Soft Real-Time Scheduling

Jeremy P. Erickson and James H. Anderson

Abstract The notion of temporal correctness applicable to a hard real-time system is quite categorical: such a system is deemed to be temporally correct if and only if no task ever misses a deadline. In contrast, soft real-time systems are sometimes permitted to miss deadlines, and there are a variety of ways in which the term "sometimes" might plausibly be defined. As a result, several different notions of soft real-time correctness have been studied in the literature. In this chapter, a survey of research results pertaining to several such notions is presented. Additionally, the related issue of overload management is considered. Overloads may be common in soft real-time systems because such systems are typically provisioned less pessimistically than hard real-time ones.

1 Introduction

The common characteristic of real-time systems, as discussed throughout this book, is that results must not only be correct, but must be produced "at the right time." The precise definition of "at the right time" depends on the type of system.

A system is typically defined to be a "hard real-time (HRT)" system if each job (i.e., invocation of a program, or "task"), has a deadline by which it must complete in order for the system to be correct. This definition of correctness is needed if drastic consequences could result from a missed deadline. For example, consider a task that adjusts a rudder on an aircraft, in response to the pilot using fly-by-wire controls. This task has a hard real-time requirement, as a missed deadline could result in a

Jeremy P. Erickson
Department of Computer Science The University of North Carolina Campus Box 3175, Brooks Computer Science Building Chapel Hill, N.C. 27599-3175 USA e-mail: jerickso@cs.unc.edu

James H. Anderson
Department of Computer Science The University of North Carolina Campus Box 3175, Brooks Computer Science Building Chapel Hill, N.C. 27599-3175 USA e-mail: anderson@cs.unc.edu

crash. In order to guarantee the correctness of such a system, it is typically necessary to make highly pessimistic assumptions about system behavior, in order to ensure that a deadline cannot be missed under any possible circumstances. This usually requires over-provisioning the system.

By contrast, a system is defined to be a "soft real-time (SRT)" system if it has less stringent requirements. In such a system, each job typically still has a deadline, but the system may be deemed to be correct even if some jobs miss their deadlines. As an example of a SRT deadline constraint, one might require that some fraction of all deadlines in a system be met. Such relaxed deadline constraints are often sufficient. For example, a video decoding system that operates at 50 frames per second must decode each frame within a 20 ms period, or the video may visibly skip. Such a skip is not catastrophic, and the reduced pessimism enabled by tolerating some skips can allow the system to be more fully provisioned.

In this chapter, we consider several different notions of SRT correctness and survey work pertaining to these various notions. We begin in Section 2 by presenting basic definitions that will be used throughout this chapter. Then, in Section 3, we consider various definitions of SRT correctness that are similar to that considered above, where only some fraction of deadlines are required to be met. Next, in Section 4, we consider a definition of SRT correctness that does not require any deadline to be met, but does require bounded *tardiness*, i.e., jobs may miss deadlines as long as the extent of miss is bounded. SRT systems are often provisioned less pessimistically than HRT ones, and as a result, the underlying hardware platform can be expected to sometimes be *overloaded*. In such situations, policies must be employed that ensure that overloads eventually abate by reducing computational demand. One way to reduce demand is by dropping work. In Section 5, we survey approaches for dealing with overload that use *value functions* in deciding which work to drop. Another way to reduce demand is by reducing the rate at which tasks submit work. In Section 6, we survey work on overload management in which such an approach is taken. Finally, in systems that contain tasks of different *criticalities*, criticality-cognizant overload management policies can be employed. We survey work on such policies in Section 7.

2 Basic Definitions

Except where otherwise noted, some variant of the *sporadic task model* is assumed in all papers surveyed in this chapter. In order to describe this model, we depict in Figure 2 an example task running by itself. A task represents one process that is composed of a (potentially infinite) series of discrete jobs. When a new job of a task becomes available for execution, we say that that job is *released* by the task. Because each task is a single process, a newly released job of a task must wait to actually begin execution if any prior jobs of the same task have not yet completed.

We denote a system comprised of n sporadic tasks as $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i is specified by defining several temporal parameters.

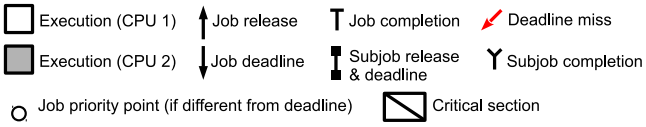


Fig. 1 Common key for many figures in this chapter.

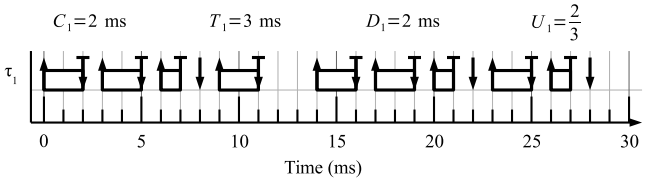


Fig. 2 Example of a sporadic task, with key in Figure 1.

One such parameter is the *worst-case execution time* (*WCET*) of τ_i, denoted C_i. This parameter specifies an upper bound on the execution time for any job of τ_i. In Figure 2, $C_1 = 2$ ms, so no job runs for over 2 ms. However, some jobs run for only 1 ms, as allowed by the model.

Another parameter is the *minimum separation time* of τ_i, denoted T_i. This parameter specifies the minimum amount of time between two successive job releases of τ_i. In Figure 2, $T_1 = 3$ ms, so job releases occur at least three time units apart. However, after the job release that occurs at time 9 ms, no new job is released until time 14 ms, as allowed by the model.

The *absolute deadline* of a *job* is the point in time by which that job should finish. In Figure 2, the absolute deadline of the first job is at time 2 ms. The precise interpretation of "should finish" depends on the particular definition of SRT correctness assumed. The *relative deadline* of a particular *task* τ_i, denoted D_i, is the time between the release time and absolute deadline of each job of that task. In Figure 2, $D_1 = 2$ ms, so the job released at time 9 ms has its absolute deadline at time 11 ms.

We often consider *implicit-deadline* task systems in which for each task τ_i, $D_i = T_i$. However, some work is also applicable to *arbitrary-deadline* task systems that may violate this assumption. For the purpose of examples, we will often use the notation $\tau_i = (C_i, T_i)$ for the tasks of an implicit-deadline task system.

A final parameter of each task τ_i is its *utilization*, denoted U_i. A task's utilization is simply the ratio of its WCET to its minimum separation time: $U_i = \frac{C_i}{T_i}$. The utilization of a task is significant because it indicates the long-term processor share needed by the task, in the worst case.

Some SRT schedulers deal with situations of *overload*. In an overload situation, there is more work than can possibly be completed with reasonable time constraints. For example, two tasks with utilization greater than 0.5 cannot execute together on a uniprocessor, or jobs will miss their deadlines by increasing amounts. SRT schedulers that work in the presence of overload may employ various methods to

reduce computational demand so as to encourage overloads to abate. Several such methods are reviewed later.

3 Meeting Some Deadlines

In this section, we review prior work in which a definition of SRT correctness is employed that requires some deadlines to be met. All of the papers reviewed in this section focus on uniprocessor platforms and the implicit-deadline *periodic* task model, where tasks have exact rather than minimum separation times.

Koren and Shasha (1995a) allowed each task to have a *skip factor s*: each time a job of that task misses a deadline, the next $s-1$ jobs must complete. The scheduler can simply skip any task that would miss a deadline, so some task sets with overload due to total utilization larger than one can be scheduled. However, Koren and Shasha showed that even on a uniprocessor, optimal scheduling with their model is NP-hard. Hamdaoui and Ramanathan (1995) considered the more general (h,k) model.[1] In that model, h jobs of a task must meet their deadlines out of any consecutive k jobs of that task. Both of these types of constraints are generalized as *weakly hard* constraints by Bernat et al. (2001). They defined a "weakly hard real-time system" as any system with a precise bound on the distribution of met and missed deadlines. (Ordinary HRT systems are a special case, where every deadline is met.) Bernat et al. described a few variants, which can be combined with logical operators:

- A task can "meet any h in k deadlines," which is identical to the (h,k) model discussed above.
- A task can "meet row h in k deadlines", meaning that it must meet h deadlines in a row in every window of k deadlines. If $k = h+1$, then this scheme reduces to a skip factor of h.
- A task can "miss any h in k deadlines," meaning that it cannot miss more than h deadlines in a window of k.
- A task can "miss row h in k deadlines," meaning that it cannot miss more than h deadlines in a row in a window of k. (The window size k is not actually required to express this condition.)

A weaker form of the (h,k) model, the *window-constrained* task model, was described by West and Poellabauer (2000). In that model, the time line is segmented into periodic windows, each containing k consecutive jobs of a given task, and within each window, h jobs of that task must meet their deadlines. (Any task system that is schedulable using the (h,k) model is also schedulable using the window-constrained model.)

Lin and Natarajan (1988) proposed the *imprecise computation model* for tasks that compute numerical results. Under that model, each job has a mandatory part that must complete before its deadline under any circumstances, and an optional part

[1] We have changed their notation slightly to avoid conflict with other terms.

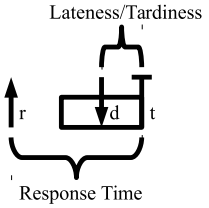


Fig. 3 Response time, lateness, and tardiness. If t were before d, then lateness would be negative, while tardiness would be zero.

that can be interrupted at any time. The mandatory part guarantees an approximate solution, and the precision of the solution must be non-decreasing as the optional part executes. The task must be defined to conform to these requirements. Ideally, every task would run its optional part to completion, but part of that computation can be cancelled when that is not possible. This model is not sufficient to provide a well-defined scheduling problem, because there must be some mechanism to determine which optional parts to execute. Several potential strategies, such as minimizing the number of dropped optional portions or minimizing the maximum error, were discussed by Liu et al. (1991). Aydin et al. (2001) proposed a metric where a *reward* is assigned for completing each job, varying based on how much of the optional part is allowed to execute. They assumed linear or concave (non-increasing derivative) nondecreasing reward functions and periodic tasks. They demonstrated that the maximum reward can provably be achieved by a system where the same amount of optional computation happens for each job of a task. They then provided a system of equations (linear in the case of linear reward functions) that can be solved to determine the optimal amount of optional computation for each task.

4 Bounded Tardiness/Lateness

In this section, we review work that uses a notion of SRT correctness that requires "bounded tardiness" or "bounded lateness."

In order to do so, we will first introduce some additional definitions that are used in these works. Suppose a job is released at time r, has an absolute deadline at time d, and completes at time t. Then, as depicted in Figure 3, its *response time* is $t-r$, its *lateness* is $t-d$, and its *tardiness* is $\max\{0, t-d\}$. Observe that, if a job completes no earlier than its deadline, then its lateness and tardiness are identical and nonnegative. Otherwise, its lateness is negative and its tardiness is zero.

With these definitions in place, we now specify the definition of SRT that we focus on in this section: *bounded lateness*. If a task has an upper bound on the lateness of any of its jobs, then such a bound is called a *lateness bound*. If all tasks have lateness bounds, then the *system* has bounded lateness. *Bounded tardiness* (with *tardiness bounds*) and *bounded response times* (with *response-time bounds*) are equiv-

alent to bounded lateness in the sense that a system has bounded lateness if and only if it has bounded tardiness and if and only if it has bounded response times. All three of these criteria are useful because each guarantees that each task receives a sufficient processor share in the long term. Some of the works in this section has used the bounded tardiness criterion for SRT. Other more recent works reviewed here have used bounded lateness because lateness bounds can indicate that jobs must finish *before* their deadlines, whereas tardiness bounds cannot.

