many cases. However, for periodic/sporadic scheduling, the key question is: is there a schedule which meets the deadlines? Bin packing is designed to minimize the number of processors. For the HEFT and CPOP heuristics, the makespan is the relevant objective. Only the last line corresponds to a minimization of several objectives, in the form either of a single objective at a time or of real multi-objective optimization using Pareto optimality.

Scheduling is similar to performance evaluation in that it cannot be constrained to a single design step. Rather, scheduling algorithms may be required a number of times during the design of such systems. Very rough calculations may already be required while fixing the specification. Later, more detailed predictions of execution

[3] Pinedo denotes the makespan as C_{max}. We prefer to avoid confusion with execution times C_i.

Table 6.1 Scheduling techniques described or mentioned in this chapter

α	β							γ	Section	Algorithm
Proc.	r_i	*prmp*	*prec*	*periodic*[a]	D_i	prio	glob	Objective		
1	–	–	–	–				L_{max}	6.2.1	Earliest due date
1	X	X	–	–				L_{max}	6.2.1	Earliest deadline first
1	X	X	–	–				L_{max}	6.2.1	Least laxity
1	X	–	–	–				L_{max}	6.2.1	(Theorem 6.3)
1	X	X	X	-		Job		L_{max}	6.2.2	Latest deadline first
1	X	–	X	–				L_{max}	6.2.2	Spring OS [508]
1	X	X	–	X	$= T_i$	Task		$\leq D_i$	6.2.3	Rate monotonic
1	X	X	–	X	$\neq T_i$	Task		$\leq D_i$	6.2.3	Deadline monotonic
Pm	–	–	–	–			X	$m = \lvert\pi\rvert$	6.3.1	Bin packing
Pm	–	–	–	–			X	$\sum b_i$	6.3.1	0/1 Multi-knapsack
Pm	X	X	–	X	$= T_i$			$\leq D_i$	6.3.1	First fit decreasing
Pm	X	X	–	X	$= T_i$	Job	X	$\leq D_i$	6.3.2	Pfair
Pm	X	X	–	–		Job	X	$\leq D_i$	6.3.3	G-EDF, fpEDF, EDZL
Pm	X	X	–	X	$= T_i$	Task	X	$\leq D_i$	6.3.4	G-RM, RM-US, RMZL
Pm	X	X	–	X	$\neq T_i$	Task	X	$\leq D_i$	6.3.4	Density-based
Pm	–	–	X	–				MS_{max}	6.4	ASAP, ALAP
Rm[b]	–	–	X	–				MS_{max}	6.4.3	List scheduling
Pm	–	–	X	–				MS_{max}	6.4.4	Integ.Lin.Progr.(ILP)
Rm	–	–	X	–				MS_{max}	6.5.2	HEFT, CPOP
Rm	–	–	X	–				MS_{max}	6.5.3	ILP, e.g., [361]
Rm	X	X	X	–			(X)	various	6.5.4	DOL, HOPES, MAPS, ..

[a]Algorithms for aperiodic task sets can be applied to periodic/sporadic task sets
[b]List scheduling supports heterogeneous processors only in a limited way

times may be required. After compilation, even more detailed knowledge exists about the execution times, and accordingly, more precise schedules can be made. Finally, it may be necessary to decide at run-time which task is to be executed next. In contrast, in time-triggered systems, RTOS scheduling may be limited to simple table look-ups for tasks to be executed.

In practice, it is very important to know whether or not a schedule exists for a given set of tasks and constraints. A set of tasks is said to be **schedulable** under a given set of constraints if a schedule exists for that set of tasks and constraints. For many applications, **schedulability tests** are important. Tests which always return precise results (called exact tests) are NP-hard in many situations [178]. Therefore, sufficient and necessary tests are used instead. For sufficient tests, sufficient conditions for guaranteeing a schedule are checked. There is a (hopefully small) probability of indicating that scheduling cannot be guaranteed even when a schedule exists. Necessary tests are based on checking necessary conditions. They can be used to show that no schedule exists. However, there may be cases in which necessary tests are passed and the schedule still does not exist.

6.2 Scheduling for Uniprocessors

Let us first consider the case of uniprocessor systems. According to the triplet notation, this corresponds to the case (1|..|..). We are using some of the material from the book by Buttazzo [81] for this section. Refer to this book for additional references.

6.2.1 Scheduling for Independent Jobs

Furthermore, we are restricting our discussion initially to the even more special case of independent jobs executed on uniprocessors.

Earliest Due Date (EDD) Algorithm

First of all, we are looking at the situation where all jobs arrive at the same time, and we try to minimize lateness. If all jobs arrive at the same time, preemption is obviously useless. Therefore, according to the triplet notation, we are considering the case (1| $|L_{max}$). A very simple rule for this case was found by Jackson in 1955 [263].

Theorem 6.1 (Jackson's Rule) *Given a set of n independent jobs with deadlines, any algorithm that executes the jobs in order of nondecreasing deadlines is optimal with respect to minimizing the maximum lateness.*

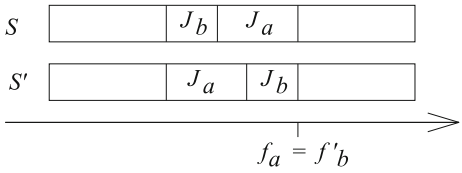


Fig. 6.4 Schedules $\mathcal{S}$ and $\mathcal{S}'$

The algorithm following this rule is called the **earliest due date** (EDD) algorithm. If the deadlines are known in advance, EDD can be implemented as a static scheduling algorithm. EDD requires all jobs to be sorted by their deadlines. Hence, its complexity is $O(n \log(n))$.

Proof of the Optimality of EDD Let $\mathcal{S}$ be a schedule generated by any algorithm A. Suppose A does not lead to the same result as EDD. Then, there are jobs J_a and J_b such that the execution of J_b precedes the execution of J_a in J, even though the deadline of J_a is earlier than that of J_b ($d_a < d_b$). Now, consider a schedule $\mathcal{S}'$. $\mathcal{S}'$ is generated from $\mathcal{S}$ by swapping the execution orders of J_a and J_b (see Fig. 6.4).

In schedule $\mathcal{S}$, the deadline of J_a is earlier than that of J_b, but J_b is executed first. Hence, the maximum lateness among jobs J_a and J_b is that of J_a, or $L_{max}(a, b) = f_a - d_a$.

For schedule $\mathcal{S}'$, $L'max(a, b) = max(L'a, L'b)$ is the maximum lateness among jobs J_a and J_b. $L'a$ is the maximum lateness of job J_a in schedule $\mathcal{S}'$. $L'b$ is defined accordingly. There are two possible cases:

1. $L'a > L'b$: In this case, we have
   $L'max(a, b) = f'a - d_a$
   J_a terminates earlier in the new schedule. Therefore, we have
   $L'max(a, b) = f'a - d_a < f_a - d_a$.
   The right side of this inequality is the maximum lateness in schedule $\mathcal{S}$. Hence, the following holds:
   $L'max(a, b) < L_{max}(a, b)$
2. $L'a \leq L'b$:
   In this case, we have
   $L'max(a, b) = f'b - d_b = f_a - d_b$ (see Fig. 6.4).
   The deadline of J_a is earlier than the one of J_b.
   This leads to
   $L'max(a, b) < f_a - d_a$
   Again, we have
   $L'max(a, b) < L_{max}(a, b)$

As a result, any schedule (which is not an EDD schedule) can be turned into an EDD schedule by a finite number of swaps. Maximum lateness can only decrease during these swaps. Therefore, EDD is optimal for this class of scheduling problems. □

Earliest Deadline First (EDF) Algorithm

Let us consider the case of different release times for uniprocessor systems next. Under this scenario, preemption can potentially reduce maximum lateness. According to the triplet notation, this corresponds to the case $(1|r_i, prmp|L_{max})$.

The earliest deadline first (EDF) algorithm is optimal with respect to minimizing the maximum lateness. It is based on the following theorem [222]:

Theorem 6.2 *Given a set of n independent jobs with arbitrary arrival times, any algorithm that at any instant executes the job with the earliest absolute deadline among all the ready jobs is optimal with respect to minimizing the maximum lateness.*

EDF requires that each time a new ready job arrives, it is inserted into a queue of ready jobs, sorted by their deadlines. Hence, EDF is a dynamic scheduling algorithm. If a newly arrived job is inserted at the head of the queue, the currently executing job is **preempted**. If sorted lists are used for the queue, the complexity of EDF is $O(n^2)$. Bucket arrays could be used for reducing the execution time, but this option is typically not considered.

Example 6.1 Figure 6.5 shows a schedule derived with the EDF algorithm. At time 4, job J_2 has an earlier deadline. Therefore, it preempts J_1. At time 5, job J_3 arrives. Due to its later deadline, it does not preempt J_2. The deadline of J_1 is lather than that of J_3, and hence, it resumes only after J_3 has terminated. Priorities are obviously dynamic: they depend on which deadline is next. Since EDF uses dynamic priorities, it cannot be used with an operating system providing only fixed priorities. However, it has been shown that operating systems can be extended to simulate an EDF policy at the application level [132]. ∇

Proof of Theorem 6.2 Let $\mathcal{S}$ be a schedule generated by some algorithm A, where A is different from EDF. Let $\mathcal{S}_{EDF}$ be a schedule generated by EDF. Now, we partition time into disjoint intervals of length 1.[4] Each interval comprises times within the

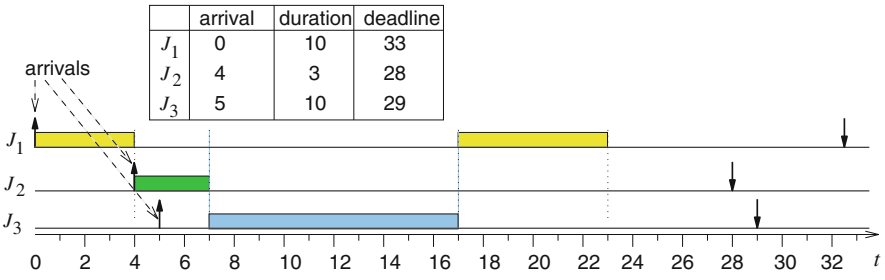


Fig. 6.5 EDF schedule

[4]This proof assumes a discrete time domain. It can be extended to a continuous time domain.

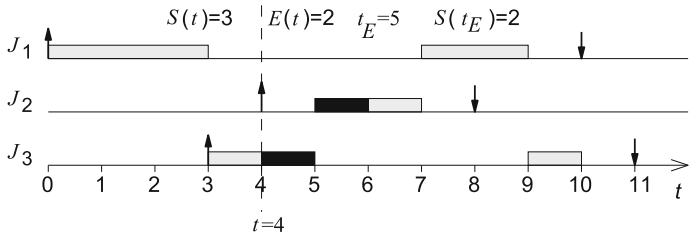


Fig. 6.6 Schedule $\mathcal{S}$

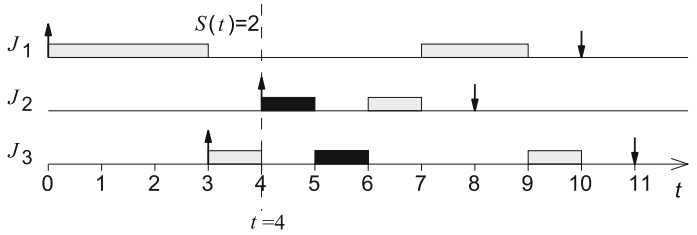


Fig. 6.7 Schedule after swapping jobs $\mathcal{S}(t)$ and $E(t)$

range $[t, t+1)$. Let $\mathcal{S}(t)$ be the job which—according to schedule $\mathcal{S}$—is executed during the interval $[t, t+1)$. Let $E(t)$ be the job which at time t has the earliest deadline among all jobs. Let $t_E(t)$ be the time ($\geq t$) at which job $E(t)$ is starting its execution in schedule $\mathcal{S}$. $\mathcal{S}$ is not an EDF schedule. Therefore, there must be a time t at which we are not executing the job having the earliest deadline. For t, we have $\mathcal{S}(t) \neq E(t)$ (see Fig. 6.6).

Using the same arguments as for Jackson's rule, we can show that swapping $\mathcal{S}(t) \neq E(t)$ like in Fig. 6.7 does not increase maximum lateness. Therefore, by a number of swaps, any non-EDF schedule can be turned into an EDF schedule without increasing maximum lateness. This proves that EDF is optimal among all possible scheduling algorithms.

