CHAPTER 2

Background

In this chapter, we discuss prior work on SRT scheduling and overload management. We first survey the work on bounded tardiness that forms the basis for the work in this dissertation in Section 2.1 and related overhead analysis in Section 2.2. We then briefly survey alternative models for SRT systems in Section 2.3. Then, we survey past work on overload management using a notion of “value functions” in Section 2.4. In Section 2.6 we briefly survey some techniques for overload management that work by modifying the rate at which work is performed. Finally, in Section 2.7, we survey overload management within mixed-criticality systems.

2.1 Prior Bounded Tardiness Work

As mentioned in Chapter 1, Devi and Anderson (2008) provided tardiness bounds for implicit-deadline sporadic task systems scheduled under G-EDF. 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 points in the analysis occur when all CPUs become simultaneously busy. 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 \( \tau_i \) is at most \( x + C_i \). The value of \( x \) they defined is as follows:

\[
x = \frac{C_{\text{sum}} - C_{\text{min}}}{m - U_{\text{sum}}}.
\]
where $C_{\text{sum}}$ is the sum of the $m - 1$ largest values of $C_i$, $C_{\text{min}}$ is the smallest value of $C_i$, and $U_{\text{sum}}$ 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 $\tau_i$ with $\tau_{i,k}$. When analyzing a job $\tau_{i,k}$ 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 $\tau_{i,k}$ 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) provided tardiness bounds for a class of “window-constrained” schedulers. Under such a scheduler, jobs are prioritized on the basis of a PP, and the system executes the eligible jobs with the earliest PPs. Furthermore, there must exist constants $\phi_i$ and $\psi_i$ such that, if a job of $\tau_i$ has a release at time $r$, a deadline at time $d$, and a PP at time $y$ (priority), then it must be the case that $r - \phi_i \leq y \leq d + \psi_i$. By using the absolute deadline of each job as its PP, we see that G-EDF is a window-constrained scheduling algorithm.

In order to model restricted supply, Leontyev and Anderson defined a service function (following from Chakraborty et al. (2003)) $\beta_p(\Delta)$ for each CPU $p$, indicating that in any interval of length $\Delta$, 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 Figure 2.1. Each CPU $p$ has an available utilization $\hat{u}_p$ and a blackout time $\sigma_p$, so that

$$\beta_p(\Delta) \triangleq \max\{0, \hat{u}_p \cdot (\Delta - \sigma_p)\}.$$  

In Figure 2.1, we assume that the same pattern of supply restriction continues indefinitely; in this case, $\hat{u}_p = \frac{1}{2}$ and $\sigma_p = 3$. $\hat{u}_p$ indicates the long-term utilization of processor $p$, and in Figure 2.1, half of the CPU time is occupied by supply restriction. $\sigma_p$ 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 $\hat{u}_p$.

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.
2.2 Overhead and Locking Analysis

In order to determine the schedulability of a task system in practice, it is necessary to determine relevant system overheads and to account for them in the analysis. Brandenburg (2011) proposed methods to adjust task system parameters to account for overheads under G-EDF. These methods can be applied to arbitrary GEL schedulers. If a locking protocol is used, a similar process of adjusting parameters is generally necessary to account for similar effects that result from the operation of the locking protocol. In this section, we review these accounting methods. We will first consider overhead accounting under GEL schedulers in the absence of locking.

2.2.1 Overhead Analysis for GEL Schedulers Without Locking

In this section, we discuss overhead analysis for GEL schedulers without locking. Consider Figure 2.2, which depicts a subset of the schedule in Figure 1.10(a) with some additional overheads included.
1. From the time when an event triggering a release (e.g., a timer firing) occurs until the time that the corresponding interrupt is received by the OS, there is event latency, denoted \( ev \) (at time 18) in Figure 2.2.

2. When the interrupt is handled, the scheduler must perform release accounting and may assign the released job to a CPU. This delay is referred to as release overhead, denoted \( rel \) (after time 18) in Figure 2.2.

3. If the job is to be executed on a CPU other than the one that ran the scheduler, then an inter-
processor interrupt (IPI) must be sent. In this case, the job will be delayed by the IPI latency of the system, denoted \( ili \) (after time 18) in Figure 2.2.

4. The scheduler within the OS must run when the IPI arrives (before time 19), creating scheduling overhead, denoted \( sch \) (before time 19) in Figure 2.2.

5. After the scheduling decision is made, a context switch must be performed, creating context switch overhead, denoted \( cxs \) (at time 19) in Figure 2.2.

Observe from Figure 1.10(a) that \( \tau_{3,0} \) had previously been preempted by \( \tau_{1,2} \) at time 12. As a result of this earlier preemption, it experiences three additional costs when it is scheduled again after time 16.

1. When \( \tau_{3,0} \) is scheduled again (time 16), it incurs another scheduling overhead \( sch \) and context switch overhead \( cxs \).