Work on bounded tardiness and bounded lateness has typically been performed on multiprocessor systems, primarily due to the relative simplicity of optimal schedulers on uniprocessors.

4.1 Review of EDF Scheduling

A widely studied uniprocessor scheduling algorithm is the *earliest-deadline-first* (*EDF*) scheduling algorithm, in which jobs are prioritized by absolute deadline, with ties broken arbitrarily but consistently. Most of the work described in this section is based on schedulers derived from EDF.

In order to describe the properties of EDF that make it useful, we first define some terms. When considering HRT scheduling, a schedule is said to be *HRT correct* if no job misses its deadline. When considering the type of SRT scheduling described in this section, a schedule is said to be *SRT correct* if it has bounded lateness. A task system is *HRT* (*respectively, SRT*) *feasible* if some scheduling algorithm can generate an HRT- (respectively, SRT-) correct schedule. A scheduler is said to be *HRT* (*respectively, SRT*) *optimal* if it generates an HRT- (respectively, SRT-) correct schedule for every HRT- (respectively, SRT-) feasible task system. On uniprocessor platforms, EDF is both HRT optimal and SRT optimal. In particular, EDF can correctly schedule any implicit-deadline task system with $\sum_{\tau_i \in \tau} U_i \leq 1$. An example EDF schedule is depicted in Figure 4.

In the rest of this section, we mainly consider multiprocessor platforms and use m to denote the number of processors. There are multiple ways to extend EDF scheduling to a multiprocessor setting. One method is *partitioned EDF* (*P-EDF*). Under P-EDF, each task is statically assigned to a processor, and each processor schedules its tasks using EDF. For implicit-deadline task systems, assigning tasks to processors is equivalent to solving a bin-packing-like problem. The items are the n tasks, with weights equal to utilizations, and the bins are the m processors, each with capacity one.

The primary limitation of P-EDF is related to the bin-packing problem: there are task systems that are feasible on m processors with techniques other than partitioning, but that cannot be partitioned onto the same set of processors. As an example, consider the task system with three identical tasks $(2,3)$. Each task has a utilization of $\frac{2}{3}$, so no two tasks can be allocated on the same processor and three processors are required. However, this task system is actually feasible using only two processors. As an example, Figure 5 depicts an HRT-correct schedule for this task system

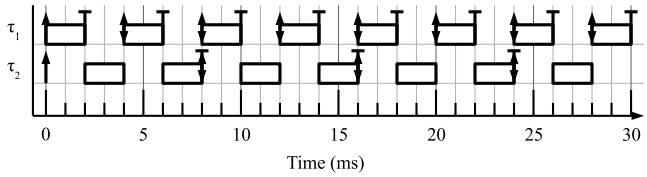


Fig. 4 EDF schedule of $\tau_1 = (2,4)$ and $\tau_2 = (4,8)$.

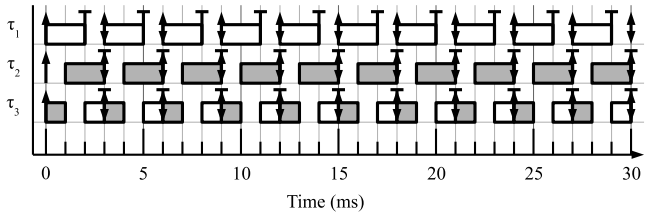


Fig. 5 Correct (both HRT and SRT) schedule of a system with three tasks where each $\tau_i = (2,3)$.

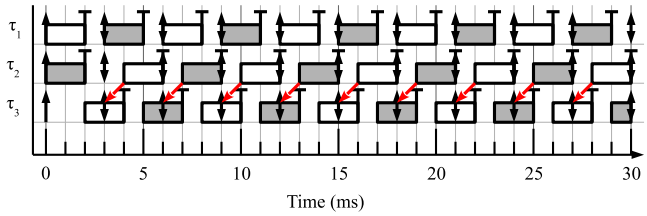


Fig. 6 G-EDF schedule of the same system as Figure 5. This schedule is SRT correct, but not HRT correct.

on only two processors when all jobs are released as early as possible. Notice that in this schedule, jobs of τ_3 *migrate* between processors during execution.

An alternative to P-EDF is *global EDF* (*G-EDF*), in which all processors share a global run queue and the m jobs with the soonest deadlines execute. A G-EDF schedule of our running example (as in Figure 5) is depicted in Figure 6. Unfortunately, as can be seen in the figure, all jobs of τ_3 miss their deadlines. This demonstrates that G-EDF is not HRT optimal. However, notice that no job of τ_3 misses its deadline by more than 1 ms. In fact, Devi and Anderson (2008) demonstrated that G-EDF is in fact SRT optimal.

Schedulers that are HRT optimal for implicit-deadline sporadic task systems do exist, e.g., (Anderson and Srinivasan, 2004; Baruah et al., 1996; Compagnin et al., 2014; Funk et al., 2012, 2011; Megel et al., 2010; Nelissen et al., 2012a,b, 2014; Regnier et al., 2011; Zhu et al., 2011). However, most such schedulers are difficult to implement in practice and all cause jobs to frequently be preempted by other jobs or migrated between CPUs. Even the schedule in Figure 5, which is for a very simple task system, requires each of τ_3's jobs to incur a migration. Furthermore, in order to

achieve optimality, it is necessary to change the relative priorities of jobs while those jobs are running. In Figure 5, each of τ_3's jobs initially has a higher priority than the corresponding job of τ_2, but only for 1 ms. This type of priority change, which does not occur under G-EDF, can cause problems for locking protocols Brandenburg (2011). Therefore, G-EDF remains a good choice for SRT systems for which bounded tardiness is acceptable.

On systems with a large number of processing cores, the overheads incurred by locking and maintaining a global run queue may result in large overheads (Bastoni et al., 2010). Therefore, a compromise between P-EDF and G-EDF called *clustered EDF* (*C-EDF*), where tasks are partitioned onto clusters of CPUs and G-EDF is used within each cluster, is preferable in such cases. Because G-EDF is used within each cluster, work analyzing G-EDF can also be applied in a straightforward manner to C-EDF.

4.2 Work on Bounded Lateness and Bounded Tardiness Without Overload

The seminal work on bounded tardiness was that by Devi and Anderson (2008), who considered G-EDF scheduling. They compared G-EDF to an *ideal scheduler* that continuously maintains for each task a processor share equal to its utilization. The difference in allocation between what a task receives under G-EDF and under the ideal scheduler is called *lag*. Lag can be analyzed at various points in the schedule in order to derive tardiness bounds. The most significant time instants in the analysis occur when all CPUs become simultaneously busy at that very instant. Because some processor was idle, there can be at most $m-1$ tasks that have remaining work just before such a time. That insight allowed Devi and Anderson to define a value x such that the tardiness of a task τ_i is at most $x+C_i$. The value of x they defined is as follows:

$$x \triangleq \frac{C_{\text{sum}} - C_{\text{min}}}{m - U_{\text{sum}}},$$

where C_{sum} is the sum of the $m-1$ largest values of C_i, C_{min} is the smallest value of C_i, and U_{sum} is the sum of the $m-2$ largest values of U_i.

Bounded tardiness is established by mathematical induction over a set of jobs. We denote job k of task τ_i with $J_{i,k}$. When analyzing a job $J_{i,k}$ with a deadline at $d_{i,k}$, jobs with lower priority than $d_{i,k}$ can be ignored. Induction begins with the highest-priority job in the system, and the inductive assumption is that no job with priority higher than $J_{i,k}$ has tardiness larger than stated in the proof. The lag is tracked inductively at key points in the execution of the system, so that a bound on the lag of the system at $d_{i,k}$ can be determined. From that lag bound the tardiness bound for $d_{i,k}$ is established.

Leontyev and Anderson (2010) performed significant extensions to Devi and Anderson's initial work. Rather than limiting their analysis to G-EDF, they considered a broader class of *window-constrained* schedulers. Under such a scheduler, jobs are

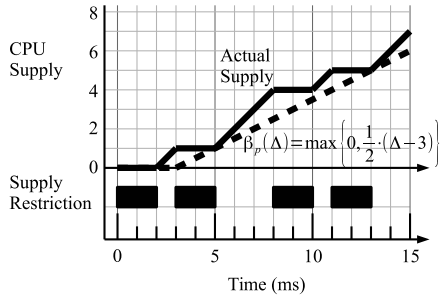


Fig. 7 Depiction of the service functions used by Leontyev and Anderson (2010).

prioritized on the basis of a *priority point* (*PP*), and the system executes the eligible jobs with the earliest PPs. Furthermore, a job's PP may change with time, but there must exist constants ϕ_i and ψ_i such that, if a job of task τ_i has a release at time r, a deadline at time d, and a PP at time y (priority), then $r - \phi_i \leq y \leq d + \psi_i$ holds. By using the absolute deadline of each job as its PP, we see that G-EDF is a window-constrained scheduling algorithm.

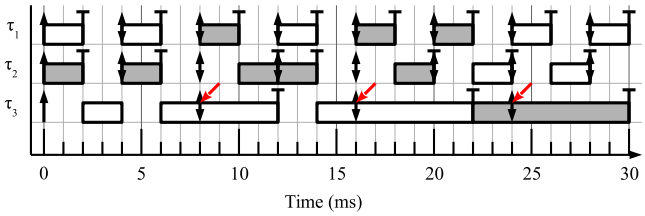
Leontyev and Anderson also considered situations in which processing supply may be *restricted*. Such restrictions are conceptualized by defining a *service function* (following from Chakraborty et al. (2003)) $\beta_p(\Delta)$ for each CPU p, indicating that in any interval of length Δ, at least $\beta_p(\Delta)$ units of time on CPU p are available to execute tasks. The form of the service functions used by Leontyev and Anderson is depicted in Figure 7. Each CPU p has an *available utilization* $\widehat{u_p}$ and a *blackout time* σ_p, so that

$$\beta_p(\Delta) \triangleq \max\{0, \widehat{u_p} \cdot (\Delta - \sigma_p)\}.$$

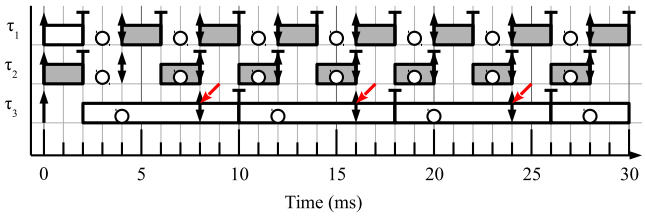
In Figure 7, we assume that the same pattern of supply restriction continues indefinitely; in this case, $\widehat{u_p} = \frac{1}{2}$ and $\sigma_p = 3$. $\widehat{u_p}$ indicates the long-term utilization of processor p. For example, in Figure 7, half of the CPU time is occupied by supply restriction. σ_p is set to the x-intercept necessary in order for $\beta_p(\Delta)$ to lower-bound the actual supply, when the slope of $\beta_p(\Delta)$ is $\widehat{u_p}$.

The proof structure used by Leontyev and Anderson is similar to that used by Devi and Anderson, but much additional complexity is added by the generalizations applied. For the same reasons, the tardiness bounds are significantly more complex, so we refer the reader to (Leontyev and Anderson, 2010) for full expressions.

Leontyev et al. (2011) considered a task model that is more general than the sporadic task model, using a framework called *real-time calculus*. They considered *delay bounds*, which correspond to response-time bounds under the sporadic task model. As discussed above, requiring bounded response times is equivalent to requiring bounded lateness and bounded tardiness. Leontyev et al. provided a method to determine whether a given set of response-time bounds could be met. Again, the expressions are complex, so we refer the reader to (Leontyev et al., 2011) for details.