We can show that swapping will keep all deadlines, provided they were kept in schedule $\mathcal{S}$. According to the initial assumption, the maximum lateness in the schedule $\mathcal{S}$ is 0. Since EDF returns the optimal schedule for minimizing the maximum lateness, the maximum lateness of the EDF schedule is also 0. Hence, for this problem class, the EDF schedule is the optimal schedule to meet the deadlines. □

Least Laxity (LL) Algorithm

Focusing on laxity, we are now considering the case $(1 \mid r_i, prmp, .. \mid ..)$, with the goal of finding a schedule if one exists. Least laxity (LL), least slack time first (LST), and minimum laxity first (MLF) are three names for a laxity-based scheduling strategy

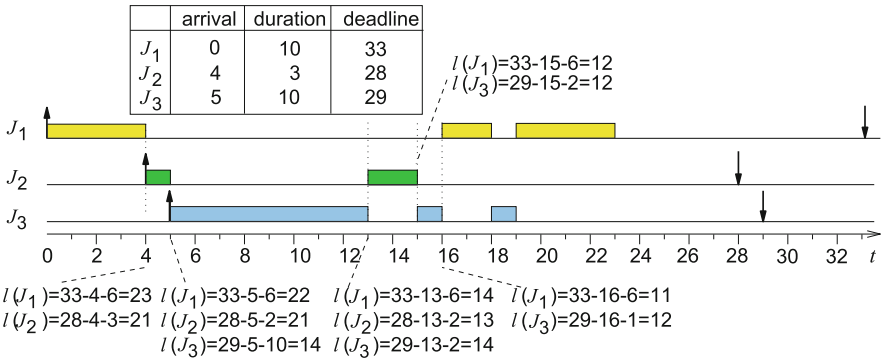


Fig. 6.8 Least laxity schedule

[347]. According to LL scheduling, job priorities are a monotonically decreasing function of the laxity (see Eq. (6.1); the less laxity, the higher the priority). Laxity is dynamically changing and needs to be dynamically recomputed.

Example 6.2 Figure 6.8 shows an LL schedule. Computation of the laxity is included. At time 4, job J_1 is preempted, as before. At time 5, J_2 is now also preempted, due to the lower laxity of job J_3. ∇

LL scheduling is also preemptive. Preemptions are not restricted to times at which new jobs become available. Negative laxities provide an early warning for deadlines to be missed. It can be shown (this is left as an exercise in [347]) that LL is also an optimal scheduling policy for uniprocessor systems with meeting deadlines as the objective. This means that it will find a schedule if one exists. Due to its dynamic priorities, it cannot be used with a standard OS providing only fixed priorities. Furthermore, LL scheduling—in contrast to EDF scheduling—requires the knowledge of the execution time and typically generates many context switches. Its use is therefore restricted to special situations where its properties are attractive. Also, laxity can play a role in multiprocessor scheduling, as will be shown in Sects. 6.3.3 and 6.3.4.

Scheduling Without Preemption

Let us now consider the case of not allowing preemptions, denoted as $(1|r_i|L_{max})$.

Theorem 6.3 *If preemption is not allowed, optimal schedules must leave the processor idle at certain times in order to finish jobs with early deadlines arriving late.*

Proof Let us assume that an optimal non-preemptive scheduler (not having knowledge about the future) never leaves the processor idle. This scheduler must schedule the example of Fig. 6.9 optimally (it must find a schedule if one exists). For the

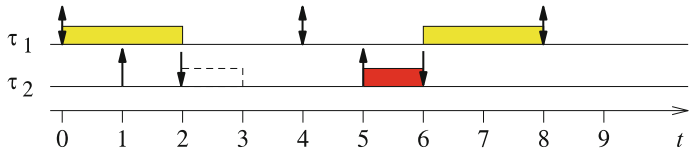


Fig. 6.9 Scheduler needs to leave processor idle

example of Fig. 6.9, we assume we are given two tasks. Let τ_1 be a periodic task with $C_1 = 2$, $T_1 = 4$, $D_1 = 4$, and $r_1 = 0$. Let τ_2 be a sporadic task with $C_2 = 1$, $D_2 = 1$, $T_2 = 4$, and $r_2 = 1$, i.e., sporadically becoming available at times $4 * n + 1$.

Under the above assumptions, our scheduler has to start the execution of task τ_1 at time 0, since it is supposed not to leave any idle time. Since the scheduler is non-preemptive, it cannot start τ_2 when it becomes available at time 1. Hence, τ_2 misses its deadline. If the scheduler had left the processor idle (as shown in Fig. 6.9 at time 4), a legal schedule would have been found. Hence, the scheduler is not optimal. This is a contradiction to the assumptions that optimal schedulers not leaving the processor idle at certain times exist. □

We conclude in order to avoid missed deadlines, the scheduler needs knowledge about the future. Such algorithms are called **clairvoyant**. An algorithm leaving the processor idle in the presence of executable tasks is not **work-conserving**:

Definition 6.8 A scheduling algorithm is **work-conserving** if it does not allow there to be a time at which a processor is idle and there is an executable task [119].

If no knowledge about the arrival times is available a priori, then no online algorithm can decide whether or not to keep the processor idle.

If arrival times are known a priori, the scheduling problem becomes NP-hard in general, and branch and bound techniques are typically used for generating schedules.

6.2.2 Scheduling with Precedence Constraints

Next, let us consider precedence constraints, according to the triplet notation denoted as $(1|\, r_i,\, prmp,\, prec \mid L_{max})$.

Task Graphs

Precedence constraints are expressed by directed acyclic graphs (DAGs, cf. Definition 2.6) $G = (\tau, E)$. The set τ represents the vertices (or nodes) of the DAG and $E \subseteq \tau \times \tau$ its edges.

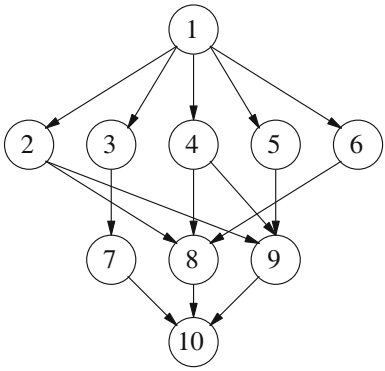


Fig. 6.10 Task DAG

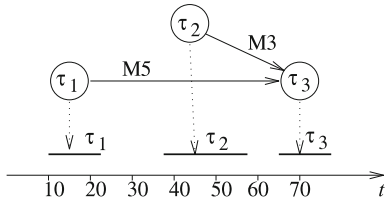


Fig. 6.11 Precedence graph and schedule

Example 6.3 In Fig. 6.10, edges express that source nodes (the first components of the tuples representing edges) must be executed before their sink nodes (the second components of the tuples representing edges). Vertex labels denote task numbers. ∇

There may be several reasons for describing applications as DAGs:

1. On the one hand, each vertex might correspond to an instance of a task, and edges would then represent dependencies between tasks.
2. On the other hand, the availability of multiprocessors leads to the idea of splitting tasks into subtasks and executing these subtasks in an overlapping manner on different processors. Each vertex could then correspond to a subtask. Automatic partitioning of tasks into subtasks such that parallel processors can be efficiently exploited is called **automatic parallelization**. Automatic parallelization is even more difficult than automatic scheduling for a given number of subtasks.

Both cases of creating DAGs can be used in combination: we can have dependencies among tasks, and tasks can be split into subtasks. In the following, we assume that the DAG represents any of the situations just described, and we will call the DAGs **task graphs**. For scheduling, it is not relevant how the DAG was actually generated.

Example 6.4 A legal schedule for a simpler task graph including message transmission is shown in Fig. 6.11. Task τ_3 can be executed only after task τ_1 and τ_2 have completed and sent messages to τ_3. ∇

Latest Deadline First (LDF) Algorithm

An optimal algorithm for minimizing the maximum lateness for the case of simultaneous arrival times of dependent tasks or jobs was presented by Lawler [326]. The algorithm is called **latest deadline first** (LDF). LDF reads the task graph. Among all tasks with no successors, it picks the one with the latest deadline and puts it into a queue. It then repeats this process, always selecting the task with the latest deadline among tasks whose successors have all been selected and inserting it into the queue. At run-time, the tasks are executed in an order **opposite** to the order in which tasks have been entered into the queue. LDF is non-preemptive and is optimal for uniprocessors.

Example 6.5 Consider the case of Fig. 6.11. LDF would first store τ_3 in a queue, since it has no successor. As a result, successors of τ_1 and τ_2 have all been selected already. Which of the two is stored in the queue first depends on their deadline. The node having the later deadline is stored in the queue first. At run-time, the queue is processed in reverse order, starting, for example, with τ_1. ∇

The case of asynchronous arrival times can be handled with a modified EDF algorithm. The key idea is to transform the problem from a given set of dependent jobs into a set of independent jobs with different timing parameters [98]. This algorithm is again optimal for uniprocessor systems.

If preemption is not allowed, the heuristic algorithm developed by Stankovic and Ramamritham [508] can be used.

6.2.3 Periodic Scheduling Without Precedence Constraints

Next, we will consider the periodic case. We will consider mostly tasks instead of jobs, since most properties for periodic systems can be derived for tasks. We will restrict ourselves to a description of the case in which tasks are independent, described as $(1|r_i,prmp,periodic|\ldots)$ in the triplet notation.

Notation

For periodic scheduling, objectives relevant for aperiodic scheduling are less useful. For example, minimization of the total length of the schedule is not an issue if we are talking about an infinite repetition of jobs. The best that we can do is to design an algorithm which will always find a schedule if one exists. This motivates the definition of optimality for periodic schedules.

Definition 6.9 For periodic scheduling, a scheduler is defined to be **optimal** iff it will find a feasible schedule if one exists.

Definition 6.10 For periodic **and sporadic** task systems $\tau = \{\tau_1, .., \tau_n\}$, we define task utilization as

$$u_i = \frac{C_i}{T_i} \tag{6.4}$$

This means that for sporadic task systems, we are using the same definition as for periodic systems, even though T_i just denotes the minimum separation of jobs.

Definition 6.11 For a task system $\tau = \{\tau_1 \dots \tau_n\}$ with utilization u_i of task τ_i, we define the maximum and the total utilization by

$$U_{max} = \max_i (u_i) \tag{6.5}$$

$$U_{sum} = \sum_i u_i \tag{6.6}$$

Rate Monotonic Scheduling

Rate monotonic (RM) scheduling [348] is probably the most well-known scheduling algorithm for independent periodic tasks. Rate monotonic scheduling is based on the following assumptions ("**RM assumptions**"):

1. All tasks that have hard deadlines are periodic.
2. All tasks are independent.
3. $D_i = T_i$, for all tasks.
4. C_i is constant and is known for all tasks. Self-suspension (voluntarily relinquishing the execution) is not allowed.
5. The time required for context switching is negligible.
6. For a single processor and for n tasks, the accumulated utilization U_{sum} does not exceed the following bound:

$$U_{sum} = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \tag{6.7}$$

Figure 6.12 shows the bound of constraint (6.7).
The bound is about 0.7 for large n:

$$\lim_{n \to \infty} n * (2^{1/n} - 1) = log_e(2) = ln(2) \approx 0.7 \tag{6.8}$$

Then, according to the policy for rate monotonic scheduling, **the priority of tasks is a monotonically decreasing function of their period**. In other words, tasks with a short period will get a high priority, and tasks with a long period will

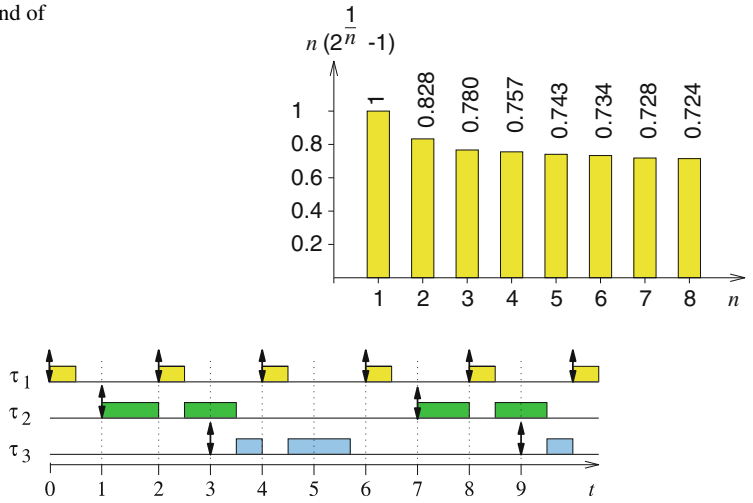


Fig. 6.12 Bound of constraint (6.7)

Fig. 6.13 Example of a schedule generated with RM scheduling ∇

be assigned a low priority. RM scheduling is a **preemptive scheduling policy** with **fixed priorities**.

Example 6.6 Figure 6.13 shows a schedule generated with RM scheduling. Task τ_2 is preempted several times. Double-headed arrows indicate the arrival time of a job as well as the deadline of the previous job. Tasks τ_1 to τ_3 have a period of 2, 6, and 6, respectively. Execution times are 0.5, 2, and 1.75. Task τ_1 has the shortest period and, hence, the highest rate and priority. Each time task τ_1 becomes available, its jobs preempt the currently active task. Task τ_2 has the same period as task τ_3, and neither of them preempts the other.

Constraint (6.7) requires that some of the computing power of the processor is not used in order to make sure that all requests are honored in time. What is the reason for this bound on the utilization? The key reason is that RM scheduling, due to its static priorities, will possibly preempt a task which is close to its deadline in favor of some higher-priority task with a much later deadline. The task having a lower priority can then miss its deadline.