2. Because \( \tau_{3,0} \) was preempted, some of its cached data items and instructions may have been evicted from caches by the time it is scheduled again. As a result, \( \tau_{3,0} \) will require extra execution time in order to repopulate caches. Although not depicted in Figure 2.2, observe from Figure 1.10(a) that \( \tau_{3,0} \) is migrated to another processor at time 21, which may cause even greater cache effects. The added time to repopulate caches is called cache-related preemption and migration delay (CPMD) and is denoted \( cpd \) (before time 17) in Figure 2.2.

Another overhead that occurs is the presence of interrupts, both from the periodic timer tick and from job releases. The maximum time for the timer tick interrupt service request routine is denoted
tck in Figure 2.2 (time 15), and the maximum cache-related delay from an interrupt is denoted cid in Figure 2.2 (after time 15).

Having discussed the nature of the overheads, we now give the precise expressions used by Brandenburg (2011) to perform overhead accounting. In these expressions, each overhead is represented as a superscript of Δ. For example, the event latency, denoted ev in Figure 2.2 as described above, is represented as Δev. Virtual tasks are defined for interrupt sources. For the timer interrupts, we let

\[ C_{tck}^0 = \Delta_{tck} + \Delta_{cid}, \] (2.1)

and we let \( T_{tck}^0 \) be the period of the timer tick. For each task \( \tau_i \), the corresponding release interrupts are handled by denoting

\[ C_{irq}^i = \Delta_{rel} + \Delta_{cid}, \] (2.2)
\[ T_{irq}^i = T_i. \] (2.3)

We let \( U_{tck}^0 = C_{tck}^0 / T_{tck}^0 \) and \( U_{irq}^i = C_{irq}^i / T_{irq}^i \) and define a parameter representing how long a CPU can be occupied by interrupts in the short term:

\[ c_{pre} = \frac{C_{tck}^0 + \Delta_{ev} \cdot U_{tck}^0 + \sum_{1 \leq j \leq n} \left( \Delta_{ev} \cdot U_{irq}^i + C_{irq}^j \right)}{1 - U_{tck}^0 - \sum_{1 \leq j \leq n} U_{irq}^j}. \] (2.4)

Each \( \tau_i \) (before overhead inflation) is replaced with \( \tau_i' \) (after overhead inflation). Each \( \tau_i \) in \( \tau \) has its \( C_i \) and \( T_i \) parameters redefined as follows, with the remaining parameters unchanged.

\[ C_i' = \frac{C_i + 2 \cdot (\Delta_{sch} + \Delta_{cxs}) + \Delta_{cpd}}{1 - U_{tck}^0 - \sum_{1 \leq j \leq n} U_{irq}^j} + 2 \cdot c_{pre} + \Delta_{ipi}, \] (2.5)
\[ T_i' = T_i - \Delta_{ev}. \] (2.6)

We sometimes dedicate one processor, called a release master, to handling interrupts. If we use a release master, then the only job that can be delayed by a release interrupt is that job itself.
Therefore, the following modifications are made.

\[
\begin{align*}
    e_{\text{pre}}' &= \frac{C_{0}^{\text{lock}} + \Delta^{\text{ev}}.U_{0}^{\text{lock}}}{1 - U_{0}^{\text{lock}}} + \Delta_{\text{ev}}.U_{0}^{\text{lock}}.U_{tck}^{0} - U_{tck}^{0}.U_{tck}^{0} - U_{tck}^{0} + 2 \cdot e_{\text{pre}} + \Delta_{\text{ipi}} + \Delta_{\text{rel}}. \\
    C_{i}' &= C_{i} + 2 \cdot (\Delta_{\text{sch}} + \Delta_{\text{css}}) + \Delta_{\text{cpd}} + 2 \cdot e_{\text{pre}} + \Delta_{\text{ipi}} + \Delta_{\text{rel}}.
\end{align*}
\]

(2.7)  

(2.8)

After these modifications are made to the task system, analysis that assumes the absence of overheads can then be applied. The soundness of this analysis is established in (Brandenburg, 2011), to which we refer the reader for details.

In the presence of a locking protocol, additional modifications to the task system are necessary. We next describe the modifications under the mutex queue spinlock protocol considered in Chapter 5.

2.2.2 Mutex Queue Spinlock Protocol Accounting

Under the mutex queue spinlock protocol, when a job requests a resource, it first spins non-preemptively until it acquires the resource. Then, it enters a critical section during which it uses the resource. It continues to execute non-preemptively until the critical section is over, at which time it releases the resource for other jobs to use.

In the presence of a locking protocol, Brandenburg (2011) describes two forms of blocking that can prevent a job from running, even if it has sufficiently high priority. When a task is spinning waiting for a resource (because another job is in a critical section using that resource), it incurs s-blocking. When a job cannot execute because it cannot preempt a lower-priority job that is running non-preemptively, it incurs pi-blocking. Both types of blocking must be bounded. Here we provide only the results from (Brandenburg, 2011), referring the reader to that work for full justification.