(a) G-EDF schedule, where $Y_i = D_i$ for all i.



(b) A different GEL schedule (in this case, G-FL) of the same task system.

Fig. 8 Comparison of two GEL schedules of the same task system, with $\tau_1 = \tau_2 = (2,4)$ and $\tau_3 = (8,8)$.

Leontyev et al. also provided a method to determine lateness bounds for a family of *G-EDF-like* (*GEL*) schedulers. Recall that, under G-EDF, jobs are prioritized based on their absolute deadlines, and the absolute deadline of each job of τ_i is D_i units of time after its release. Under a GEL scheduler, jobs are prioritized based on fixed (nonchanging) PPs that may differ from absolute deadlines. In an analogous manner to G-EDF and using absolute deadlines, a job under a GEL scheduler has a higher priority than another if it has an earlier PP. A per-task constant Y_i (priority) takes the place of D_i: the PP of each job is Y_i time units after its release. The implementation of any GEL scheduler is identical to that of G-EDF, except that Y_i is used for prioritization in place of D_i.

An example comparing two GEL schedulers is depicted in Figure 8. Figure 8(a) depicts G-EDF itself, where $Y_i = D_i$ for all i, and Figure 8(b) depicts a different GEL scheduler, the *global fair lateness* (*G-FL*) scheduler proposed by Erickson et al. (2014), as discussed below.

While Leontyev et al. (2011) provided analysis for arbitrary GEL schedulers, they did not provide substantial guidance on how to select values of Y_i in order to obtain desired scheduler characteristics. Furthermore, although they allowed delay bounds to be specified, they did not provide an efficient method to obtain the tightest possible delay bounds using their analysis, and the bounds provided are not as tight as possible for sporadic task systems given the more general task model considered. Erickson et al. (2014) addressed these limitations using an analysis framework similar to that of Devi and Anderson (2008).

Compared to Devi and Anderson (2008), Erickson et al. provided further improvements on the tightness of tardiness/lateness bounds, and also provided a way to handle arbitrary deadlines (deadlines may differ from minimum separation times) and arbitrary GEL schedulers. Their method does not require the additional pessimism from the more general models considered by Leontyev and Anderson (2010) and Leontyev et al. (2011). Erickson et al. also provided methods to choose the best lateness bounds by optimizing parameters such as maximum or average lateness.

As discussed above, Devi and Anderson define the tardiness bound for τ_i as $x + C_i$, with a single value of x for the entire task system. One fundamental change in Erickson et al.'s analysis is to define a separate x_i for each τ_i. They also allow for relative PPs that differ from minimum separation times, which allows consideration of both arbitrary deadlines and arbitrary GEL schedulers.

The tardiness bound $x + C_i$ from Devi and Anderson is equivalent to a response-time bound of $D_i + x + C_i$. In the analysis of Erickson et al., Y_i replaces D_i, so response-time bounds are of the form $Y_i + x_i + C_i$. Stated as lateness bounds, they are of the form $Y_i + x_i + C_i - D_i$.

Erickson et al. defined a term

$$S_i(Y_i) \triangleq C_i \cdot \max\left\{0, 1 - \frac{Y_i}{T_i}\right\},$$

that accounts for the difference between Y_i and T_i, and they use it to provide the following bound on x_i.

$$x_i \geq \frac{\sum_{m-1 \text{ largest}}(x_j U_j + C_j - S_j(Y_j)) + \sum_{\tau_j \in \tau} S_j(Y_j) - C_i}{m} \tag{1}$$

Notice that x_i effectively appears on *both* sides of (1), so (1) cannot be used directly to compute x_i. However, Erickson et al. showed how to define a linear program in order to determine the smallest values of x_i that satisfy (1) for all i. Furthermore, if each Y_i is treated as a variable rather than as a constant, one can also use linear programming to select Y_i values in order to optimize any linear criterion of lateness bounds, such as minimizing the maximum or average lateness bound.

Erickson et al. also proposed G-FL, the same scheduler that was depicted in Figure 8(b). Under G-FL, for each τ_i,

$$Y_i \triangleq D_i - \frac{m-1}{m} \cdot C_i.$$

As can be seen in Figure 8, G-FL can provide better observed lateness than G-EDF. Additionally, it can provide better lateness bounds. Erickson et al. also showed that it provably provides the smallest possible maximum lateness bound, given their analysis.

For the particular case of G-EDF, Valente (2016) provided a method to tighten tardiness bounds further. He used a lag proof similar to that of Devi and Anderson (2008). He demonstrated that lags for different tasks within a task system have a "balancing" effect on each other. Using this observation, he was able to provide

tighter bounds for G-EDF than were possible with earlier research, albeit using an algorithm that requires exponential time to compute them in the worse case.

Erickson and Anderson (2011) proposed a task-system modification that can further reduce lateness. Recall that, because a task is a single-threaded process, each job must wait to begin executing until its predecessor completes. This is an *intra-task precedence constraint*. If jobs run in separate threads, however, this constraint can be removed, and multiple jobs of the same task can execute at the same time on different processors. Doing so can further reduce lateness bounds.

Some of the pessimism in previous lateness bounds results directly from the fact that work can be backed up within a task, even when there are idle CPUs. It is possible that a task has several jobs that have sufficient priority to run, but only one can make progress. Without the intra-task precedence constraint, however, multiple pending jobs from the same task can make progress at the same time. This change allows for smaller bounds.

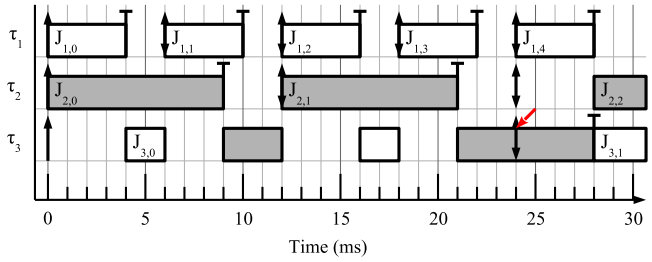
Furthermore, in the presence of the intra-task precedence constraint, the amount by which a task is backed up can grow unboundedly even when there are idle CPUs. Therefore, it is necessary that $U_i \leq 1$ holds for every task. However, without the intra-task precedence constraint, this requirement is no longer necessary, and the simple system utilization requirement $\sum_{\tau_j \in \tau} U_j \leq m$ is sufficient.

Erickson and Anderson (2012) proposed another modification to the scheduler to improve lateness bounds. The lateness bounds from Erickson et al. (2014) depend heavily on task execution times. A task's execution time can be reduced by an integral factor if each of its jobs is split. For example, a task that has a WCET of 2 ms and a period of 4 ms could have its jobs split in half, resulting in a task with a WCET of 1 ms and a period of 2 ms. Notice that the utilization of the task remains constant. Each consecutive pair of subjobs in the split task corresponds to a real job in the original task.

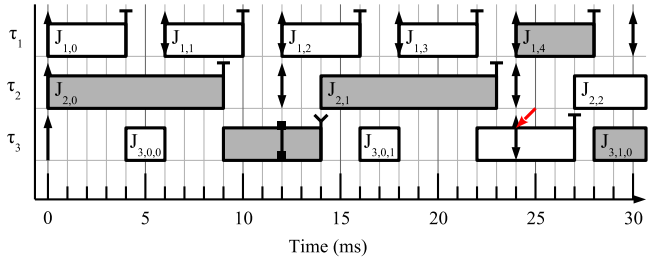
An example of job splitting under G-EDF is depicted in Figure 9. Figure 9(a) depicts an example schedule in the absence of splitting. Notice that $J_{3,0}$ completes 4 ms late. Figure 9(b) depicts the schedule where jobs of τ_3 are split into two subjobs. $J_{i,j,k}$ is used to denote subjob k of $J_{i,j}$. Notice that $J_{3,0}$ now completes only 3 ms late.

Job splitting becomes more complicated in the presence of critical sections, because many locking protocols require that job priorities do not change during execution, but every time a subjob ends, the priority of the underlying job changes. However, this problem can be overcome by not allowing a subjob to end while holding or waiting for a lock, reducing the length of the subsequent subjob. This procedure is depicted in Figure 9(c), where $J_{3,0,0}$ runs for 8 ms instead of 7 ms, and $J_{3,0,1}$ then runs for only 6 ms.

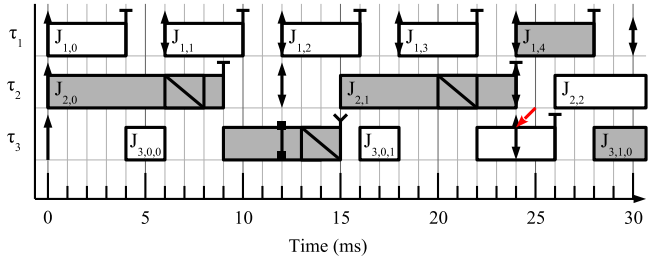
In the absence of overheads and critical sections, because task utilizations remain constant with splitting, lateness bounds could be made arbitrarily close to zero. However, on a real system, more overheads are incurred as a result of job splitting. Whenever a subjob ends, the operating system must decide what job should subsequently be scheduled, creating more scheduling decisions. Additionally, jobs may be preempted at subjob completion, rather than only at job releases, causing a po-



(a) No splitting.



(b) Each job of τ_3 split into two subjobs.



(c) Each job of τ_3 split into two subjobs, in the presence of critical sections.

Fig. 9 Schedules of a task system with $\tau_1 = (4,6)$, $\tau_2 = (9,12)$, and $\tau_3 = (14,24)$, to illustrate job splitting.

tential loss of cache affinity. These additional overheads effectively *increase* a task's utilization, so it is necessary to account for these overheads in order to determine the actual benefits of job splitting.

Erickson and Anderson's lateness analysis remains correct if jobs are allowed to begin execution prior to their proper release times, as long as job PPs are determined based on their proper release times. Therefore, when one subjob completes, it is sufficient to simply lower the priority of the underlying job. It is not necessary to unconditionally preempt the job. Furthermore, even if the job does need to be preempted, it can simply be added to the ready queue immediately; it is not necessary

to set a timer for a future release. This approach significantly limits the additional overheads that splitting creates.

Provisioning SRT applications based upon the worst case may be overkill in many settings. Indeed, by inspecting the tardiness and lateness bounds given above, it is evident that such bounds can be reduced if tasks are provisioned using average-case execution times rather than worst-case times. Mills and Anderson (2011) explored this possibility and showed that the prior tardiness analysis of Devi and Anderson (2008) can be extended so that only average-case task execution times are assumed. The basic idea is to encapsulate each task within a single-task server that is sporadically allocated execution budget based on that task's provisioned average-case execution time. A single job of a task may require budget from multiple server invocations to complete. Mills and Anderson showed that such an approach allows tardiness to be bounded in expectation. In later work, Liu et al. (2014) revisited the independence assumptions applied in the analysis of Mills and Anderson and showed that such assumptions can be relaxed.

5 Overload Management Using Value Functions

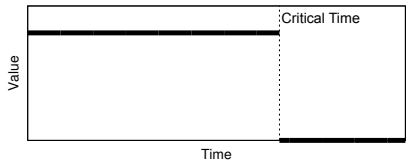
In this section, we discuss prior work on scheduling algorithms that use value functions to define correct behavior during overload.