Example 6.7 In Fig. 6.14, task parameters are $T_1 = 5, C_1 = 3, T_2 = 8,$ and $C_2 = 3$. In this case, we have $U_{sum} = \frac{3}{5} + \frac{3}{8} = \frac{39}{40} = 0.975$. This value exceeds the bound: $2 * (2^{\frac{1}{2}} - 1) \approx 0.828$. Not enough idle time is available to guarantee schedulability for RM scheduling. Hence, schedulability is not guaranteed for RM scheduling, and in fact, the deadline is missed at time 8. We assume that the missing computations are not scheduled in the next period. ∇

Such missed deadlines cannot happen if the utilization of the processor is very low, and obviously, they can happen when the utilization is high, as in Fig. 6.14.

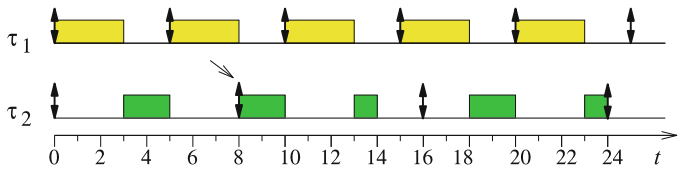


Fig. 6.14 RM schedule does not meet deadline at time 8

If the constraint (6.7) is met, the utilization is guaranteed to be low enough to prevent problems like that of Fig. 6.14. Constraint (6.7) is a **sufficient** condition. This means we might still find a schedule if the condition is not met. Other sufficient conditions exist [54].
RM scheduling has the following important advantages:

- We can show that it is an optimal fixed priority preemptive scheduling algorithm for uniprocessor systems [54].
- It is based on **static** priorities, enabling its application in an operating system with fixed priorities.
- If the above six RM assumptions (see p. 312) are met, all deadlines will be met (see Buttazzo [81]).

RM scheduling is also the basis for a number of formal proofs of schedulability. Designing examples and proofs is facilitated if the most problematic situations for scheduling are known. To get started, we assume the following property:

Property 6.1 We assume that every job completes before the next job of the same task is released.

Definition 6.12 A critical instant for a task τ_i is defined to be an instant t at which a release of that task will have the largest response time.

Theorem 6.4 (Critical Instant Theorem) *For fixed priority scheduling, the response time for execution on a uniprocessor system is maximized for each task τ_i if τ_i is released at the same time as all tasks having a higher priority.*

Proof Here we present the original proof by Liu and Layland [348], using the wording of these authors (except for making the notation consistent with ours): "Let $\tau = \{\tau_1, \ldots, \tau_n\}$ denote a set of priority-ordered tasks with τ_n being the task with the lowest priority. Consider a particular request for τ_n that occurs at t_1. Suppose that between t_1 and $t_1 + T_n$, the time at which the subsequent request of τ_n occurs, requests for task τ_i, $i < n$, occur at t_2, $t_2 + T_i$, $t_2 + 2T_i$, $\ldots$, $t_2 + kT_i$, as illustrated in Fig. 6.15. Clearly, the preemption of τ_n by τ_i will cause a certain amount of delay in the completion of the request for τ_n that occurred at t_1, unless the request for τ_n is completed before t_2. Moreover, from Fig. 6.15 we see immediately that advancing the request time t_2 will not speed up the completion of τ_n. The completion time is either unchanged or delayed by such an advancement. Consequently, the delay in

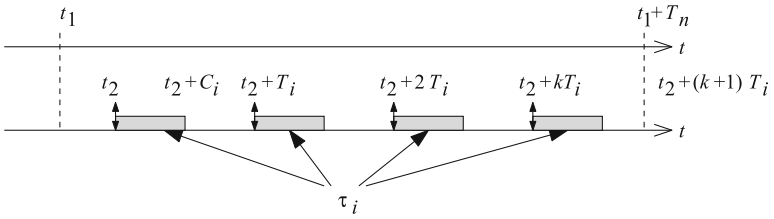


Fig. 6.15 Delaying task τ_n by some τ_i of higher priority

the completion of τ_n is largest when t_2 coincides with t_1. Repeating the argument for all $\tau_i, i = 2, \ldots, m-1$, we prove the theorem." □

Implicitly, we have used Property 6.1 in the proof. If we consider the general case (i.e., the situation in which the assumption of Property 6.1 does not hold; see, for example, Baker [35]), Theorem 6.4 remains valid, but the proof becomes more complex, as shown by Devillers et al. [129] and Bril [69].[5]

The critical instant theorem is of great help when scheduling uniprocessor systems. In general, the critical instant theorem does not hold for multiprocessor systems, which makes proofs much harder. So, the validity of this theorem should really be appreciated!

Let us look at other properties of RM scheduling now. The idle time or *spare capacity* of the processor is not always required.

Theorem 6.5 *Let τ be a system of periodic tasks. If the period of all tasks is a multiple of the period of the task having the next higher priority, τ can be scheduled with RM scheduling if*

$$U_{sum} \leq 1 \tag{6.9}$$

Example 6.8 This requirement is met if tasks in a TV set must be executed at rates of 25, 50, and 100 Hz (or 30, 60, and 120 Hz). ∇

Proof of Theorem 6.5 Let tasks be sorted by priorities, such that $\forall i : T_i \leq T_{i+1}$. Consider some task τ_i and the task with the next lower priority, task τ_{i+1} (see Fig. 6.16). Note that the second deadline of τ_{i+1} matches the fourth deadline of τ_i neatly. Therefore, we can fold the execution times of task τ_{i+1} into the execution times of τ_i and create a new task τ'_{i+1}, containing the execution times of the two original tasks. This folding is feasible if the total execution time of the two tasks does not exceed the period of τ_{i+1}. The process can be repeated in the same way with the next lower-priority task. Overall, folding is feasible as long as the overall utilization does not exceed 1. □

The bounds in Constraints (6.7) and (6.9) allow us to check for schedulability.

[5] I owe this hint to J.J. Chen of TU Dortmund.

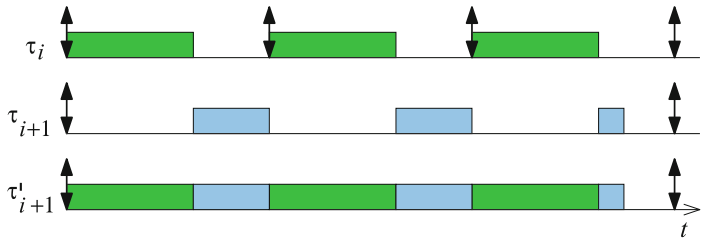


Fig. 6.16 Folding of tasks of adjacent priorities

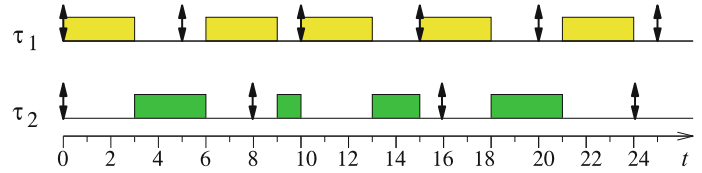


Fig. 6.17 EDF generated schedule for the example of 6.14

Due to the critical instant theorem, the proof of optimality of RM scheduling needs to consider only the case in which tasks are released concurrently with all other tasks of higher priority.

Earliest Deadline First Scheduling

EDF can also be applied to periodic task sets. Obviously, it is sufficient to solve the scheduling problem for a single hyper-period. This schedule can then be repeated for the other hyper-periods. The hyper-period for the example of Fig. 6.14 is 40. It follows from the optimality of EDF for nonperiodic schedules that EDF is also optimal for a single hyper-period and therefore also for the entire scheduling problem. No additional constraints must be met to guarantee optimality. This implies that EDF is optimal also for the case of $U_{sum} = 1$.

Example 6.9 No deadline is missed if the example of Fig. 6.14 is scheduled with EDF (see Fig. 6.17). At time 5, the behavior is different from that of RM scheduling: due to the earlier deadline of τ_2, it is not preempted. ∇

Explicit-Deadline Tasks

Now we move toward the consideration of tasks whose deadline is not the same as the period. Such tasks are called **explicit-deadline tasks**. Each task τ_i in such a system is characterized a triple (C_i, D_i, T_i), where D_i is the relative deadline. The case $D_i \leq T_i$ is called the **constrained-deadline** case. The **arbitrary-deadline**

case is characterized by the absence of such a constraint. Obviously, the class of explicit-deadline tasks is more general than the class of implicit-deadline tasks, and each implicit-deadline task is also an explicit-deadline task.

Utilization is of limited value for the characterization of computational demands of explicit-deadline tasks. To some extent, density plays the role which utilization played to far. Density is defined as

$$dens_i = \frac{C_i}{\min(D_i, T_i)} \tag{6.10}$$

$$dens_{sum}(\tau) = \sum_{\tau_i \in \tau} dens_i \tag{6.11}$$

$$dens_{max}(\tau) = \max_{\tau_i \in \tau}(dens_i) \tag{6.12}$$

Density values characterize computational requirements. A tighter bound is provided by the so-called demand bound function (DBF):

Definition 6.13 *For any sporadic task τ_i and any real number $t \geq 0$, the **demand bound function** $DBF(\tau_i, t)$ is the largest cumulative execution requirement of all jobs that can be generated by τ_i to have both their release times and their deadlines within a contiguous interval of length t.*

The overall execution requirements of task τ_i over an interval $[t_0, t_0 + t)$ are maximized if one of its jobs arrives at the start of the interval—i.e., at time instant t_0—and its subsequent jobs arrive as rapidly as permitted, i.e., at instants $t_0 + T_i, t_0 + 2T_i, t_0 + 3T_i, \ldots$. This observation leads to Eq. (6.13) [39, 41]:

$$DBF(\tau_i, t) = \max\left(0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) * C_i\right) \tag{6.13}$$

Density and the demand bound function are related:

Lemma 6.1 *For all tasks τ_i and for all $t \geq 0$:*

$$t * dens_i \geq DBF(\tau_i, t) \tag{6.14}$$

Proof Let us compare the graphs depicting density and DBF as a function of time. Figure 6.18 shows both functions. The left hand side of Eq. (6.14) is visualized as the straight line with slope $dens_i$. DBF is a step function with steps of height C_i. Whenever a task must be executed, the step function increases by C_i. The first step is at $t = D_i$. By definition of the density, this step does not exceed the straight line. The next steps will be at $t = D_i + T_i, t = D_i + 2T_i, t = D_i + 3T_i$, and so on, since these are the intervals of time after which the demand increases by C_i. Again, these steps will not exceed the straight line. □

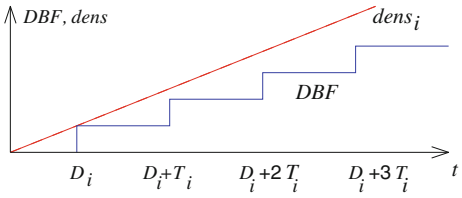


Fig. 6.18 Comparison of density and DBF

EDF can be easily extended to handle the case when deadlines are different from the periods. For RM scheduling, the extension is called deadline monotonic scheduling.

Deadline Monotonic Scheduling

Explicit-deadline tasks can be dealt with in **deadline monotonic (DM) scheduling**. For DM scheduling, static task priorities are based on nonincreasing deadlines: for any two tasks τ_i and $\tau_{i'}$, the priority of τ_i will be higher than that of $\tau_{i'}$ if $D_i < D_{i'}$.

For constrained-deadline tasks, constraint (6.7) can be generalized into constraint (6.15) which is sufficient, but not necessary [81]:

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{1/n} - 1) \tag{6.15}$$

6.2.4 Periodic Scheduling with Precedence Constraints

Scheduling dependent tasks is more difficult than scheduling independent tasks, in particular in the non-preemptive case ($(1|\, r_i, prec, periodic \mid L_{max})$ in the triplet notation). The problem of deciding whether or not a non-preemptive schedule exists for a given set of dependent tasks and a given deadline is NP-complete [178]. In order to reduce the scheduling effort, different strategies are used:

- adding additional resources such that scheduling becomes easier,
- partitioning of scheduling into static and dynamic parts. With this approach, as many decisions as possible are taken at design time, and only a minimum of decisions is left for run-time.

6.2.5 Sporadic Events

In the case of sporadic events, we could connect sporadic events to interrupts and execute them immediately if their interrupt priority is the highest in the system.

However, quite unpredictable timing behavior would result for all the other tasks. Therefore, special **sporadic task servers** are used which execute at regular intervals and check for ready sporadic tasks. This way, sporadic tasks are essentially turned into periodic tasks, thereby improving the predictability of the whole system.

6.3 Scheduling for Independent Jobs on Identical Multiprocessors

Next, we are going to consider multiprocessors, due to their widespread use in the form of multi-cores in contemporary embedded systems. A large number of issues have to be considered during the transition from uniprocessors to multiprocessors. Initially, we assume having m identical processors (or "cores"). Furthermore, we assume dealing with a task system $\tau = \{\tau_1, \ldots, \tau_n\}$ where each task i is characterized by its worst case execution time (WCET) C_i and—in case of periodic or sporadic tasks—its period T_i which is considered to also define the deadline unless otherwise noted. Whenever the periodic or sporadic nature of tasks is not relevant, we may also consider a set of jobs with explicit deadlines d_i instead.

For multiprocessor s, it is not sufficient to decide when to execute tasks or their jobs. Rather, we must decide **when** to execute jobs and **where** to execute them. Thus, a one-dimensional problem becomes a two-dimensional problem.

For m identical processors, obvious necessary conditions for schedulability are

$$\forall i : u_i \leq 1 \tag{6.16}$$

$$U_{sum} \leq m \tag{6.17}$$

6.3.1 Partitioned Scheduling

Our presentation in the next sections is based predominantly on a book written by Baruah et al. [41] and complemented by material from other sources like a survey paper by Davis et al. [119] and slides by I. Puaut [461, 462]. Baruah et al. focus on sporadic task systems. This is partly motivated by the fact that for such systems—in contrast to periodic task systems—no global time synchronization is required for releasing jobs. Rather, it is sufficient to maintain a time base which ensures that the minimum intervals T_i are kept. Also, sporadic task systems are considered for complexity reasons. We start by considering sporadic implicit-deadline tasks on identical multiprocessors. In the triplet notation, this corresponds to the case $(Pm \mid D_i = T_i, sporadic \mid \ldots)$.

Furthermore, we are initially restricting ourselves to the case of partitioned scheduling. This means that each task is allocated to a particular processor. Task migration is not allowed. Partitioned scheduling for synchronous arrival times can

be done by bin packing [306], defined in a notation adjusted for real-time scheduling as follows:

Definition 6.14 Let $\tau = \{1, \ldots, n\}$ be a set of items, where each item $i \in \tau$ has a size $c_i \in (0, 1]$. Let $\pi = \{1, \ldots m\}$ be a set of bins with capacity one. The problem of finding an assignment $a : \tau \to \pi$ such that the number of nonempty bins $m \leq n$ is minimal and such that allocated sizes do not exceed the bin capacity is called the **bin packing problem**.

Bin packing is known to be NP-hard [178]. Hence, optimal algorithms such as the one proposed by Korf [305] need large run-times. Formalization of the scheduling problem as a bin packing problem aims at the minimization of the number of processors m.

For a given number m of processors, it is more appropriate to model scheduling for synchronous arrival times as a knapsack problem, more precisely as a 0/1 multiple knapsack problem. This problem can be defined as follows, again using a notation adjusted for real-time scheduling:

Definition 6.15 (Martello [367]) Let $\tau = \{1, \ldots, n\}$ be a set of n items, each with a size c_i and a benefit b_i. Let π be a set of m knapsacks, each with a capacity κ_k, with $(m \leq n)$. Suppose that we can partially allocate a subset of items to knapsacks $(a : \tau \to \pi)$ such that size constraints are respected:

$$\forall k : \sum_{i,a:i \to k} c_i \leq \kappa_k. \tag{6.18}$$

The problem of selecting disjoint subsets of items so that the total profit $\sum_i b_i$ for items in knapsacks is maximized is called **the 0/1 multiple knapsack problem (MKP)**.

Given an algorithm for the 0/1 multiple knapsack problem, we can allocate jobs to m processors. For identical processors, capacities would all be equal. For uniform processors, we can use capacities to take processor speeds into account. The 0/1 multiple knapsack problem is NP-hard as well. Note that we would possibly not schedule all tasks.

Due to the complexity of scheduling for synchronous arrival times, there is no hope for efficient optimal algorithms for the general problem, and in practice, heuristics are used. Common heuristics are considering tasks and processors in a certain sequence. Heuristics differ by the sequence they use. Lopez et al. [355] have compared several heuristics. They restrict themselves to the so-called **reasonable** allocation algorithms, defined as follows:

Definition 6.16 A **reasonable allocation (RA) algorithm** is defined as one that fails to allocate a task to a multiprocessor platform only when the task does not fit into any processor upon the platform.

Definition 6.17 A **reasonable allocation decreasing (RAD) algorithm** is defined as an RA algorithm considering tasks in a nonincreasing order of utilization.

The algorithms studied by Lopez et al. are obtained by combining all possible combinations of two characteristics:

1. The order in which tasks are considered: tasks can be considered in decreasing order of utilization (denoted by **D**), in increasing order of utilization (denoted by **I**), and in arbitrary order (denoted by an empty character).
2. The search strategy for processor allocation: we consider processors to be ordered in some way. Then, the **first fit** strategy (**FF**) will allocate the first processor on which it fits. The worst fit strategy (**WF**) will allocate the processor with the largest remaining capacity. The best fit strategy (**BF**) will allocate the processor with the minimum remaining capacity on which it fits.

There are a total of nine combinations. All combinations can be implemented efficiently. For example, algorithm **FFD** can be detailed as follows:

```
Sort task set according to nonincreasing utilizations u_i = C_i / T_i;
/* Assume task set is renumbered according to the sorting;*/
for (mt=0; mt ≤ m; mt++) K[mt] =1;            /* initialize capacity */
for (i=1; i≤n; i++) {                               /* for each task */
   for (mt=1; (u_i >K[mt]) and (mt≤m); mt++);  /* sufficient capacity? */
   if (mt > m) mt=0;                      /* no solution, use index 0 */
   a[i]=mt;                   /* return processor allocation in array */
   K[mt]=K[mt]-u_i;                       /* update remaining capacity */
}
```

The heuristic algorithm is certainly not optimal. There may be the question: how far are we off the optimum? Many publications discuss upper bounds on the number of additional processors needed, if compared to the minimum number of processors needed for optimal bin packing. The paper by Dosa [136] is an example of this. For real-time systems, a different question is relevant: is there, for a given number of processors, any bound on the overall utilization up to which schedulability is guaranteed? One utilization bound was proved by Lopez et al. [355]:

Theorem 6.6 *Any reasonable allocation algorithm has a utilization bound no smaller than*

$$U_{B1}(U_{max}) = m - (m-1)U_{max} \tag{6.19}$$

Proof When a task with utilization u_i cannot be allocated, every processor must have tasks allocated to it with a per processor utilization exceeding $(1 - u_i)$. The overall utilization over all allocated tasks and including τ_i must then exceed:

$$m(1-u_i) + u_i = m - (m-1)u_i \tag{6.20}$$

$$\geq m - (m-1)U_{max} \tag{6.21}$$

This condition must be met for allocation not to be feasible. □

Furthermore, define β as

$$\beta = \left\lfloor \frac{1}{U_{max}} \right\rfloor \tag{6.22}$$

β is a lower bound on the number of tasks of our task set which we can run on a single processor. Let us assume that EDF is used for local scheduling on each processor. Lopez et al. also showed the following theorem:

Theorem 6.7 *No allocation algorithm can have a utilization bound larger than*

$$U_{B2}(\beta) = \frac{\beta m + 1}{\beta + 1} \tag{6.23}$$

Proof See Lopez et al. [355]. □

Lopez et al. also proved that **WF** and **WFI** have Eq. (6.19) as their lower bound; the remaining algorithms have Eq. (6.23) as their lower bound. Whenever U_{max} approaches 1, the bound in Eq. (6.19) also approaches 1:

$$U_{B1}(1) = 1 \tag{6.24}$$

When U_{max} gets close to 1, β becomes 1, and U_{B2} becomes

$$U_{B2}(1) = \frac{m + 1}{2} \tag{6.25}$$

The bound in Eq. (6.25) allows us to use multiple processors in a much more efficient way compared to the bound in Eq. (6.24). Hence, with respect to these bounds, **WF** and **WFI** are inferior to the other seven algorithms. Experimentally, it has been shown that **FFD** seems to be superior to **FF** or **FFI** and **BFD** seems to be superior to **BF** and **BFI** [41]. There is also some theoretical evidence which supports this observation [41].

The sketched nine algorithms are relatively simple algorithms. We refrain from presenting more elaborate algorithms for the same problem since the problem considered is too much simplified to apply to realistic applications:

- The scheduling problem, as it has been addressed in this section, is a very much restricted one. There are no precedences, no preemption, and only identical processors.
- Partitioned scheduling may lead to unused processor resources even in situations where jobs are available. This means that partitioned scheduling is not work-conserving. Therefore, optimality is not guaranteed.

Hence, the information in this section provides fundamental knowledge, but practical applications require more sophisticated approaches, like the ones to be presented in the following sections.

6.3.2 Global Dynamic-Priority Scheduling

Having unused processors in the presence of available jobs can be avoided with global scheduling. For global scheduling, the allocation of processors to tasks or jobs is dynamic. This gives us more flexibility, especially in the presence of changing workloads or processor availabilities. In the absence of execution constraints, upper bounds on the utilization like the ones in Constraints (6.19) and (6.23) are replaced by

$$U_{sum} \leq m \tag{6.26}$$

However, this better utilization bound and flexibility comes at the price of a certain overhead for scheduling decisions, preemptions, and job migrations.

Proportional Fair (Pfair) Scheduling

The key idea of proportional fair (pfair) scheduling [40] is to execute each task at a rate corresponding to its utilization.[6] For example, if $u_i = 0.5$ for a set of tasks, then each task should be executed approximately half of the time, regardless of the number of processors. For pfair scheduling, we assume that time is quantized and enumerated with integers. Also, C_i and T_i parameters are represented by integers.

Definition 6.18 The **lag** of a task τ_i at time t with respect to schedule $\mathcal{S}$, denoted as $lag(\mathcal{S}, \tau_i, t)$, is the difference between the number of slots that a task has received and the number of slots that it should have received:

$$lag(\mathcal{S}, \tau_i, t) = u_i * t - \sum_{u=0}^{t-1} alloc(\mathcal{S}, \tau_i, u) \tag{6.27}$$

The first term is the target execution time of task τ_i; the second is the time during which this task has been executed in schedule $\mathcal{S}$. A schedule is said to be a pfair schedule if the lag remains in the interval $(-1, +1)$.

[6] The presentation of pfair scheduling is based on slides by I. Puaut [462].

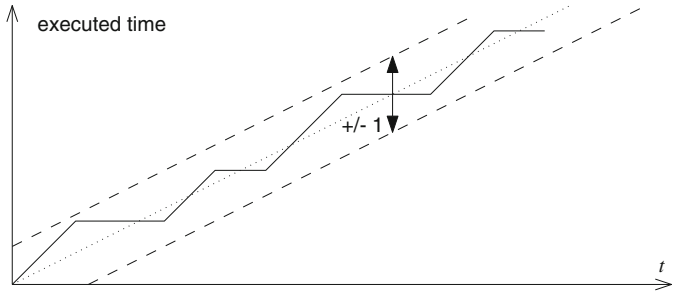


Fig. 6.19 Execution time as a function of real time ∇

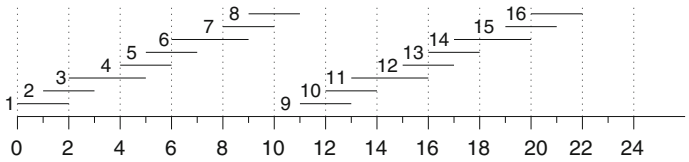


Fig. 6.20 Intervals for allocated execution time

Example 6.10 Figure 6.19 shows the function of actually executed time as a function of real time. The amount of executed time should not reach the two dashed lines.

For pfair scheduling, we divide each task τ_i into subtasks τ_i^j, where j enumerates the execution intervals. For each subtask, we define a pseudo-release time and a pseudo-deadline:

$$r(\tau_i^j) = \left\lfloor \frac{j-1}{u_i} \right\rfloor \tag{6.28}$$

$$d(\tau_i^j) = \left\lceil \frac{j}{u_i} \right\rceil \tag{6.29}$$

Example 6.11 Consider a task τ_i with $C_i = 8$ and $T_i = 11$. Possible intervals for the number of allocated execution slots for each j are shown in Fig. 6.20.
For example:

$$r(\tau_i^6) = \left\lfloor \frac{6-1}{8/11} \right\rfloor = \left\lfloor \frac{55}{8} \right\rfloor = 6$$

$$d(\tau_i^6) = \left\lceil \frac{6}{8/11} \right\rceil = \left\lceil \frac{66}{8} \right\rceil = 9$$

Hence, the sixth subtask of task τ_i must be executed in time interval (6:9). ∇

One particular approach for allocation of a correct number of execution slots is presented in the book by Baruah et al. [41]. In general, there are variations of this scheme: we can apply EDF to pseudo-deadlines, or we can modify EDF by defining rules which are applied in case of ties. It is feasible to obtain schedulability for full processor utilization, i.e., for $U_{sum} \leq m$.

Pfair scheduling potentially suffers from a large number of migrations between processors. Also, due to the integer (over-)approximation of execution times, it is not work-conserving. Variants have been proposed which reduce the overhead for job migrations. Also, the overall complexity can be reduced with some variants.

Pfair scheduling finds many applications in operating systems, for example, for scheduling virtual machines.