5.1 Locke's Best-Effort Heuristic

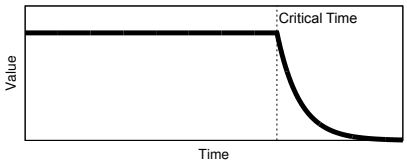
Locke (1986) considered a system that resembles the sporadic task model, scheduled globally on a multiprocessor. Most of the other papers in this section are written for special cases of this model, so we review it in some detail. However, rather than having a per-task *upper bound* on job execution times, there is a stochastic per-task *distribution* of execution times. Similarly, rather than having a per-task *lower bound* on separation time between job releases, job releases follow a stochastic per-task distribution. Given these modifications to the task system, it is possible for the system to experience overload if there is a burst of jobs that either are released closer together than generally expected, or that run for longer than generally expected.

Locke (1986) assigned to each task a *value function* that specifies the value to a system of completing a job at a particular time after its release. "Value" is a unit-less quantity that can be compared between jobs, to determine which job to complete in the event of an overload. Ideally, the system should accrue as much total value as possible.

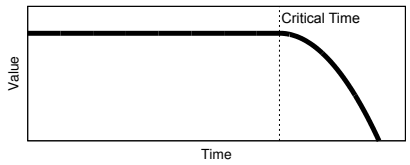
Examples of value functions are depicted in Figure 10. In each example, the x axis represents the completion time of a job after its release, while the y axis represents the value to the system from completing that job. For example, suppose τ_i is the task considered in Figure 10(a). If some $J_{i,k}$ completes before the time marked



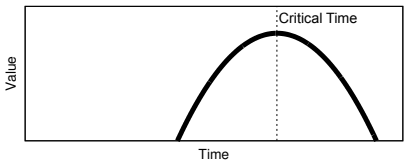
(a) Value function with step at critical time.



(b) Value function with exponential drop-off after critical time.



(c) Value function with quadratic drop-off after critical time.



(d) Value function with a specific target completion time, and quadratic increase and drop-off before and after that time, respectively.

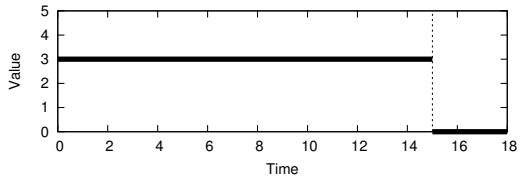
Fig. 10 Example value functions from Locke (1986).

"Critical Time," then the system achieves some constant value. However, if the job completes after that time, the system receives no value whatsoever. Thus, the system should only execute $J_{i,k}$ if it is possible for $J_{i,k}$ to complete before its critical time. Furthermore, suppose there are two tasks τ_i and τ_j that each have value functions of this form and that at time t there are ready jobs $J_{i,k}$ and $J_{j,\ell}$. If it is possible to complete either $J_{i,k}$ or $J_{j,\ell}$ before its respective critical time, but not to complete both before their respective critical times, then $J_{i,k}$ should generally be selected if τ_i has a higher constant value than τ_j, and $J_{j,\ell}$ should be selected if the reverse is true. (This is not strictly true because the choice of $J_{i,k}$ or $J_{j,\ell}$ may affect the ability to complete other jobs at appropriate times.)

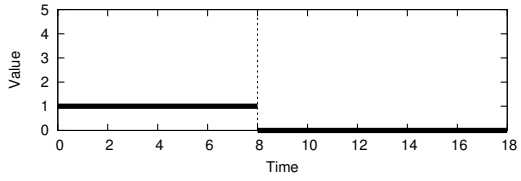
Although not required by the definition of "value function," for tractability Locke (1986) considered value functions that are continuous and have continuous first and second derivatives, except for (possibly) a single discontinuity at the *critical time*. This is why the time of the discontinuity in Figure 10(a) is labelled as the "critical time." Although the step function discussed above is the simplest value function, Locke proposed others. Figure 10(b) depicts a value function that drops off exponentially after the critical time, indicating that there is still some value to completing jobs late, but this value rapidly drops off as jobs complete later. Figure 10(c) depicts a value function that drops off more slowly after the critical time, indicating that completing jobs slightly late has a smaller impact than for the value function in Figure 10(b). Finally, as depicted in Figure 10(d), it is also possible to use value functions to indicate that a job should not complete too *early*. In this case, a job that finishes very quickly will achieve zero value, just as if the job finished very late.

The system should try to achieve the maximum cumulative value even if an overload occurs. However, there are two difficulties that arise in attempting to do so: uncertainty about system behavior and the intractability of the scheduling problem.

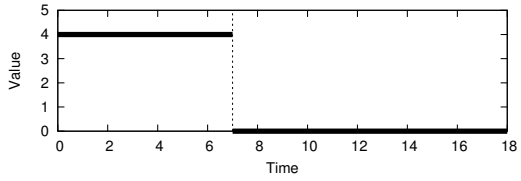
Locke assumed that the system does not know the timing of job releases until they occur and does not know the actual run time of each job until it completes. We show by example that even the first assumption is itself sufficient to prevent the system from always maximizing the cumulative value. Consider the task system with value functions depicted in Figure 11, as scheduled on a uniprocessor. Suppose that τ_1 releases $J_{1,0}$ and τ_2 releases $J_{2,0}$ at time 0 and that no other job is released before time 15. Further suppose that $J_{1,0}$ is known to require 14 ms of execution, while the job of $J_{2,0}$ is known to require 7 ms of execution. This scenario is depicted in Figure 12(b)–(c). Because these two jobs together require 21 ms of execution, while their last critical time is at time 15, the system cannot complete both jobs before their critical times. It is therefore better for it to select $J_{1,0}$ as in Figure 12(b), in order to achieve a cumulative value of three, rather than selecting $J_{2,0}$ as in Figure 12(c), which would only achieve a cumulative value of one.



(a) Value function for τ_1.



(b) Value function for τ_2.

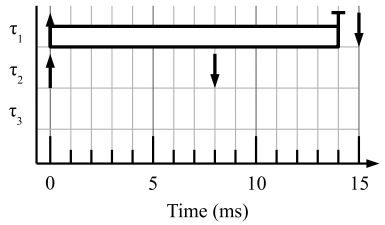


(c) Value function for τ_3.

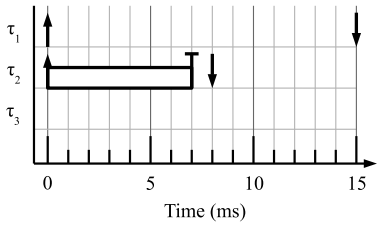
Fig. 11 Value functions for an example task system.

Suppose, however, that τ_3 actually releases $J_{3,0}$ at time 8, and that $J_{3,0}$ is known to require 6 ms of execution. Because $J_{3,0}$ can complete with a value of 4, which is greater than the value that can be achieved by either $J_{1,0}$ or $J_{2,0}$, the system should execute $J_{3,0}$ to maximize the cumulative value. If the system initially chose to execute $J_{1,0}$, as depicted in Figure 12(d), then because $J_{1,0}$ does not actually complete,

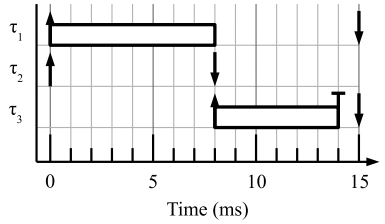
(a) Key for schedules in this figure.



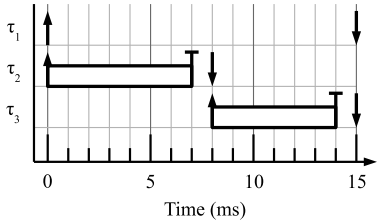
(b) Schedule running only the job from τ_1, with a cumulative value of 3.



(c) Schedule running only the job from τ_2, with a cumulative value of 1.



(d) Schedule running jobs from τ_1 (though not to completion) and τ_3, with a cumulative value of 4.



(e) Schedule running jobs from τ_2 and τ_3, with a cumulative value of 5.

Fig. 12 Several possible schedules for the task system with value functions depicted in Figure 11. $J_{1,0}$ is released at time 0 and is known to have an execution time of 14. $J_{2,0}$ is also released at time 0 and is known to have an execution time of 7. In (d) and (e), $J_{3,0}$ is released at time 8 and is known to have an execution time of 6.

the cumulative value achieved is only four. However, if the system initially chose to execute $J_{2,0}$, as depicted in Figure 12(e), then because $J_{1,0}$ completes, the cumulative value achieved is five. Therefore, *the optimal choice at time* 0 *depends on whether* $J_{3,0}$ *is released at time* 8, so making an optimal choice is impossible under Locke's assumptions. A similar example could be constructed if the execution times were unknown.

Locke also noted that, even if optimal decision-making were possible, the problem is likely to be NP-complete. However, the results of this decision-making process are most important *precisely when the system is already overloaded and cannot complete all jobs*. Furthermore, *running the scheduling algorithm requires the same computing resource as the jobs themselves*. Therefore, running an optimal scheduling algorithm would cause more harm than it prevents. Thus, Locke could have chosen to develop either a heuristic algorithm or an approximation algorithm with a corresponding provable bound on achieved value, but he chose the former.

Locke's heuristic algorithm is based on the assumption that, under typical circumstances, it will be possible for nearly every job to complete at a time that allows it to achieve nearly all of its possible value. This assumption simply means that the system was properly provisioned for the common case. In order to exploit this assumption, Locke assigned for each job a *deadline* that is the latest time it can complete while continuing to achieve a user-configurable fraction of its maximum achievable value. In the case of a step value function, as in Figure 10(a), a job's deadline is simply its critical time. However, under any of the other types of value functions depicted in Figure 10, a job's deadline is usually after its critical time. Locke's heuristic simply prioritizes all jobs by deadline until the probability of a deadline miss exceeds a user-configurable threshold.

Once a deadline miss is likely, the system switches prioritization to a heuristic based on *value density*. The value density for $J_{i,k}$ at time t is computed as follows. Let $e^r_{i,k}(t)$ be the expected remaining execution time for $J_{i,k}$ at time t, conditioned on how long it has already executed. The expected value $V(t)$ of $J_{i,k}$ at time t is defined to be the value that $J_{i,k}$ will accumulate if it completes at time $t + e^r_{i,k}(t)$. The value density of $J_{i,k}$ at time t is simply $V(t)/e^r_{i,k}(t)$. For example, consider the schedule in Figure 12(d). At time $t = 0$, $J_{1,0}$ is expected to have 14 units of execution remaining, completing at time 14, and has an expected value of three, resulting in a value density of $3/14$. At time $t = 8$, $J_{1,0}$ is expected to have $14 - 8 = 6$ units of execution remaining, still completing at time 14, and has an expected value of three. Its value density is now $3/6 = 1/2$. The heuristic that the system uses during overload, when it is likely that some job will miss its deadline, is to prioritize jobs by decreasing value density.

Locke demonstrated the effectiveness of his heuristic through experiments that simulate global multiprocessor schedules where one CPU is dedicated to making scheduling decisions for the rest of the system. He demonstrated that his scheme provides significantly higher achieved value than other considered schedulers in the presence of overload, while also meeting most deadlines in the absence of overload. However, he did not provide any theoretical guarantees comparing the achieved value to the maximum achievable value.