6.3.3 *Global Fixed-Job-Priority Scheduling*

G-EDF Scheduling

We can also try to solve the two-dimensional problem with extensions of uniprocessor scheduling algorithms. For example, we could use global EDF (G-EDF). G-EDF, just like EDF, defines job priorities based on the closeness of the next deadlines. If m processors are available, those m jobs having the highest priorities among all available jobs are executed. Obviously, such **priorities are job-dependent** and not just task-dependent. In a global scheduling strategy, we would like to keep preemptions and task migrations to a minimum. For G-EDF, these numbers depend on how we allocate tasks/jobs to a particular processor [189].

Lemma 6.2 *G-EDF is not optimal.*

Proof The proof is by counterexample, adopted from Cho et al. [102]. Suppose $m = 2$ and $C_1 = 3$, $D_1 = 4$, $C_2 = 2$, $D_2 = 3$, $C_3 = 2$, and $D_3 = 3$. As shown in Fig. 6.21 (left), G-EDF schedules J_2 and J_3 first, due to their earlier deadline. J_1 misses its deadline. However, a schedule is feasible, as shown in Fig. 6.21 (right). □

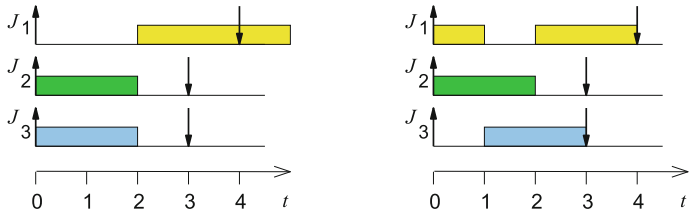


Fig. 6.21 **Left**, G-EDF violates deadline at $t = 4$; **right**, feasible schedule

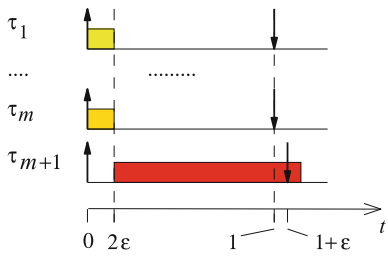


Fig. 6.22 Dhall effect

Obviously, the problem for G-EDF results from not being able to use the second processor for $t > 2$.

In general, G-EDF may suffer anomalies like the so-called Dhall effect [130]: periodic task sets for which one task has a utilization close to one cannot be scheduled with G-EDF.

Example 6.12 To demonstrate the effect, let us consider the case of $n = m + 1$ and

$$\forall i \in [1..m] : T_i = 1, C_i = 2\varepsilon, u_i = 2\varepsilon \tag{6.30}$$

$$T_{m+1} = 1 + \varepsilon, C_{m+1} = 1, u_{m+1} = \frac{1}{1+\varepsilon} \tag{6.31}$$

A corresponding schedule is shown in Fig. 6.22. Initially, only tasks $\tau_1, .., \tau_m$ are executed. The execution of task τ_{m+1} starts only after the first m tasks have completed their execution, and it will miss its deadline. The presence of a single task τ_{m+1} with a high utilization is sufficient to cause a deadline miss at $t = 1 + \varepsilon$. This happens even though the utilization of the other tasks is very small. In fact, the utilization of tasks $\tau_1, ..\tau_m$ can be arbitrarily small, and we will still have a deadline miss. ∇

This motivates using variants of algorithms which assign high priorities to tasks with a high utilization, regardless of their deadline or period.

Algorithm fpEDF is such an algorithm. We assume that we are given an implicit-deadline sporadic task system $\tau = \{\tau_1, \ldots \tau_n\}$ and that tasks are ordered by nonincreasing utilizations u_i. Our goal is to schedule these tasks on m identical processors while avoiding the Dhall effect. Algorithm fpEDF works as follows [41]:

```
for (i=1; i ≤ m − 1; i++){
  if (u_i >0.5) τ_i's jobs obtain highest priority (ties broken arbitrarily)
  else break;
}                    /* Remaining jobs get priorities according to EDF. */
```

This means that the $m - 1$ tasks of largest utilization will obtain the highest priority if their utilization exceeds a value of 0.5.

Theorem 6.8 *Algorithm fpEDF has a utilization bound no smaller than* $\frac{m+1}{2}$.

This is the best bound which we can expect unless some additional information is known, as is evident from the following theorem.

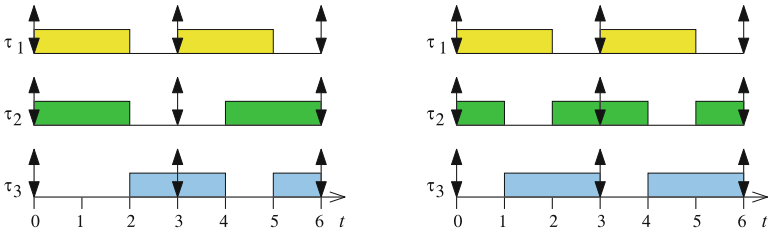


Fig. 6.23 G-EDF: **left**, missed deadlines; **right**, ZL improvement

Theorem 6.9 *No m-processor fixed-job-scheduling algorithm has a schedulable utilization greater than* $\frac{m+1}{2}$.

The proofs of both theorems can be found in [41]. As in the case of partitioned scheduling, stronger bounds are feasible if the largest utilization is known.

A similar idea is used in scheduling algorithm EDF(k): for EDF(k), k tasks of highest utilization obtain the highest priority, breaking ties arbitrarily. All other tasks are scheduled according to EDF.

Theorem 6.10 *EDF(k) will schedule* τ *on m unit-speed (homogeneous) processors, where* τ *is an implicit-deadline sporadic task system.*

$$m = (k-1) + \left\lceil \frac{U(\tau^{(k+1)})}{1-u_k} \right\rceil \tag{6.32}$$

and $U(\tau^{(k+1)})$ *is the utilization for the task set with the first k tasks removed.*

The proof of this theorem can again be found in [41].

EDZL Scheduling

Obviously, G-EDF can miss deadlines for task sets that are schedulable. We can improve G-EDF by adding a consideration of laxity: the EDZL algorithm applies G-EDF as long as the laxity of jobs is greater than zero (see [41, Chapter 20]). However, whenever the laxity of a job becomes zero, the job gets the highest priority among all jobs, even including currently executing jobs.

Example 6.13 Consider the example in Fig. 6.23, adopted from Puaut [461]. In this example, parameters are as follows: $n = 3$, $m = 2$, $T_1 = T_2 = T_3 = 3$, and $C_1 = C_2 = C_3 = 2$. For this example, G-EDF misses the deadlines for τ_3 at times $t = 3n$ for $n = 1, 2, 3..$, as can be seen in Fig. 6.23 (left). However, EDZL keeps the deadlines as can be seen in Fig. 6.23 (right). The detailed behavior depends somewhat on the processor allocation used by EDZL. ∇

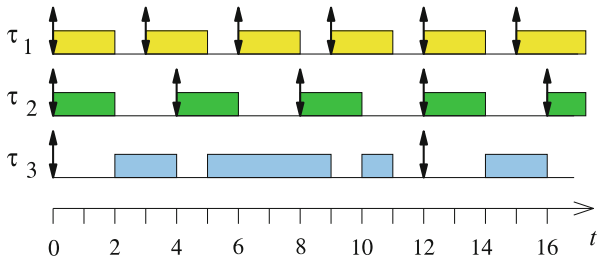


Fig. 6.24 Schedule generated by G-RM

EDZL is strictly superior to EDF, as shown by Choi et al. [101]. Informally, this can be shown as follows:[7] suppose that $\mathcal{S}$ is a schedule from EDF, and $\mathcal{S}'$ is a schedule from EDZL for the same input task set. If a job at time t is scheduled in EDZL but not in EDF, then the job misses the deadline in EDF but not in EDZL. If both schedule the job, then the schedule remains the same. That is, the first moment when $\mathcal{S}$ differs from $\mathcal{S}'$ has the following results:

- either EDZL remains feasible but EDF becomes infeasible
- or both EDZL and EDF are infeasible.

Therefore, EDZL is superior to EDF. Piao et al. [452] proved the following utilization bound for EDZL

$$U_{sum} \leq \frac{m+1}{2} \tag{6.33}$$

6.3.4 Global Fixed-Task-Priority Scheduling

Global Rate Monotonic Scheduling

In a similar way, we can extend rate monotonic scheduling to global rate monotonic scheduling (G-RM). For G-RM, there is an anomaly concerning relaxed schedules:

Lemma 6.3 *For G-RM, there may be situations in which schedules exist for a certain task system, but deadlines are violated if periods are extended.*

Proof We prove the existence of such situations by means of an example, adopted from Puaut [461]. Consider the case $m = 2$, $n = 3$, $T_1 = 3$, $C_1 = 2$, $T_2 = 4$, $C_2 = 2$, $T_3 = 12$, and $C_3 = 7$. Figure 6.24 shows a schedule generated by G-RM. If we extend the period of τ_1 to $T_1 = 4$, τ_3 will miss its deadline (see Fig. 6.25).

[7] I owe this informal explanation to J.J. Chen, TU Dortmund.

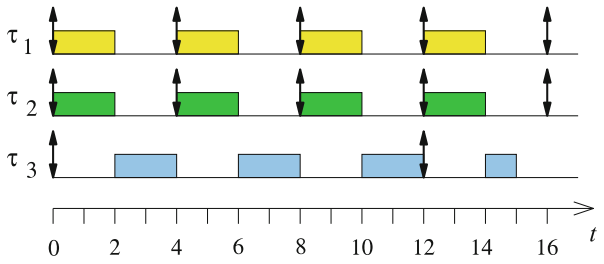


Fig. 6.25 Schedule with a missed deadline at $t = 12$ generated by G-RM

This counterintuitive result makes the design of proofs and examples much more complex, compared to the uniprocessor case. □

The critical instant theorem for uniprocessors (see p. 314) is also not valid for multicore systems.
The following utilization bound has been shown for G-RM:[8]

Theorem 6.11 *Any implicit-deadline periodic or sporadic task system τ satisfying*

$$U_{sum} \leq \frac{m}{2}(1 - U_{max}(\tau)) + U_{max}(\tau) \tag{6.34}$$

is successfully scheduled by G-RM on m unit-speed (homogeneous) processors [50].

G-RM also suffers from the Dhall effect: note that U_{sum} in Eq. (6.34) approaches zero as U_{max} approaches one. Also, like G-EDF, the algorithm cannot fully exploit the presence of multiple processors.

Therefore, algorithm RM-US(ξ) with threshold ξ has been proposed, where US stands for utilization threshold. Given an implicit-deadline sporadic task system $\tau = \{\tau_1, \ldots \tau_n\}$ and tasks ordered by nonincreasing utilizations u_i, the goal is to schedule these up to $(m-1)$ high utilization tasks on $m-1$ identical processors while avoiding the Dhall effect, leaving at least one processor for the remaining tasks. RM-US(ξ) works as follows:

```
for (i=1; i≤ m − 1; i++) {
  if (u_i > ξ) τ_i is assigned highest priority
  else break;
}                    /* remaining tasks are allocated according to G-RM */
```

Theorem 6.12 *Algorithm RM-US(ξ) has a utilization bound no smaller than $\frac{m^2}{(3m-2)}$ upon m unit-speed processors.*

The proof was published by Andersson et al. [16]. For $3m \gg 2$, this bound approaches $\frac{m}{3}$. A tighter bound was shown by Chen et al. [97].

[8] A tighter bound has been shown by Chen et al. [97].

RMZL Scheduling

G-RM might miss deadlines for task sets that are schedulable, and we can consider improvements. One such improvement is RMZL scheduling. For RMZL scheduling, we use (G-)RM scheduling as long as the current laxity is larger than zero. However, when the laxity becomes zero for one of the jobs, we raise its priority to the highest. RMZL scheduling is superior to RM scheduling, since schedules are changed only when RM scheduling could have missed a deadline [41].

Partitioned Scheduling for Explicit Deadlines

Partitioned scheduling for explicit-deadline task systems can be done similar to partitioned scheduling for implicit-deadline task systems by replacing sorting by utilization with sorting by density. However, this approach is not recommended, since density can be unbounded in certain cases. Baruah et al. present a better approach for partitioned scheduling [41].

6.4 Dependent Jobs on Homogeneous Multiprocessors

Results presented in the previous section constitute fundamental basic knowledge, but the restriction to independent tasks and identical processors inhibits their application for many design problems. Next, we will be dropping these restrictions. First of all, we will be dropping the restriction to independent tasks and focus on some simple algorithms used in the design automation community. For example, as-soon-as-possible (ASAP), as-late-as-possible (ALAP), and list (LS) and force-directed scheduling (FDS) are very popular for automated synthesis from algorithmic design descriptions, the so-called high-level synthesis (HLS) (see, for example, Coussy [113]).

6.4.1 As-Soon-as-Possible Scheduling

Considering precedence constraints, as-soon-as-possible (ASAP) scheduling tries to schedule each task as soon as feasible. ASAP scheduling, as used in HLS, considers a mapping of tasks to integer start times: $\mathcal{S} : \tau \rightarrow \mathbb{N}_0$. Allocation to specific processors has to be performed after ASAP scheduling. Preemptions are not allowed.

We assume that the execution times of all tasks are known and that they are independent of the processor executing the tasks. Hence, we are assuming that processors are homogeneous. The algorithm does not consider any constraints on

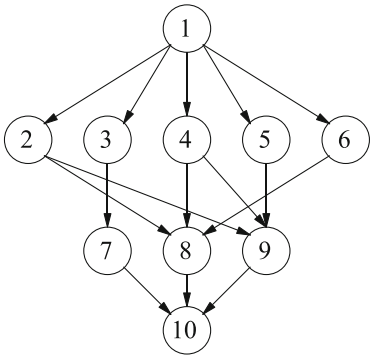
the number of processors and assumes that the number of processors needed for the resulting schedule is available. The ASAP algorithm works as follows:

```
for (t=0; there are unscheduled tasks; t++)   {
   τ′={all tasks for which all predecessors finished};
   set start time of all tasks in τ′ to t;
}
```

Example 6.14 Let us assume that the task graph of Fig. 6.26 (left) is given.

Each node labeled i represents a task τ_i. Furthermore, let us assume that execution times correspond to those listed in Fig. 6.26 (right).

Then, ASAP scheduling will generate the schedule shown in Fig. 6.27. Numbers in blue denote start times; numbers in green denote finish times. Tasks τ_2 to τ_6 all start immediately after task τ_1 has finished, since they do not depend on any other task. Also, tasks τ_7 to τ_9 start as soon as the last of their predecessors has finished, and the same holds for task τ_{10}. The red line in Fig. 6.27 (right) shows that a maximum of five processors is needed, since ASAP scheduling does not consider any constraints on the number of processors. ∇



Task	C_i
1	9
2	13
3	11
4	8
5	10
6	9
7	7
8	5
9	12
10	7

Fig. 6.26 **Left**, task graph; **right**, execution times of tasks

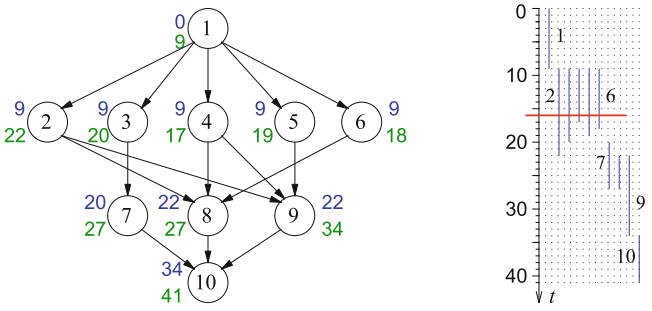


Fig. 6.27 **Left**: ASAP scheduled task graph; **right**: time line

ASAP scheduling minimizes the makespan, since all tasks are scheduled as early as possible. The presented algorithm could be extended to also cover real numbers as execution times. We may consider ASAP scheduling to be of linear complexity, provided that we use a clever technique for computing τ'. The algorithm can also be applied to personal life, corresponding to a situation where each person is eager to perform available work as early as possible.

6.4.2 As-Late-as-Possible Scheduling

As-late-as-possible (ALAP) scheduling is the second simple scheduling algorithm for dependent tasks. For ALAP scheduling, all tasks are started as late as possible. The algorithm works as follows:

```
for (t=0; there are unscheduled tasks; t--) {
     τ'={all tasks on which no unscheduled task depends};
     set start time of all tasks in τ' to (t - their execution time);
}
Shift all times such that the first tasks start at t=0.
```

The algorithm starts with tasks on which no other task depends. These tasks are assumed to finish at time 0. Their start time is then computed from their execution time. The loop then iterates backward over time steps. Whenever we reach a time step, at which a task should finish the latest, its start time is computed, and the task is scheduled. After finishing the loop, all times are shifted toward positive times such that the first task starts at time 0. We could also consider ALAP scheduling as a case of ASAP scheduling starting at the "other" end of the graph.

Example 6.15 For the task graph in Fig. 6.26, ALAP scheduling would generate the result shown in Fig. 6.28. The color coding is the same as for the ASAP example. Note that each task finishes as late as possible. In particular, tasks τ_7 to τ_9 finish only at time 34. Tasks τ_4 to τ_6 finish later than for the ASAP schedule. Tasks τ_1, τ_2, τ_9, and τ_{10} are scheduled as in the ASAP schedule, since these tasks determine the makespan. Tasks which determine the makespan are said to be on the **critical path**. Five processors are needed, as indicated by the red line. ∇

This scheduling strategy can also be applied to personal life. It corresponds to a situation where each person (is lazy and) finishes tasks as late as possible. Many processors are needed if the task graph is very wide at its lower end.[9]

[9]This corresponds to a lot of work in the final phase if people start lazy.

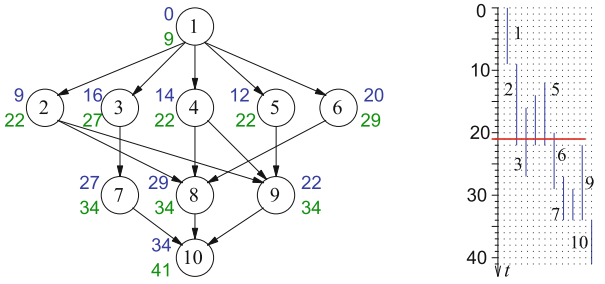


Fig. 6.28 **Left**, ALAP scheduled task graph; **right**, time line

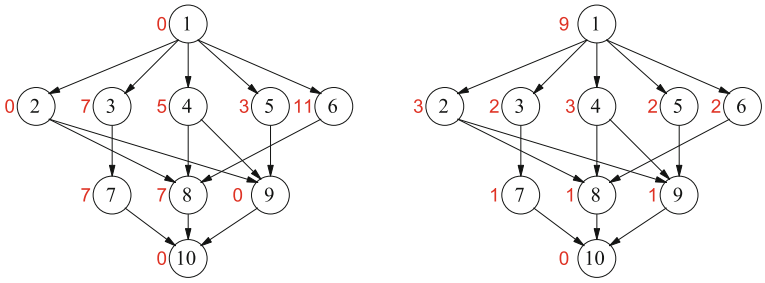


Fig. 6.29 Running example, **left**: mobility; **right**, number of successors

6.4.3 List Scheduling

With list scheduling (LS), we try to maintain the low complexity of ASAP and ALAP scheduling while making the algorithm aware of available processors. Processors may be of different types, but we do still assume that there is a one-to-one mapping between tasks and processor types. Hence, processors may be heterogeneous, but the crucial mapping from tasks to processor types is not generated by list scheduling.

We assume that we have a set L of processor types. List scheduling respects upper bounds B_l on the number of processors for each type $l \in L$.

List scheduling requires the availability of a priority function reflecting the urgency of scheduling a certain task τ_i. The following urgency metrics are in use [528]:

- **Mobility** is defined as the difference between the start times for the ASAP and ALAP schedule. Figure 6.29 (left) shows the mobility for our running example in red. Obviously, scheduling is *urgent* for the four tasks on the critical path for which mobility is zero.
- The **number of nodes below task** τ_i in the tree (see Fig. 6.29 (right)).
- The **path length for a task** τ_i is defined as the length of the path from starting at τ_i to finishing the entire graph G. The path length is typically weighted by

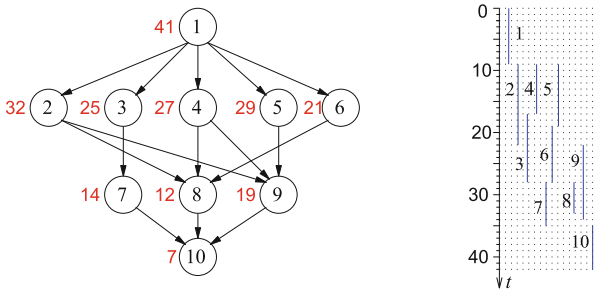


Fig. 6.30 **Left**, task graph with path lengths; **right**, time line for path length based list scheduling

the execution time associated with the nodes, assuming that this information is known. In Fig. 6.30 (left), path lengths have been added.

List scheduling requires the knowledge of the task graph $G = (\tau, E)$ to be scheduled, a mapping from each node of the graph to the corresponding resource type $l \in L$, an upper bound B_l for each l, a priority function (as just explained), and the execution time for each task $\tau_i \in \tau$. List scheduling then fits nodes of maximum priority into each of the time steps such that the constraints are not violated [528]:

```
for (t=0; there are unscheduled tasks; t++)        /* loop over times */
  for (l ∈ L) {                                    /* loop over resource types */
    τ*_{t,l} = set of tasks of type l still executing at time t;
    τ**_{t,l} = set of tasks of type l ready to start execution at time t;
    Compute set τ'_t ⊆ τ**_{t,l} of maximum priority such that
    |τ'_t| + |τ*_{t,l}| ≤ B_l.
    Set start times of all τ_i ∈ τ'_t to t: s_i = t;
  }
```

Example 6.16 Figure 6.30 shows the result of list scheduling as applied to our example in Fig. 6.26, using path length as priority. We assume that all processors are of the same type and that we allow no more than three processors ($B_1 = 3$). At time 9, tasks τ_2, τ_4, and τ_5 have the longest path length and hence the highest priority. τ_4 finishes at time 17, and τ_3 and τ_6 have the longest path length of the remaining tasks. We assume that we schedule τ_3. At time 19, τ_5 finishes and τ_6 can be started. At time 28, τ_3 and τ_6 finish, freeing processors for τ_7 and τ_8. τ_7 finishes at time 35, enabling dependent task τ_{10} to start and to finish at time 42, only slightly later than in the ASAP and ALAP schedules, despite using only three processors. ∇

LS—like ASAP and ALAP scheduling—does not allocate tasks to processors, but there is also no need for doing this for the restricted resource model. LS can also be extended to real numbers as execution times. The algorithm typically generates good results and is easy to adapt to various scenarios. These two features make LS a very popular scheduling algorithm for tasks with precedences.

Force-directed scheduling (FDS) is another heuristic scheduling algorithm for dependent tasks. FDS aims at an efficient use of processors. It tries to balance the number of processors that may be needed at any particular time [449].

6.4.4 Optimal Scheduling with Integer Linear Programming

Next, we will be describing an approach for mapping tasks to multiple processors for which decisions are taken on a more global view of the design problem. It is based on integer linear programming (ILP) (see Appendix A). In this way, constraints and optimization goals are made explicit. We are adopting material from a publication of Coscun et al. [112] in our presentation.

ILP models consist of a linear cost function and a set of linear constraints. We will use the following variables in these two parts of the model:

$x_{i,k}$: = 1 if task τ_i is executed on processor π_k and =0 otherwise

s_i : start time of task τ_i

f_i : finish time of task τ_i

C_i : execution time of task τ_i

$b_{i,j}$: = 1 if task τ_i is executed before τ_j on the same processor, else = 0

Let us assume that our task graph $G = (\tau, E)$ has a common exit node τ_{exit}. If no such node is initially present, we add a virtual node. The finish time of this node is equivalent to the makespan MS_{max}. We can use this finish time as our cost function to be minimized. Hence, the objective of ILP minimization can be expressed as:

$$Min(f_{\tau_{exit}}) \tag{6.35}$$

First, the set of constraints ensures that each task is executed on some processor:

$$\forall \tau_i \in \tau : \sum_{k \in \{1..m\}} x_{i,k} = 1 \tag{6.36}$$

Second, the different times are related by the following equations:

$$\forall \tau_i \in \tau : f_i = s_i + C_i \tag{6.37}$$

Third, in order to respect precedence relations, the following equations can be used:

$$\forall (\tau_i, \tau_j) \in E : s_j - f_i \geq 0 \tag{6.38}$$

Fourth, in a single core, execution is in a sequence as determined by variable $b_{i,j}$:

$$\forall(\tau_i, \tau_j) : f_i \leq s_j \text{ if } b_{i,j} = 1 \tag{6.39}$$

Fifth, each processor can execute only a single task at a time:

$$\forall(\tau_i, \tau_j) : b_{i,j} + b_{j,i} = 1 \text{ if } \exists\, \pi_k : x_{i,k} = x_{j,k} = 1 \tag{6.40}$$

Equations (6.39) and (6.40) can be turned into the linear form required for ILP [112].

The resulting ILP model can be fed into some available ILP solver. ILP models have the advantage of precisely modeling the design problem and the objectives. They enable optimizations from a global viewpoint, using mathematical optimization techniques and stepping away from imperative programming.

The ILP problem is NP-hard. Therefore, run-times of ILP solvers can become large, but there has been significant progress in the design of ILP solvers. Hence, moderately large problems can be solved in acceptable times. However, due to the complexity of ILP, these approaches do not scale to really large designs, and run-times may be unacceptable. Nevertheless, these models can be used for exact optimization of moderately large design problems and serve as a good starting point for heuristics for larger problems.

6.5 Dependent Jobs on Heterogeneous Multiprocessors

6.5.1 Problem Description

Next to dropping the restriction to independent tasks, we would like to drop the restriction to homogeneous processors. We assume that the processing speeds of processors of our execution platform $\pi = \{\pi_1, \ldots, \pi_m\}$ are unrelated. According to Pinedo's triplet notation, we are considering the case $(R_m|r_i, prec, \ldots|\ldots)$, including platforms comprising a mixture of execution units, like FPGAs, GPUs, etc.