5.2 Providing a Guarantee on Achieved Value: D^*

Unless otherwise noted, all papers discussed in the remainder of Section 5 consider only step value functions, as depicted in Figure 10(a). In such cases, we say that the *deadline* of each job is simply its critical time, and that its *value* is the value that it achieves if it completes before its critical time. Furthermore, the job's *value density* is simply its value divided by its total execution time. (This differs from the notion of "value density" used by Locke (1986), who used remaining execution time.) We

use the constant q to denote the *importance ratio*, or the ratio of the largest value density in the system to the smallest value density in the system.[2]

Baruah et al. (1991) considered scheduling on uniprocessors. They observed that Locke (1986) provided only heuristics, but did not provide any guarantee about the value that could be achieved during an overload. In order to provide such a guarantee, they developed a new scheduling algorithm, D^*. They assumed that job release times are not known ahead of time, but that the exact execution time of each job is known upon release.

D^* is similar to the later-proposed *earliest deadline until zero laxity* (*EDZL*) scheduling algorithm (Baker et al., 2008; Lee, 1994). A job's *laxity* is the time until its deadline minus its remaining execution time. If it reaches a *zero-laxity* state, then it must be scheduled immediately, or it will miss its deadline. Like EDZL, D^* behaves identically to EDF until some $J_{i,k}$ reaches a zero-laxity state. If no other job is in a zero-laxity state when this occurs, then $J_{i,k}$ runs immediately. To handle the case when some $J_{j,\ell}$ is already in a zero-laxity state, D^* maintains the sum of the values of all jobs that have been preempted in a zero-laxity state since the last successful job completion. Such preempted jobs have been abandoned, as it was impossible for them to meet their deadlines. If $J_{i,k}$ has a value greater than this sum plus the value of $J_{j,\ell}$, then $J_{i,k}$ preempts $J_{j,\ell}$, $J_{j,\ell}$ is abandoned, and the sum is updated. Otherwise, $J_{i,k}$ is abandoned.

Baruah et al. (1991) proved that, if value densities are normalized such that the smallest is at least one, the total value achieved during an overloaded interval using D^* is at least $1/5$ the *length* of that interval. Although this value is small, they proved that no algorithm can guarantee more than $1/4$ of the length of such an interval without knowing job releases ahead of time. Baruah et al. also showed experimentally that D^* performs similarly to Locke's best effort scheduler in the common case, but provides drastically better behavior in certain pessimistic cases.

5.3 Providing the Optimal Guarantee: D^{over}

Koren and Shasha (1995b) provided a scheduler D^{over} that can guarantee an achieved value of $1/4$ of the length of an overloaded interval, closing the gap between D^* and the theoretical limit. The design of D^{over}, like the design of D^*, is based on EDF and is identical to EDF until an overload occurs.

Even during underload, D^{over} maintains two sets of ready jobs, not including the currently running job: *waiting* jobs and *privileged* jobs. If a job is preempted by a normal job release, then it becomes a privileged job. The system keeps track of the amount of time that a newly arriving job can execute without causing the current job or any privileged job to miss its deadline. When a new job arrives, if its execution cost is less than this time, it preempts the current job. Otherwise, because adding the new job could cause some existing job to miss its deadline, an overload has

[2] Several of the papers cited herein use k for the importance ratio, but we use q ($q$uotient) to avoid conflict with the job index k.

occurred. Therefore, the new job is instead added to the queue of waiting jobs. This strategy ensures that a privileged job can never reach a zero-laxity state.

Let $V_{j,\ell}$ denote the value of $J_{j,\ell}$. When waiting $J_{i,k}$ reaches a zero-laxity state, then either it must be scheduled immediately or it is not worth running at all. However, running it may prevent other jobs from running. To determine whether is worth running, its value is compared to $(1-\sqrt{q})\cdot(\sum_{J_{i,k}\in\Theta} V_{j,\ell})$, where Θ is the set containing all privileged jobs and the currently running job. If its value is larger than this expression, then $J_{i,k}$ preempts the currently running job, and all privileged jobs become waiting jobs. Otherwise, $J_{i,k}$ is discarded.

Koren and Shasha (1995b) also demonstrated that D^{over} achieves the optimal *competitive ratio* of $\frac{1}{(1+\sqrt{q})^2}$. In other words, D^{over} is guaranteed to achieve at least $\frac{1}{(1+\sqrt{q})^2}$ times as much value in an overloaded interval as could be achieved by a clairvoyant algorithm. Baruah et al. (1991) demonstrated that no better competitive ratio is possible.

5.4 Providing Guarantees on Multiprocessors: MOCA

While D* and D^{over} provide guarantees on a *uniprocessor*, Koren and Shasha (1994) proposed the *multiprocessor on-line competitive algorithm* (*MOCA*) to provide such guarantees on a *multiprocessor*. MOCA requires an even number of processors and works by dividing the system into $m/2$ *bands* of two processors, as depicted in Figure 13. ψ of the bands are designated with specific value densities, and ω form a *central pool*. It must be the case that $\psi+\omega=m/2$, so that each CPU is assigned to exactly one band. Each band contains a *safe processor* for executing jobs that can be guaranteed to meet their deadlines and a *risky processor* for executing jobs for which such a guarantee cannot be made.

When a job is released, the system first tries to assign it to the band designated for its value density. If it can be assigned to the safe processor without compromising the guarantees made to other jobs that are already on that safe processor, the system assigns it there. Otherwise, the system tries to assign it to the safe processor for a band designated for lower value density, considering such bands in decreasing value density order. If even that fails, the system then tries to assign it to a safe processor in the central pool, considering such processors in arbitrary order. If all else fails, the system adds it to a list of *waiting* jobs and does not consider it until it reaches a zero-laxity state.

When a waiting job $J_{i,k}$ reaches a zero-laxity state, the system tries to schedule it on a risky processor. Bands are considered in the same order as for safe processors. If it finds an idle risky processor, it begins executing $J_{i,k}$ there. Otherwise, it considers the same set of risky processors as before and finds the one running the $J_{j,\ell}$ with the earliest deadline. If $J_{i,k}$ has a *later* deadline than $J_{j,\ell}$, then $J_{j,\ell}$ is abandoned, and $J_{i,k}$ begins running in its place. Otherwise, $J_{i,k}$ is abandoned. This heuristic is

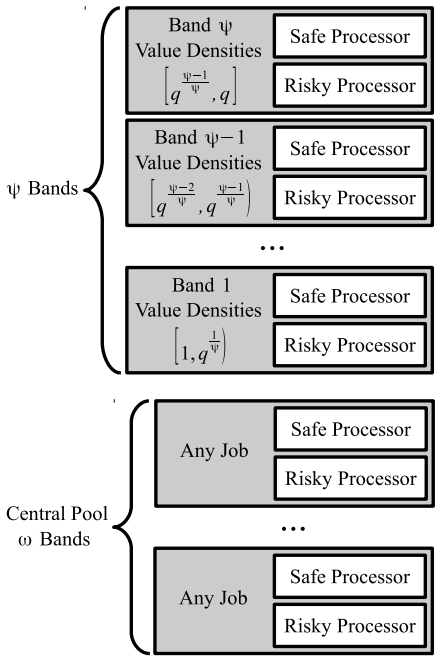


Fig. 13 Grouping of CPUs used by MOCA.

used to minimize the risk of unnecessary idleness on a risky processor, as a job in a zero-laxity state will run continuously until its deadline.

Whenever some safe processor becomes idle, the safe and risky processors within that band switch roles. This guarantees that the job running on the risky processor will complete (as it is now on a safe processor) and ensures that an idle risky processor is now ready to schedule a waiting job that reaches a zero-laxity state.

Koren and Shasha (1994) also showed that no scheduler executing on m processors can achieve a competitive ratio above

$$\frac{q-1}{qm\cdot(q^{\frac{1}{m}}-1)},$$

and that MOCA achieves a competitive ratio of

$$\frac{1}{1+2m\cdot\left(\max_{1\le i\le\psi}\frac{q^{\frac{i}{\psi}}}{\omega+\frac{q^{\frac{i}{\psi}}-1}{q^{\frac{1}{\psi}}-1}}\right)}.$$

ψ should be chosen to maximize this ratio. Observe that MOCA is not necessarily optimal in the sense of achieving the best possible competitive ratio. However, unlike heuristic approaches, it does provide a guarantee.

5.5 Rate-Based Earliest Deadline Scheduling

Buttazzo and Stankovic (1995) proposed the *robust earliest-deadline* (*RED*) scheduler, which uses a model based on value functions. Each task has an associated deadline, value, and *deadline tolerance*. It has a step value function with the critical time at the deadline *plus deadline tolerance*. However, scheduling decisions are based only on deadline, without accounting for deadline tolerance.

Under RED, each task has a WCET that will not be exceeded, but the arrival pattern of jobs is not known. In this respect, its assumptions are like those used by D^*, D^{over}, and MOCA.

When a job is released, it can be accepted or rejected. If it is rejected, it will not run unless slack is created in the future by jobs that underrun their WCETs. In addition to considering value, RED also divides tasks into two classes: *hard* and *critical*. If a hard job is accepted, then it must complete unless overload later occurs. If a critical job is accepted, then it must complete under all circumstances.

At runtime, RED keeps a list of all unfinished accepted jobs, both hard and critical, ordered by deadline. Whenever a new $J_{i,k}$ is released, RED uses the list to determine whether adding $J_{i,k}$ will cause a deadline miss. If it will not, $J_{i,k}$ is immediately accepted. Otherwise, RED will attempt to find one or more hard jobs that can be dropped. Dropped jobs may still be completed if other jobs complete early.

RED always executes the job at the beginning of its list of accepted jobs, thus running the job with the earliest deadline. In the absence of overload, RED reduces simply to EDF. Buttazzo and Stankovic (1995) provided experimental evidence that RED can achieve significantly higher value than other schedulers such as EDF when an overload occurs.

Spuri et al. (1995) proposed the *robust total bandwidth* (*RTB*) scheduler, a similar scheduler to RED. RTB also supports a class of guaranteed[3] periodic tasks that are not subject to being rejected. It does so by scheduling the aperiodic tasks (i.e., the same types of tasks as the hard and critical tasks under RED) inside a *server*. A server is a budgeted container for other tasks. The server can be scheduled with EDF, using a budget to guarantee that it will not interfere with guaranteed periodic tasks. When RTB chooses to schedule that server, it actually executes one of the aperiodic jobs running inside that server. Tasks are accepted or rejected using a similar strategy to RED.

[3] Spuri et al. (1995) use the term "hard periodic" for these tasks, but we use "guaranteed" here to avoid confusion with hard RED tasks.

5.6 Schedulers Accounting for Dependencies

Some work has been performed on scheduling with value functions in the presence of dependencies such as shared resources. These schedulers have additional constraints they must consider, such as needing to let a critical section finish in order to free the resource for another job. For examples, see (Cho, 2006; Clark, 1990; Garyali, 2010; Li, 2004; Li et al., 2006).

6 Overload Management by Changing Minimum Separation Times

Most of the overload management techniques surveyed thus far in this chapter have worked by dropping certain jobs. An alternative technique is to adjust the minimum separation time of a task, slowing down the rate at which it releases jobs.

Adaptive scheduling algorithms allow such a scaling of minimum separation times. Such algorithms were surveyed in detail by Block (2008). However, most of these algorithms are intended for use in systems where high variability in job execution times is expected, and minimum separation times must be decided online for that reason. We are concerned primarily with systems that are provisioned for the common case, but that need to recover from *transient* overloads.

The related problem of choosing new minimum separation times was addressed by Buttazzo et al. (2002), who proposed the *elastic model*. Under the elastic model, tasks are assigned initial and maximum periods, as well as *elasticity factors* that are used to determine the extent of "stretching" of each task. During a transient overload, minimum separation times can be determined based on elasticity factors.

One adaptive scheduling algorithm, the *earliest eligible virtual deadline first* (*EEVDF*) algorithm (Stoica et al., 1996), uses a notion of *virtual time*. We provide a description of EEVDF here.

EEVDF is a *proportional share* scheduling algorithm. Each task is assigned a weight, and each task should receive a processor share that is commensurate with its weight. For example, consider the task system in Figure 14. The actual progression of time is graphed on the bottom axis. From time 0 to time 2, only τ_1 is present in the system. Therefore, it receives all of the CPU time. At time 2, τ_2 enters the system. Because τ_1 has a weight of 4 and τ_2 has a weight of 2, τ_1 receives twice as much processor time as τ_2. Until τ_3 arrives at time 8, τ_1 receives $2/3$ of the processor time and τ_2 receives $1/3$. As long as some task is present, the CPU is never idle.

In order to distribute processor time in accordance with the weights, EEVDF maintains the current *virtual time*. The speed of virtual time relative to actual time depends on the total weight of all tasks in the system. Specifically, if $A(t)$ is the set of active tasks at time t and W_i is the weight of τ_i, then the speed of virtual time at actual time t is $\frac{1}{\sum_{\tau_i \in A(t)} W_i}$, and the virtual time $v(t)$ corresponding to actual time t is

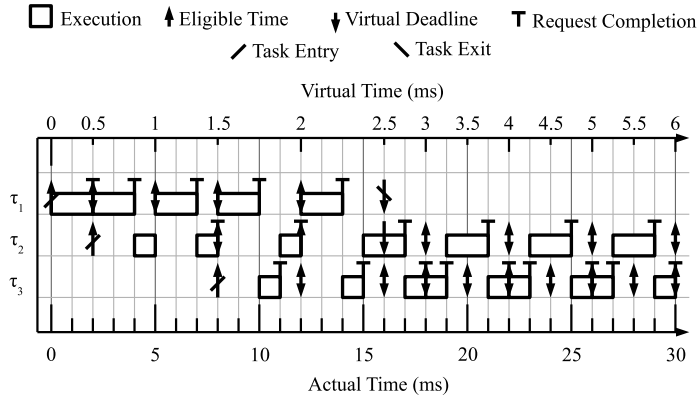


Fig. 14 EEVDF schedule of a task system. τ_1 has a weight of 4 and always issues requests of size 2 ms. τ_2 has a weight of 2 and always issues requests of size 2 ms, although its second request issued at actual time 8 completes early. τ_3 has a weight of 2 and always issues requests of size 1 ms.

$$v(t) = \int_0^t \frac{1}{\sum_{\tau_i \in A(t)} W_i}\, dt.$$

For example, between time 0 and time 2 in Figure 14, only τ_1 is present, with a weight of 4. Therefore, the speed of virtual time in this interval is $\frac{1}{4}$, and $v(2) = \int_0^2 \frac{1}{4}\, dt = 0.5$.

Each task repeatedly makes *requests* for CPU time, making a new request as soon as its previous request has completed (unless it instead exits the system at that time). When a task enters the system, it makes its first request. That request is said to have an *eligible time* at that time. Each request also has an associated size s, indicating the amount of actual time desired for computation. However, it is possible for the task to complete executing its request before it has used a full s units of execution, as τ_2 does in Figure 14 for the request issued at virtual time 1.5.

Once a task completes executing its request, it usually initiates another request. If the just-finished request had an eligible *virtual* time of r and an *actual* execution time of a, the new request has an eligible time at *virtual* time $r + \frac{a}{W_i}$. Alternatively, the task may exit the system at the time its next request would otherwise be eligible, as τ_1 does in Figure 14 at virtual time 2.5.[4]

For example, the first request of τ_1 in Figure 14 has an eligible virtual time of 0 and executes for 2 ms. Therefore, the eligible virtual time for the second request is $0 + \frac{2}{4} = 0.5$. Similarly, the second request of τ_1 has an eligible virtual time of 0.5, as just computed, and also executes for 2 ms. Thus, the eligible virtual time for the third request is $0.5 + \frac{2}{4} = 1$. Observe that the difference between eligible *virtual* times is 0.5 ms in both cases, but the difference between eligible *actual* times is

[4] Stoica et al. (1996) provide more complex rules that allow a task to leave at other times, but we do not consider those here.

2 ms between the first and second request, but 3 ms between the second and third requests. This occurs because the virtual time clock runs more slowly once τ_2 enters the system.

Each virtual request has a *virtual deadline* that is used to determine scheduling priority. If the request has a virtual eligible time of r and a request size of s, then its virtual deadline is at time $d = r + \frac{s}{W_i}$. For example, the first request of τ_1 in Figure 14 has a virtual eligible time of 0 and a request size of 2 ms, so its virtual deadline is $0 + \frac{2}{4} = 0.5$. If a request runs for its full request size, then its virtual deadline is identical to the virtual eligible time of the next request. However, if a request completes early, as happens to the second request of τ_2 that completes at virtual time 2, then the virtual eligible time of the next request may be earlier than the virtual deadline of the just-finished request.

EEVDF prioritizes requests by earliest virtual deadline, considering only requests that have reached their eligible times but have not completed. For example, at virtual time 0.5 in Figure 14, τ_1 has a request with a virtual deadline of 1 and τ_2 has a request with a virtual deadline of 1.5. Because τ_1 has an earlier virtual deadline, its request runs for the requested 2 ms. The next request of τ_1 does not have an eligible time until virtual time 1, so τ_2's request runs until that time. In Figure 14, deadline ties are broken by task index, so when τ_1's third request becomes eligible at virtual time 1, it preempts the executing request of τ_2.

Observe in Figure 14 that τ_3 and τ_2 receive the same processor share, even though their request sizes differ. The request size of τ_2 is always 2 ms (even though the full size may not be used) and the request size of τ_3 is always 1 ms. However, from virtual time 1.5 onward (when τ_3 enters the system), τ_3 releases jobs twice as frequently as τ_2, except for the shift in release time for τ_2 caused by the early completion. This occurs because both tasks have the same weight.

7 Overload Management in Mixed-Criticality Scheduling

Sometimes different applications that will be run on the same physical machine have different requirements for timing correctness. For example, some applications have HRT constraints (requiring all deadlines to be met), while others have SRT constraints (where bounded lateness is acceptable). An example of a system with this sort of requirement is next-generation unmanned air vehicles (UAVs), which will have tasks with different requirements that will realize in software functionality that has traditionally been performed by humans. For example, safety-critical software performing functions such as flight control has stringent HRT constraints, whereas mission-critical software performing planning functions has only SRT constraints. Running both sets of software on the same machine could significantly reduce the size, weight, and power required for the aircraft.

Furthermore, there may be further distinctions in requirements than simply the difference between HRT and SRT constraints. For example, some tasks may be so critical that it is necessary to use WCET estimates determined by a tool that provides

a provable upper bound on execution time, in order to provide the strongest possible guarantee that no WCET is exceeded. Such a level of certainty may be necessary in order for the system to be acceptable to a relevant certification authority. However, for other tasks, it may be sufficient to use less pessimistic WCET estimates, such as those determined by measuring the largest execution on a real system and multiplying by a safety factor.

Under most real-time scheduling analysis, the system can only be deemed correct if it can be proven to be correct *even using the most pessimistic assumptions for all tasks*. For example, in order to prove that the flight-control software will behave correctly, it is necessary to use highly pessimistic WCET estimates for the mission-control software as well. This may result in a system that is unnecessarily underutilized.

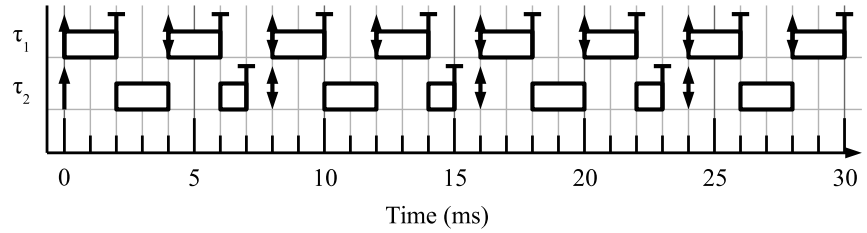
Mixed-criticality scheduling algorithms and analysis address this problem. Vestal (2007) proposed that a single scheduling algorithm could be analyzed under multiple sets of assumptions about WCET estimates. The system has a finite number of criticality levels, and each task is assigned a *criticality level* and, for each criticality level in the system including its own, a *provisioned execution time* (*PET*). For arbitrary level ℓ, the system is considered to be *correct* at level-ℓ if all tasks with a criticality level at or above level ℓ are scheduled correctly, assuming that no job of any task exceeds its level-ℓ PET. An example is depicted in Figure 15 with two criticality levels, A (high) and B (low). Figure 15(a) depicts the worst-case behavior assuming that no job of any task exceeds its level-B PET, and Figure 15(b) depicts the worst-case behavior assuming that no job of any task exceeds its level-A PET. Observe that deadlines are only missed in Figure 15(b), that only τ_2 (which is a level-B task) has jobs that miss their deadlines, and that this schedule involves jobs exceeding their level-B PET. As depicted in Figure 15, statically prioritizing τ_1 over τ_2 correctly schedules the task system.

Observe that guarantees at level ℓ are conditioned on all jobs running for at most their respective level-ℓ PETs. However, it is possible that some task's level-ℓ PET was insufficiently pessimistic and is overrun by some job. This is a form of overload.

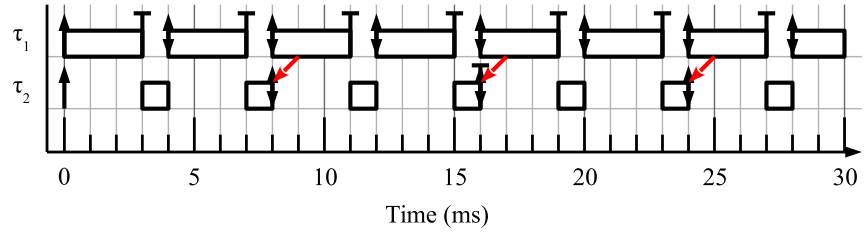
As Santy et al. (2012) pointed out, many mixed-criticality scheduling algorithms respond to such a PET overrun by simply dropping all jobs of level-ℓ tasks from that point forward. However, this is usually an unacceptable response. In Section 7.1, we survey some methods that reduce the number of low-criticality jobs that are dropped, and in Section 7.2, we survey some methods that scale minimum separation times as an alternative to dropping jobs. In Section 7.2 we discuss how to handle PET overruns in a scheduler called MC^2. For a more comprehensive survey of recent work on mixed-critical scheduling, see (Burns and Davis, 2017).

7.1 Techniques to Reduce Dropped Low-Criticality Jobs

Baruah et al. (2010) introduced the *own-criticality based priority* (*OCBP*) technique for determining static task priorities for mixed-criticality scheduling. Traditionally,



(a) Level-B worst-case behavior.



(b) Level-A worst-case behavior.

Fig. 15 Possible schedules for a uniprocessor mixed-criticality system with two criticality levels, A (high) and B (low), both with HRT requirements. Level-A τ_1 has a minimum separation time of 4 ms, a level-A PET of 3 ms, and a level-B PET of 2 ms. Level-B τ_2 has a minimum separation time of 8 ms, a level-A PET of 4 ms, and a level-B PET of 3 ms. τ_1 is statically prioritized over τ_2.

when some job scheduled using this technique overruns its level-ℓ PET, all jobs at levels ℓ and below are dropped from that point forward. However, Santy et al. (2012) proposed three improvements to this technique. For each, suppose that some job $J_{i,k}$ of τ_i overruns its PET.