The theory of the resulting scheduling problems has not been studied comprehensively. As a result, Baruah et al. [41] state (in Chapter 22): "*although unrelated multiprocessors are becoming increasingly more important in real-time systems implementation, the resulting scheduling theoretic study of such systems is, relatively speaking, still in its infancy*." Some first results are presented in the book by Baruah, but we resort to presenting methods published in the design automation community. They can handle realistic design tasks, sacrificing proofs of optimality.

6.5.2 Static Scheduling with Local Heuristics

We will now describe the heterogeneous-earliest-finish-time (HEFT) and the critical-path-on-a-processor (CPOP) algorithms for static scheduling of tasks in a task graph $G = (\tau, E)$ onto a heterogeneous multiprocessor system $\pi = \{\pi_1, \ldots, \pi_m\}$ [545]. These two algorithms are standard examples of fast algorithms. In a way, they extend ASAP and ALAP scheduling for heterogeneous processors. This is the notation we need:

- We assume that the task graph has a common entry node τ_{entry}. If no such node is initially present, we will add an artificial node having zero execution time and communication bandwidth requirements.
- We assume that the task graph has a common exit node τ_{exit}. If no such node is initially present, we will add an artificial node having zero execution time and communication bandwidth requirements.
- Matrix $C = (c_{i,k})$ denotes the execution time of task τ_i on processor π_k.
- Matrix $B = (b_{k,l})$ denotes the communication bandwidth for communication from processor π_k to processor π_l.
- Matrix $data = (data_{i,j})$ represents the amount of data which must be transmitted from task τ_i to task τ_j.
- Vector $\kappa = (\kappa_k)$ contains the communication startup costs on processor π_k.
- Matrix $H = (h_{i,j,k,l})$ describes the communication cost from task τ_i to task τ_j under the assumption that τ_i is mapped to processor π_k and task τ_j is mapped to processor π_l.[10]

  We will use index i for the source of precedences and index k for its allocated processor. For the sink, we use j and l accordingly.
- For a mapping to processors π_k and π_l, $h_{i,j,k,l}$ represents the communication cost from task τ_i to task τ_j:

$$h_{i,j,k,l} = \kappa_k + \frac{data_{i,j}}{b_{k,l}} \text{ if } k \neq l \tag{6.41}$$

$$= 0 \text{ if } k = l \tag{6.42}$$

- The average communication cost is defined as

$$\overline{h_{i,j}} = \overline{\kappa} + \frac{data_{i,j}}{\overline{B}} \tag{6.43}$$

  where $\overline{\kappa}$ is the average communication startup time and $\overline{B}$ is the average communication bandwidth.
- Given a partial schedule, $s_e(\tau_i, \pi_k)$ is the earliest start time for task τ_i on processor π_k. Obviously, $s_e(\tau_{entry}, \pi_k)$ is zero, for any k.

[10] Indexes k and l are not explicit in the original paper.

- We define $f_e(\tau_i, \pi_k)$ as the earliest finishing time for task τ_i on heterogeneous processor π_k. $f_e(\tau_{entry}, \pi_k)$ is equal to $c_{entry,k}$.
- Once the decision to schedule task τ_i on processor π_k has been taken, the actual start time $s(\tau_i, \pi_k)$ and the actual finish time $f(\tau_i, \pi_k)$ can be computed.

  $s_e(\tau_j, \pi_l)$ and $f_e(\tau_j, \pi_l)$ can be computed from a partial schedule iteratively as follows:

$$s_e(\tau_j, \pi_l) = max\left\{avail(l), max_{\tau_i \in pred(\tau_j)}(f(\tau_i) + h_{i,j,k,l})\right\} \tag{6.44}$$

$$f_e(\tau_j, \pi_l) = c_{j,l} + s_e(\tau_j, \pi_l) \tag{6.45}$$

  where $pred(\tau_j)$ is the set of immediate predecessor tasks of task τ_j, k is the processor task τ_i is mapped to in the partial schedule, and $avail(l)$ is the time that processor π_l completed the execution of its last task. The max expression in the inner term is the time when all data needed by τ_j has arrived at processor π_l.
- For HEFT and CPOP, we assume that the *makespan* is to be minimized. The *makespan* is computed from the actual finish time of the exit node:

$$makespan = f(\tau_{exit}) \tag{6.46}$$

- The average execution time $\overline{c_i}$ is the execution time $c_{i,k}$ averaged over all π_k.
- The **upward rank** $rank_u(\tau_i)$ of a task τ_i is the length of the critical path from the exit node up to and including node τ_i:

$$rank_u(\tau_{exit}) = \overline{c_{exit}} \tag{6.47}$$

$$rank_u(\tau_i) = \overline{c_i} + \max_{\tau_j \in succ(\tau_i)}(\overline{h_{i,j}} + rank_u(\tau_j)) \tag{6.48}$$

  $succ(\tau_i)$ is the set of successors of task τ_i in the task graph.
- The **downward rank** $rank_d(\tau_j)$ of a task τ_j is the length of the critical path from the start node up to and **excluding** node τ_j:

$$rank_d(\tau_{entry}) = 0 \tag{6.49}$$

$$rank_d(\tau_j) = \max_{\tau_i \in pred(\tau_j)}(rank_d(\tau_i) + \overline{c_i} + \overline{h_{i,j}}) \tag{6.50}$$

The HEFT algorithm is shown below:

```
Set the computation and communication costs to mean values;
Compute rank_u(τ_i) ∀τ_i (upward traversal starting at τ_exit);
Sort tasks in nonincreasing order of rank_u values;
while there are unscheduled tasks in the list do {
 select the first task τ_i in the list for scheduling;
  for each processor π_k ∈ π {
   compute f_e(τ_i, π_k) using an insertion based scheduling policy;
  }
  assign task τ_i to processor π_k minimizing f_e(τ_i, π_k);
 }
```

In this context, "insertion-based policy" means that the algorithm searches for a sufficiently large gap among already scheduled tasks such that an allocation into this gap would respect precedence constraints.

Example 6.17 Suppose that execution times are given by the table in Fig. 6.31 (left). Note that for each task, the execution times in Fig. 6.26 (right) have been selected as the minimum time among the three processors. Figure 6.31 (center) shows the schedule obtained by HEFT for the DAG shown in Fig. 6.26 (left). Precedences have been correctly taken into account. We cannot expect to generate the same short schedule as for ASAP or ALAP scheduling as these policies ignore resource constraints. ∇

Task	π_1	π_2	π_3
1	14	16	9
2	13	19	18
3	11	13	19
4	13	8	17
5	12	13	10
6	13	16	9
7	7	15	11
8	5	11	14
9	18	12	20
10	21	7	16

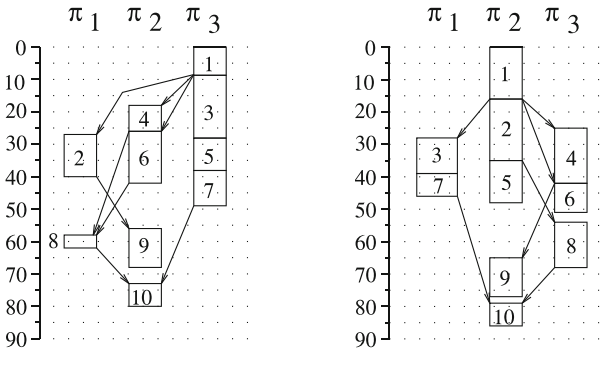


Fig. 6.31 **Left**, execution times; **center**, results for HEFT; **right**, results for CPOP

The CPOP algorithm focuses on the critical path in the DAG and uses different task priorities and different processor allocation strategies. The CPOP algorithm works as follows:

```
Set the computation and communication costs to mean values;
Compute ∀i : rank_u(τ_i) and rank_d(τ_i);
Compute ∀i : priority(τ_i) = rank_d(τ_i) + rank_u(τ_i);
|CP| = priority(τ_entry); /* length of the critical path */
SET_CP = {τ_entry}, where SET_CP is the set of tasks on the critical path;
τ_i = τ_entry;
while τ_i is not the exit task {
  Select τ_j ∈ succ(τ_i), where priority(τ_j) == |CP|.
  SET_CP = SET_CP ∪ {τ_j};
  τ_i = τ_j
};
Select processor π_CP minimizing execution time on the critical path;
Initialize the priority queue with the entry task;
while there is an unscheduled task in the priority queue {
  Select the highest priority task τ_i from the priority queue;
  if τ_i ∈ SET_CP {assign task τ_i on π_CP }
  else{assign task τ_i to the processor which minimizes f_e(τ_i, π_k)};
 Update priority queue with successors of τ_i if they become ready;
}
```

Example 6.18 Figure 6.31 (right) shows the scheduling result for algorithm CPOP. ∇

The HEFT and CPOP algorithms are fast and relatively simple algorithms. Obviously, these algorithms make use of several approximations (e.g., average communication costs) and heuristics. They were selected for this book to demonstrate some key issues of scheduling algorithms for heterogeneous scheduling algorithms. However, it is possible to improve over the results of these two algorithms.

For example, Kim et al. [294] present more complex algorithms generating better results. A mapping for KPNs aiming at makespan minimization has been published by Castrillon et al. [86].

6.5.3 Static Scheduling with Integer Linear Programming

Integer linear programming can also be applied to the case of heterogeneous processors. One approach has been published by Maculan et al. [361]. Most importantly, processor-dependent execution times are taken care of. However, the presented equations require some refinement before they can be fed into an ILP solver and applications have not been included. Also, it is possible to adapt techniques published in the context of high-level synthesis [44, 314].

In most of the publications, optimizations aim at optimizing a single objective. In general, more objectives should be considered. For example, Fard et al. [162] present an algorithm taking four different objectives into account.

6.5.4 *Static Scheduling with Evolutionary Algorithms*

Integer programming based approaches potentially suffer from long execution times. In many cases, the use of evolutionary algorithms allows a better optimization while still keeping execution times reasonably short. We will demonstrate this by means of the distributed operation layer (DOL) tools from ETH Zürich [537]. These tools incorporate

- **Automatic selection of computation templates:** Processor types can be completely heterogeneous. Standard processors, micro-controllers, DSP processors, FPGAs, etc. are all possible options.
- **Automatic selection of communication techniques:** Various interconnection schemes like central buses, hierarchical buses, rings, etc. are feasible.
- **Automatic selection of scheduling and arbitration:** DOL design space exploration tools automatically choose between rate monotonic scheduling, EDF, and TDMA- and priority-based schemes.

The input to DOL consists of a set of tasks together with use cases. The output describes the execution platform, the mapping of tasks to processors together with task schedules. This output is expected to meet constraints (like memory size and timing constraints) and to minimize objectives (like size, energy, etc.). Applications are represented by the so-called problem graphs, which in essence are special task graphs. Figure 6.32 shows a simple DOL problem graph. This graph models computations (see nodes 1, 2, 3, 4) and communication (see nodes 5, 6, 7).

In addition, possible execution platforms are represented by the so-called architecture graphs. Figure 6.33 shows a simple hardware platform together with its architecture graph. Again, communication is modeled explicitly.

The problem graph and the architecture graph are connected in the **specification graph**. Figure 6.34 shows a DOL specification graph. Specification graphs consist of a problem graph and an architecture graph. Edges between the two subgraphs represent feasible implementations. For example, computation 1 can be implemented only on the RISC processor and computation 3 on the RISC processor or on HWM1. Communication 5 can be implemented on the shared bus or locally on the processor if computations 1 and 3 are both mapped to the processor.

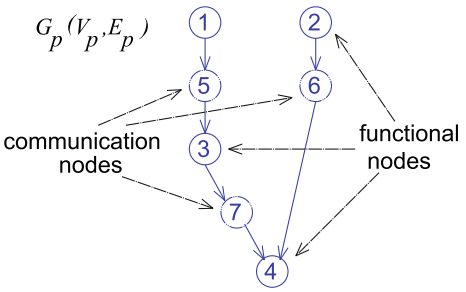


Fig. 6.32 DOL problem graph

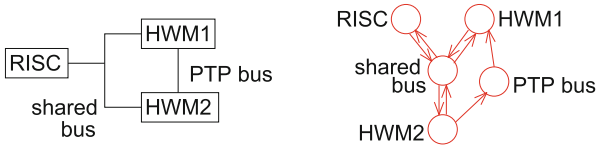


Fig. 6.33 DOL architecture graph

Fig. 6.34 DOL specification graph

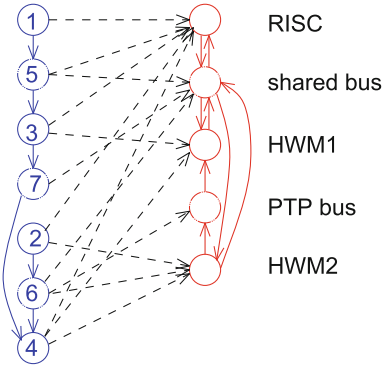
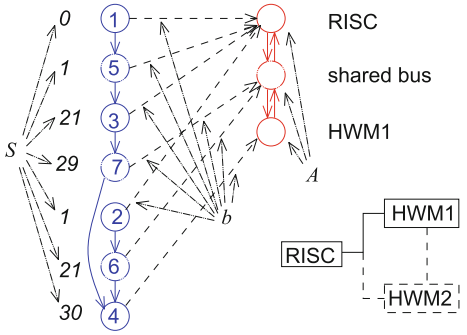


Fig. 6.35 DOL implementation



Implementations are represented by a triple:

- **An allocation** A: A is a subset of the architecture graph, representing hardware components allocated (selected) for a particular design.
- **A binding** b: A selected subset of the edges between specification and architecture identifies a relation between the two. Selected edges are called **bindings**.
- **A schedule** $\mathcal{S}$: $\mathcal{S}$ assigns start times to each node τ_i in the problem graph.