1. If some τ_j has a lower criticality *but a higher priority* than τ_i, it is not necessary to drop jobs from τ_j. This follows from a property of the analysis.
2. It is possible to set an *allowance* for each such τ_i and criticality level ℓ below that of τ_i, so that if $J_{i,k}$ exceeds its PET by less than that allowance, it is not necessary to drop jobs at level ℓ. This technique is based on the work of Bougueroua et al. (2007), and is enforced by the *Latest Completion Time* (LCT) mechanism that Santy et al. propose.
3. If no jobs at the level of τ_i are eligible for execution, then jobs no longer need to be dropped, and the system can be returned to normal operation.

Santy et al. demonstrated that these techniques can significantly reduce the number of dropped jobs, primarily due to the ability to only temporarily drop jobs from a task.

Santy et al. (2013) proposed two similar mechanisms to stop dropping jobs for low-criticality tasks, but on multiprocessors.

The first mechanism Santy et al. (2013) proposed applies to fixed-priority systems. In order to restore the system to level ℓ, the system keeps track of a series of

times $f_i^{\checkmark}$, ordered by decreasing task priority. $f_0^{\checkmark}$ is the last completion time of a job that overran its level-ℓ PET. For $i > 0$, $f_i^{\checkmark}$ is the earliest time not earlier than $f_{i-1}^{\checkmark}$ such that there is no active job of τ_i. Once $f_n^{\checkmark}$ has been detected, where n is the number of tasks, all tasks with criticalities at least ℓ can execute jobs. Furthermore, Santy et al. demonstrated that summing bounds on the response time of all tasks provides a bound on the time it will take for such an $f_n^{\checkmark}$ to occur after an overload finishes.

The second mechanism Santy et al. (2013) proposed applies to any system where job priorities are fixed. The mechanism to return the system to level ℓ works by tracking the actual schedule relative to a *reference schedule* in which all jobs run for their level-ℓ PETs. In order to do so, the system must simulate the reference schedule and compare the remaining execution for each job between the actual schedule and the reference schedule. Once all jobs have sufficiently short remaining execution to complete ahead of the reference schedule, all tasks with criticalities at least ℓ can execute jobs.

These mechanisms prevent low-criticality tasks from being permanently impacted by an overload. However, they do not allow these tasks to run at all for a period of time.

7.2 Scaling Separation Times of Low-Criticality Jobs Instead of Dropping Jobs

Su and Zhu (2013) proposed an alternative task model that allows for low-criticality tasks to have both a *desired* period and a *maximum* period. For a properly provisioned system, it is possible to guarantee that low-criticality tasks can execute with their maximum periods *even when high-criticality tasks run for their full PETs*, while executing tasks at or close to their desired periods in the expected case. This task model is called the *elastic mixed-criticality* (*E-MC*) task model. Unlike the similarly named model from Buttazzo et al. (2002), E-MC does not use an elasticity factor to determine the extent of scaling of each task.

In order to schedule E-MC task systems, Su and Zhu (2013) also proposed a modified version of the *earliest deadline first with virtual deadlines* (*EDF-VD*) scheduler (Baruah et al., 2012), called the *early-release EDF* (*ER-EDF*) scheduler. ER-EDF maintains a set of *wrapper-tasks* (Zhu and Aydin, 2009) that keep track of the slack that is created when high-criticality jobs finish ahead of their high-level PETs. Each low-criticality job is guaranteed to release no later than its task's maximum period after the release of its predecessor. However, such a job also has a set of *early release points*. Each time such a point arrives, if there is enough slack (as indicated by the wrapper-tasks) for the job to be released early, ER-EDF does so. In the common case, high-criticality jobs usually run for less than their high-level PETs, so low-criticality jobs run more frequently than their minimum guarantee. However, even during an overload, low-criticality jobs continue to receive a minimum level of service.

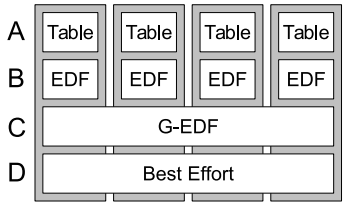


Fig. 16 Architecture of MC^2.

Su et al. (2013) later extended this work to multicore systems. The extension is basically a partitioned variant of ER-EDF. Su et al. considered partitioning the task system using several different partitioning heuristics. For high-criticality tasks, they used utilizations based on high-criticality PETs, and for low-criticality tasks, they used utilizations based on low-criticality PETs and maximum periods. A worst-fit decreasing heuristic based on those utilizations, ignoring criticalities, tended to perform the best and to significantly outperform the global EDF-VD algorithm (Li and Baruah, 2012).

Su et al. also considered two different techniques to reclaim slack. The simplest is to use the same strategy as ER-EDF, allowing low-criticality tasks to reclaim slack from high-criticality tasks on the same processor. They also considered a *global* slack reclamation technique. Under that technique, when there is not enough slack to release a job early on the core to which its task has been assigned, if there is enough slack on a remote processor, then that single job is migrated to the remote processor. Su et al. demonstrated that this technique can significantly improve the performance of their algorithm.

Jan et al. (2013) provided a different mechanism to minimize the separation time of low-criticality releases. They assumed that high-criticality jobs are statically prioritized over low-criticality jobs, and that the system optimistically schedules low-criticality jobs with deadlines that match their desired separation times. However, when a likely deadline miss is expected, the deadline is pushed back at that time. Jan et al.'s task model provides per-task parameters to specify how much deadline stretching is allowable, as well as which tasks to scale back first.

Overload and MC^2

Motivated by UAV systems, Herman et al. (2012) proposed a specific scheduler, the *mixed-criticality on multicore* (*MC^2*) scheduler, which supports four criticality levels, A through D. (An earlier version of MC^2 that supports five criticality levels was proposed by Mollison et al. (2010).) The architecture of MC^2 is depicted in Figure 16. Each criticality level is scheduled independently, and higher criticality levels are statically prioritized over lower criticality levels. Level A has HRT requirements. Tasks are partitioned onto CPUs and scheduled using a per-CPU ta-

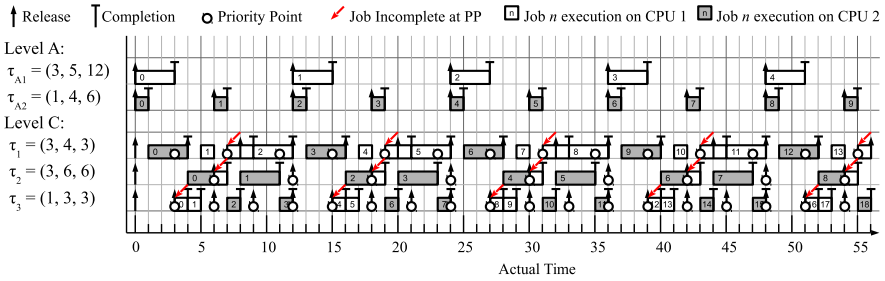
ble with a precomputed schedule. Level B also has HRT requirements and requires tasks to be partitioned onto CPUs, but uses P-EDF for scheduling. Level C has SRT requirements, and tasks are scheduled using G-EDF. Finally, level D is *best effort*, which means that it has no real-time guarantees. Level D can be scheduled using the general-purpose scheduler provided by the underlying operating system (OS).

Erickson (2014) and Erickson et al. (2015) considered the problem of overload within MC^2. In order to address scheduling in MC^2, Erickson (2014) added restricted supply to the analysis of GEL schedulers. The basic strategy for handling restricted supply is like that of Leontyev and Anderson (2010), but because Erickson does not use the full generality of window-constrained scheduling, the resulting bounds are tighter.

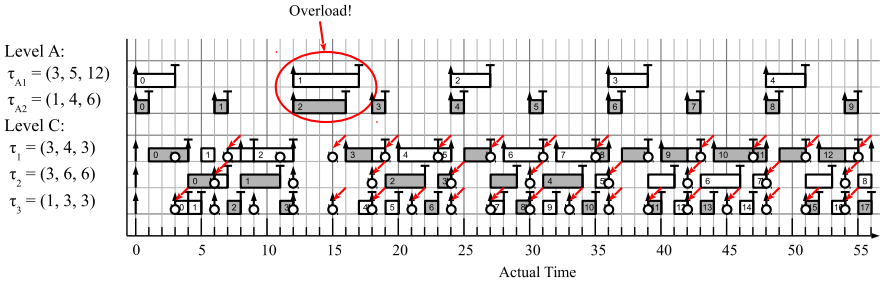
Because level-C PETs are not as pessimistic as level-A or -B PETs, it is possible that jobs at any level may overrun their level-C PETs. (MC^2 can optionally enforce job *budgets* to ensure that jobs do not overrun their PETs at their own criticality levels, but even if this feature is enabled, level-A and -B jobs can still overrun their level-C PETs.) The effects of overload are depicted in Figure 17, which depicts an MC^2 system that has only level-A and -C tasks. For this example, level-A tasks are depicted using the notation (C_i^C, C_i^A, T_i), where C_i^C is its level-C PET and C_i^A is its level-A PET, while level-C tasks are depicted using the notation (C_i^C, T_i, Y_i). Figure 17(a) depicts a schedule in the absence of overload, while Figure 17(b) depicts the results of some level-A jobs running for their full level-A PETs. As a result of the overload, all future job release times are impacted.

To analyze this situation, Erickson generalizes both the restricted supply model and the task model. He then describes a technique that can be used to recover from such an overload situation. His technique is depicted in Figure 17(c). He uses a notion of *virtual time*, as originally introduced by Zhang (1990) and used in uniprocessor real-time scheduling by Stoica et al. (1996). Essentially, there is a secondary "virtual" clock that, at actual time t, is operating at a speed of $s(t)$ relative to the actual clock. In the absence of overload, $s(t) = 1$, so that the two clocks operate at the same speed. However, after an overload occurs, the operating system can choose to use a slower speed, as occurs from actual time 19 to actual time 29 in Figure 17(c). Erickson's technique does not prescribe a particular choice of $s(t)$, but Erickson et al. (2015) provide experimental results that provide guidance. Additionally, Erickson et al. demonstrate that the system can recover relatively quickly under experimental conditions.

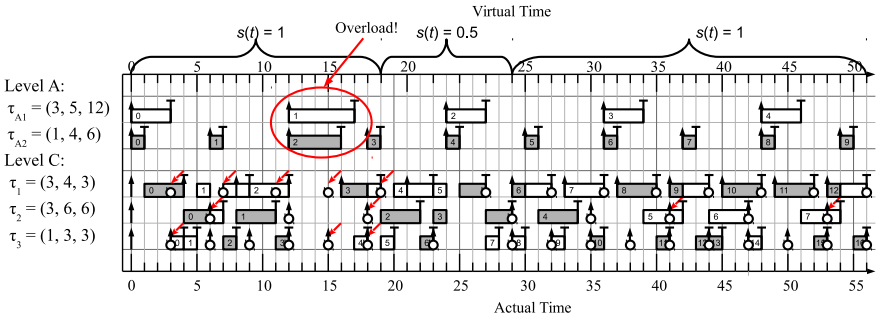
Job minimum separation times and relative PPs are defined in terms of the virtual clock, rather than the actual clock. This has the effect of reducing the number of level-C job releases for an interval of time and allows the system to recover from overload. The time required to do so is called a *dissipation time*. Erickson derives *dissipation bounds*, or upper bounds on the dissipation time.



(a) Example MC^2 schedule in the absence of overload, illustrating bounded response times.



(b) The same schedule in the presence of overload caused by level-A jobs starting at time 20 running for their full level-A PETs. Notice that response times of level-C jobs settle into a pattern that is degraded compared to (a). For example, consider $J_{2,6}$, which is released at actual time 36. In (a), it completes at actual time 43 for a response time of 7, but in this schedule it does not complete until actual time 46, for a response time of 10.



(c) The same schedule in the presence of overload and the recovery techniques from Erickson (2014). Notice that response times of level-C jobs settle into a pattern that is more like (a) than like (b).

Fig. 17 Example MC^2 task system, illustrating overload and recovery.

8 Summary

In this chapter, we reviewed prior work on SRT scheduling and overload. We discussed both prior SRT work using the bounded tardiness model and prior SRT work using other models of SRT. We then focused in more detail on prior work dealing with overload management, including those focusing on MC systems.

References

Anderson J, Srinivasan A (2004) Mixed pfair/erfair scheduling of asynchronous periodic tasks. Journal of Computer and System Sciences 68(1):157–204

Aydin H, Melhem R, Mosse D, Mejia-Alvarez P (2001) Optimal reward-based scheduling for periodic real-time tasks. IEEE Transactions on Computers 50(2):111–130

Baker T, Cirinei M, Bertogna M (2008) Edzl scheduling analysis. Real-Time Systems 40(3):264–289

Baruah S, Koren G, Mishra B, Raghunathan A, Rosier L, Shasha D (1991) Online scheduling in the presence of overload. In: Proceedings of the 32nd Annual Symposium On Foundations of Computer Science, pp 100–110

Baruah S, Cohen N, Plaxton C, Varvel D (1996) Proportionate progress: A notion of fairness in resource allocation. Algorithmica 15(6):600–625

Baruah S, Bonifaci V, D'Angelo G, Li H, Marchetti-Spaccamela A, Megow N, Stougie L (2010) Scheduling real-time mixed-criticality jobs. In: Hliněný P, Kučera A (eds) Mathematical Foundations of Computer Science 2010, Lecture Notes in Computer Science, vol 6281, pp 90–101

Baruah S, Bonifaci V, D'Angelo G, Li H, Marchetti-Spaccamela A, Van der Ster S, Stougie L (2012) The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In: Proceedings of the 24th Euromicro Conference on Real-Time Systems, pp 145–154

Bastoni A, Brandenburg B, Anderson J (2010) An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In: Proceedings of the 31st Real-Time Systems Symposium, pp 14–24

Bernat G, Burns A, Llamosi A (2001) Weakly hard real-time systems. IEEE Transactions on Computers 50(4):308–321

Block A (2008) Adaptive multiprocessor real-time systems. PhD thesis, The University of North Carolina at Chapel Hill

Bougueroua L, George L, Midonnet S (2007) Dealing with execution-overruns to improve the temporal robustness of real-time systems scheduled FP and EDF. In: Proceedings of the 2nd International Conference on Systems, pp 52–52

Brandenburg B (2011) Scheduling and locking in multiprocessor real-time operating systems. PhD thesis, The University of North Carolina at Chapel Hill

Burns A, Davis R (2017) Mixed criticality systems - a review. http://www-users.cs.york.ac.uk/ burns/review.pdf

Buttazzo G, Stankovic J (1995) Adding robustness in dynamic preemptive scheduling. In: Fussell D, Malek (eds) Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems, The Springer International Series in Engineering and Computer Science, vol 297, pp 67–88

Buttazzo G, Lipari G, Caccamo M, Abeni L (2002) Elastic scheduling for flexible workload management. IEEE Transactions on Computers 51(3):289–302

Chakraborty S, Kunzli S, Thiele L (2003) A general framework for analysing system properties in platform-based embedded system designs. In: Proceedings of the 2003 Design, Automation and Test in Europe Conference and Exhibition, pp 190–195

Cho H (2006) Utility accrual real-time scheduling and synchronization on single and multiprocessors: Models, algorithms, and tradeoffs. PhD thesis, Virginia Polytechnic Institute and State University

Clark R (1990) Scheduling dependent real-time activities. PhD thesis, Carnegie Mellon University

Compagnin D, Mezzetti E, Vardanega T (2014) Putting run into practice: Implementation and evaluation. In: Proceedings of the 26th Euromicro Conference on Real-Time Systems, pp 75–84

Devi U, Anderson J (2008) Tardiness bounds under global edf scheduling on a multiprocessor. Real-Time Systems 38(2):133–189

Erickson J (2014) Managing tardiness bounds and overload in soft real-time systems. PhD thesis, The University of North Carolina at Chapel Hill

Erickson J, Anderson J (2011) Response time bounds for G-EDF without intra-task precedence constraints. In: Proceedings of the 15th International Conference on Principles of Distributed Systems, pp 128–142

Erickson J, Anderson J (2012) Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling. In: Proceedings of the 24th Euromicro Conference on Real-Time Systems, pp 3–12

Erickson J, Anderson J, Ward B (2014) Fair lateness scheduling: reducing maximum lateness in G-EDF-like scheduling. Real-Time Systems 50(1):5–47

Erickson J, Kim N, Anderson J (2015) Recovering from overload in multicore mixed-criticality systems. In: Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium, pp 775–785

Funk S, Levin G, Sadowski C, Pye I, Brandt S (2011) Dp-fair: a unifying theory for optimal hard real-time multiprocessor scheduling. Real-Time Systems 47(5):389

Funk S, Berten V, Ho C, Goossens J (2012) A global optimal scheduling algorithm for multiprocessor low-power platforms. In: Proceedings of the 20th International Conference on Real-Time and Network Systems, pp 71–80

Garyali P (2010) On best-effort utility accrual real-time scheduling on multiprocessors. Master's thesis, The Virginia Polytechnic Institute and State University

Hamdaoui M, Ramanathan P (1995) A dynamic priority assignment technique for streams with (m, k)-firm deadlines. IEEE Transactions on Computers 44(12):1443–1451

Herman J, Kenna C, Mollison M, Anderson J, Johnson D (2012) Rtos support for multicore mixed-criticality systems. In: Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium, pp 197–208

Jan M, Zaourar L, Pitel M (2013) Maximizing the execution rate of low-criticality tasks in mixed criticality systems. In: Proceedings of the 1st Workshop on Mixed Criticality Systems, pp 43–48

Koren G, Shasha D (1994) MOCA: a multiprocessor on-line competitive algorithm for real-time system scheduling. Theoretical Computer Science 128(1–2):75–97

Koren G, Shasha D (1995a) Skip-over: Algorithms and complexity for overloaded systems that allow skips. In: Proceedings of the 16th IEEE Real-Time Systems Symposium, pp 110–117

Koren G, Shasha D (1995b) D^{over}: An optimal on-line scheduling algorithm for overloaded uniprocessor real-time systems. SIAM Journal on Computing 24(2):318–339

Lee S (1994) On-line multiprocessor scheduling algorithms for real-time tasks. In: Proceedings of IEEE Region 10's Ninth Annual International Conference, pp 607–611 vol.2

Leontyev H, Anderson J (2010) Generalized tardiness bounds for global multiprocessor scheduling. Real-Time Systems 44(1-3):26–71

Leontyev H, Chakraborty S, Anderson J (2011) Multiprocessor extensions to real-time calculus. Real-Time Systems 47(6):562–617

Li H, Baruah S (2012) Global mixed-criticality scheduling on multiprocessors. In: Proceedings of the 24th Euromicro Conference on Real-Time Systems, pp 166–175

Li P (2004) Utility accrual real-time scheduling: Models and algorithms. PhD thesis, Virginia Polytechnic Institute and State University

Li P, Wu H, Ravindran B, Jensen E (2006) A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints. IEEE Transactions on Computers 55(4):454–469

Lin K, Natarajan S (1988) Expressing and maintaining timing constraints in flex. In: Proceedings of the 9th IEEE Real-Time Systems Symposium, pp 96–105

Liu J, Lin K, Shih W, Yu A, Chung J, Zhao W (1991) Algorithms for scheduling imprecise computations. Computer 24(5):58–68

Liu R, Mills A, Anderson J (2014) Independence thresholds: Balancing tractability and practicality in soft real-time stochastic analysis. In: Proceedings of the 35th IEEE Real-Time Systems Symposium, pp 314–323

Locke C (1986) Best-effort decision making for real-time scheduling. PhD thesis, Carnegie Mellon University

Megel T, Sirdey R, David V (2010) Minimizing task preemptions and migrations in multiprocessor optimal real-time schedules. In: Proceedings of the 31st IEEE Real-Time Systems Symposium, pp 37–46

Mills A, Anderson J (2011) A multiprocessor server-based scheduler for soft real-time tasks with stochastic execution demand. In: Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp 207–217

Mollison M, Erickson J, Anderson J, Baruah S, Scoredos J (2010) Mixed-criticality real-time scheduling for multicore systems. In: Proceedings of the IEEE International Conference on Embedded Software and Systems, pp 1864–1871

Nelissen G, Berten V, Nelis V, Goossens J, Milojevic D (2012a) U-edf: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In: Proceedings of the 24th Euromicro Conference on Real-Time Systems, pp 13–23

Nelissen G, Funk S, Goossens J (2012b) Reducing preemptions and migrations in ekg. In: Proceedings of the 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp 134–143

Nelissen G, Su H, Guo Y, Zhu D, Nélis V, Goossens J (2014) An optimal boundary fair scheduling. Real-Time Systems 50(4):456–508

Regnier P, Lima G, Massa E, Levin G, Brandt S (2011) Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In: Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS), pp 104–115

Santy F, George L, Thierry P, Goossens J (2012) Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp. In: Proceedings of the 24th Euromicro Conference on Real-Time Systems, pp 155–165

Santy F, Raravi G, Nelissen G, Nelis V, Kumar P, Goossens J, Tovar E (2013) Two protocols to reduce the criticality level of multiprocessor mixed-criticality systems. In: Proceedings of the 21st International Conference on Real-Time Networks and Systems, pp 183–192

Spuri M, Buttazzo G, Sensini F (1995) Robust aperiodic scheduling under dynamic priority systems. In: Proceedings of the 16th IEEE Real-Time Systems Symposium, pp 210–219

Stoica I, Abdel-Wahab H, Jeffay K, Baruah S, Gehrke J, Plaxton C (1996) A proportional share resource allocation algorithm for real-time, time-shared systems. In: Proceedings of the 17th IEEE Real-Time Systems Symposium, pp 288–299

Su H, Zhu D (2013) An elastic mixed-criticality task model and its scheduling algorithm. In: Proceedings of the 2013 Design, Automation Test in Europe Conference Exhibition, pp 147–152

Su H, Zhu D, Mosse D (2013) Scheduling algorithms for elastic mixed-criticality tasks in multicore systems. In: Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp 352–357

Valente P (2016) Using a lag-balance property to tighten tardiness bounds for global edf. Real-Time Systems 52(4):486–561

Vestal S (2007) Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In: Proceedings of the 28th IEEE Real-Time Systems Symposium, pp 239–243

West R, Poellabauer C (2000) Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In: Proceedings of the 21st IEEE Real-Time Systems Symposium, pp 239–248

Zhang L (1990) Virtual clock: A new traffic control algorithm for packet switching networks. In: Proceedings of the 5th ACM Symposium on Communications Architectures & Protocols, pp 19–29

Zhu D, Aydin H (2009) Reliability-aware energy management for periodic real-time tasks. IEEE Transactions on Computers 58(10):1382–1397

Zhu D, Qi X, Moss D, Melhem R (2011) An optimal boundary fair scheduling algorithm for multiprocessor real-time systems. Journal of Parallel and Distributed Computing 71(10):1411–1425