Example 6.19 Figure 6.35 shows how the specification of Fig. 6.34 can be turned into an implementation. HWM2 and the PTP bus are not used and not included in the set A. A subset b of the edges have been selected for mapping. Nodes 1, 2, 3, and 5 have indeed all been mapped to the RISC processor, turning communication 5 into local communication. Node 4 is mapped to HWM1 and communicates via shared bus. Schedule $\mathcal{S}$ specifies that computation 1 starts at time 0, communication 5 and

computation 2 start at time 1, computation 3 and communication 6 start at time 21, communication 7 starts at time 29, and finally computation 4 starts at time 30. ∇

In DOL, implementations are generated with evolutionary algorithms. With such algorithms, solutions are represented as strings in chromosomes of "individuals" [31, 32, 107]. Using evolutionary algorithms, new sets of solutions can be derived from existing sets of solutions. The derivation is based on evolutionary operators such as mutation, selection, and recombination. The selection of new sets of solutions is based on **fitness values**. Evolutionary algorithms are capable of solving complex optimization problems not tractable by other types of algorithms. Finding appropriate ways of encoding solutions in chromosomes is not easy. On the one hand, the decoding should not require too much run-time. On the other hand, we must deal with the situation after the evolutionary transformations. These transformations could generate infeasible solutions, except for some carefully designed encodings.

In DOL, chromosomes encode allocations and bindings. In order to evaluate the fitness of a certain solution, allocations and bindings must be decoded from the individuals (see Fig. 6.36). In DOL, schedules are not encoded in the chromosomes. Rather, they are derived from the allocation and binding. This way, overloading evolutionary algorithms with scheduling decisions is avoided. Once the schedule has been computed, the fitness of solutions can be evaluated.

The overall architecture of DOL is shown in Fig. 6.37.

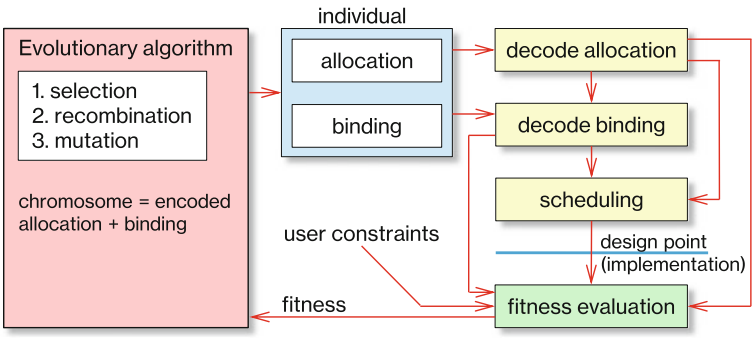


Fig. 6.36 Decoding of solutions from chromosomes of individuals

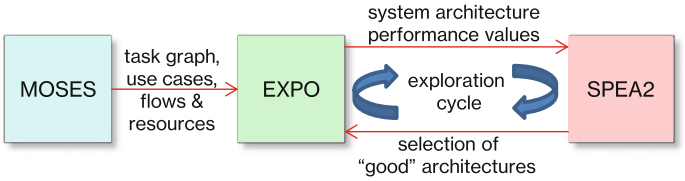


Fig. 6.37 DOL tool

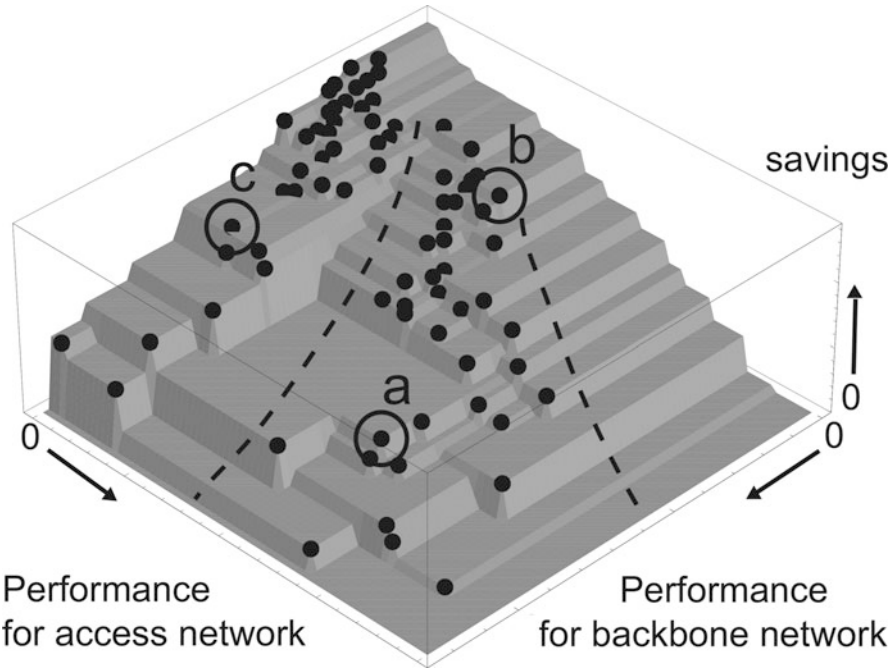


Fig. 6.38 Pareto front of solutions for a design problem, ©ETHZ

Initially, the task graph, use cases, and available resources are defined. This can be done with a specialized editor called MOSES. This initial information is evaluated in the evaluation framework EXPO. Performance values computed by EXPO are then sent to SPEA2, an evolutionary algorithm-based optimization framework. SPEA2 selects good candidate architectures. These are sent back to EXPO for an evaluation. Evaluation results are then communicated again to SPEA2 for another round of evolutionary optimizations. This kind of ping-pong game between EXPO and SPEA2 continues until good solutions have been found. The selection of solutions is based on the principle of Pareto optimality. A set of Pareto optimal designs is returned to the designer, who can then analyze the trade-off between the different objectives.

Example 6.20 Figure 6.38 shows the resulting visualization of the Pareto front. Trade-offs between the performance for two applications and the savings in cost can be seen. ∇

Holzkamp designed a variant of DOL which focuses on memory optimizations [220]. Evolutionary algorithms have become a standard technique for more advanced scheduling problems, beyond the problems solved by HEFT or CPOP.

The functionality of the SystemCodesigner [285] is somewhat similar to that of DOL. However, it differs in the way specifications are described (they can be represented in SystemC) and in the way the optimizations are performed. The

mapping of applications is modeled as an ILP model. A first solution is generated using an ILP optimizer. This solution is then improved by switching to evolutionary algorithms.[11]

Daedalus [422] incorporates automatic parallelization. For this purpose, sequential applications are mapped to Kahn process networks. Design space exploration is then performed using Kahn process networks as an intermediate representation.

Other approaches start from a given task graph and map to a fixed architecture. For example, Ruggiero maps applications to cell processors [475]. The HOPES system is able to map to various processors [195], using models of computation supported by the Ptolemy tools. Some tools take additional objectives into account. For example, Xu considers the optimization of the dependable lifetime of the resulting system [605]. Simunic incorporates thermal analysis into her work and tries to avoid too hot areas on the MPSoC [492]. Further work includes that of Popovici et al. [457]. This work uses several levels of modeling, employing Simulink and SystemC as languages.

Auto-parallelizing approaches for fixed architectures include work at the University of Edinburgh [168]. MAPS tools [88] combine automatic parallelization with a limited DSE. Cordes [110] worked on the automatic parallelization for multicores, using high-level cost models. Neugebauer et al. [417] designed an approach to parallelization and used it for the optimization of an innovative sensor for bioviruses. The combination of sensing and information processing demonstrates the value of cyber-physical systems.

6.5.5 Dynamic and Hybrid Scheduling

For dynamic scheduling, processor allocation is performed at run-time rather than at design time. Dynamic scheduling has a number of advantages [493]:

- **Adaptability to the available resources**: Dynamic scheduling is able to take changing resource availabilities like energy, memory space, and communication bandwidth into account.
- **Ability to enable unforseeable upgrades**: Changing application requirements are easier to integrate when scheduling is dynamic.
- **Resilience to defects**: Defective resources like failed processors can be taken into account by dynamic scheduling.
- **Use of non-real-time platforms**: Dynamic scheduling is the standard for non-real-time computing. Hence, techniques for non-real-time computing can be applied, which helps to reduce development efforts.

However, there are also disadvantages:

- **Lacking real-time guarantees**: In a fully dynamically scheduled system, it is difficult if not impossible to give real-time guarantees.

[11] A more recent version uses a satisfiability (SAT) solver for the same purpose.

- **Run-time overhead**: Dynamic scheduling requires run-time for taking scheduling decisions. Therefore, complex scheduling techniques must be avoided.
- **Limited knowledge**: At run-time, there is typically limited knowledge concerning the task system and its parameters.

There are two approaches for dynamic scheduling: **on-the-fly mapping** and hybrid **mapping using previously analyzed (DSE) results**.

Singh et al. [493] provide an overview of 25 different approaches for on-the-fly mapping. This type of mapping is closest to mapping in non-real-time systems.

Hybrid mapping techniques using previously analyzed (DSE) results try to avoid some of the disadvantages listed above by making results from design time analysis available at run-time. For example, we could pre-compute schedules for likely run-time scenarios and then select at run-time the schedule for the current scenario. Singh et al. distinguish between multiple mappings pre-computed for a single application, multiple mappings pre-computed for a multiple applications, and reliability-aware analysis.[12] The authors provide an overview of 21 different approaches for performing design-time analysis and run-time mapping in a sequence.

One could go one step further by integrating scheduling with the application. For example, Kotthaus [307] has designed an approach to mathematical optimization. In this approach, the number of evaluations of an objective function is not fixed, but depends also on the progress of parallel function evaluations on a multi-core system. Similar integration would also be possible for other applications.

6.6 Problems

We suggest solving the following problems either at home or during a flipped classroom session:

6.1 Suppose that we have a set of four jobs. Release times r_i, deadlines D_i, and execution times C_i are as follows:

- J_1: r_1=10, D_1=18, C_1=4
- J_2: r_2=0, D_2=28, C_2=12
- J_3: r_3=6, D_3=17, C_3=3
- J_4: r_4=3, D_4=13, C_4=6

[12] We merge Singh's hybrid mappings with these three classes.

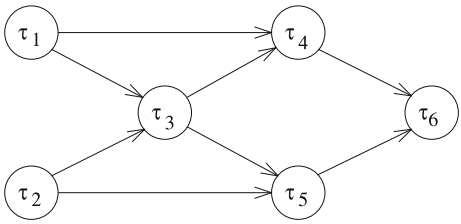


Fig. 6.39 Precedences

Generate a graphical representation of schedules for this job set, using *earliest deadline first* (EDF) and *least laxity* (LL) scheduling algorithms! For LL scheduling, indicate laxities for all jobs at all context switch times. Will any job miss its deadline?

6.2 Suppose that we have a task set of six tasks τ_1 to τ_6. Their execution times and their deadlines are as follows:

- τ_1: D_1=15, C_1=3
- τ_2: D_2=13, C_2=5
- τ_3: D_3=14, C_3=4
- τ_4: D_4=16, C_4=2
- τ_5: D_3=20, C_3=4
- τ_6: D_4=22, C_4=3

Precedences are as shown in Fig. 6.39. Tasks τ_1 and τ_2 are available immediately. Generate a graphical representation of schedules for this task set, using the *latest deadline first* (LDF) algorithm!

6.3 Suppose that we have a system comprising two tasks. Task 1 has a period of 5 and an execution time of 2. The second task has a period of 7 and an execution time of 4. Let the deadlines be equal to the periods. Assume that we are using *rate monotonic scheduling* (RMS). Could any of the two tasks miss its deadline, due to a too high processor utilization? Compute this utilization, and compare it to a bound which would guarantee schedulability! Generate a graphical representation of the resulting schedule! Suppose that tasks will always run to their completion, even if they missed their deadline.

6.4 Consider the same task set as in the previous assignment. Use *earliest deadline first* (EDF) for scheduling. Can any of the tasks miss its deadline? If not, why not? Generate a graphical representation of the resulting schedule! Suppose that tasks will always run to their completion.