Brandenburg considered clustered scheduling, in which tasks are partitioned onto clusters of CPUs, and global scheduling is used within each cluster. The number of CPUs in cluster \( v \) is denoted \( c_{v} \), and the set of tasks in cluster \( v \) is denoted \( \theta_{v} \).

**Resource Model.** We consider a set \( \{\psi_{1}, \psi_{2}, \ldots, \psi_{n_{r}}\} \) of \( n_{r} \) resources. Each job can request only one resource at a time, but can request the same resource multiple times. We denote as \( N_{i,q} \) the maximum number of requests for \( \psi_{q} \) that any \( \tau_{i,k} \) can issue, and as \( L_{i,q} \) the maximum length of such a request.
Definitions Used in Analysis. We let $R_i$ denote a response-time bound for any job of $\tau_i$. Job response times can be affected by the amount of blocking a job experiences, even though $R_i$ must be used while analyzing blocking. As a result, blocking terms should be computed iteratively. Initially, each $R_i$ can be computed while ignoring blocking. The newly computed blocking can then be used to compute new values for $R_i$. This process is repeated until each $R_i$ no longer grows larger.

Brandenburg defines the request interference bound for a task $\tau_i$, with respect to resource $\psi_q$ and interval length $t$, as a set of $N_{i,q} \cdot \left\lceil \frac{t + R_i}{L_i} \right\rceil$ requests, each of length $L_{i,q}$. The request interference bound is denoted $tif(\tau_i, \psi_q, t)$. He also defines, for a request set $S$, $top(l,S)$ as the $l$ largest requests, and $total(l,S)$ as the sum of the lengths of all requests in $top(l,S)$. In addition, the aggregate interference bound of a set of tasks $\theta$, with respect to resource $\psi_q$, interval length $t$, and interference limit $l$, is the set of all requests in $top(l,tifs(\tau_i, \psi_q, t))$ for any $\tau_i \in \theta$. The aggregate interference bound is denoted $tifs(\theta, \psi_q, t, l)$.

S-Blocking Analysis. Brandenburg shows that the greatest s-blocking that an arbitrary $\tau_{i,k}$ can incur due to requests for $\psi_q$ issued by jobs of tasks assigned to cluster $v$ is

$$spin(\tau_i, v, \psi_q) = \begin{cases} 
    total(N_{i,q} \cdot c_v, tifs(\theta_v, \psi_q, R_i, N_{i,q})) & \text{if cluster } v \text{ is not } \tau_i \text{'s cluster,} \\
    total(N_{i,q} \cdot (c_v - 1), tifs(\theta_v \setminus \{ \tau_i \}, \psi_q, R_i, N_{i,q})) & \text{if cluster } v \text{ is } \tau_i \text{'s cluster.} 
\end{cases}$$

The total s-blocking that arbitrary $\tau_{i,k}$ can incur is computed by summing $spin(\tau_i, v, \psi_q)$ over all clusters and resources.

PI-Blocking Analysis. Recall that $\tau_{i,k}$ is “pi-blocked” if a lower-priority job is executing non-preemptively, preventing $\tau_{i,k}$ from beginning execution.

Brandenburg defines $lower(\tau_i)$ as the set of tasks within the same cluster as $\tau_i$ that could cause some $\tau_{i,k}$ to incur pi-blocking. For a fixed-priority scheduler, $lower(\tau_i)$ is simply the set of tasks that have lower priority than $\tau_i$. For a GEL scheduler, $lower(\tau_i)$ is the set of tasks $\tau_j$ with $Y_j > Y_i$. This is because if $\tau_{j,\ell}$ blocks $\tau_{i,k}$ at $r_{i,k}$, then $r_{j,\ell} < r_{i,k}$ (as $\tau_{j,\ell}$ is already running). Thus, if $Y_j \leq Y_i$, then by the definition of $Y_i$ in Section 1.3, $y_{j,\ell} \leq y_{i,k}$, and $\tau_{j,\ell}$ does not have lower priority.
A job executes non-preemptively only during an interval from the request of a resource until the completion of the corresponding critical section. Therefore, bounding the length of such an interval for any job of any task in $\text{lower}(\tau_i)$ is sufficient to bound the pi-blocking incurred by any job of $\tau_i$.

Let $\tau_{j,\ell}$ be an arbitrary job that causes $\tau_{i,k}$ to incur pi-blocking. Brandenburg demonstrates that the maximum amount of time that $\tau_{j,\ell}$ may be blocked waiting for resource $\psi_q$ due to jobs in cluster $v$ is at most

$$\text{spin}'(\tau_j, v, \psi_q) = \begin{cases} 
\text{total}(c_v, tifs(\theta_v, \psi_q, R_j, 1)) & \text{if cluster } v \text{ is not } \tau_j's \text{ cluster}, \\
\text{total}(c_v - 1, tifs(\theta_v \setminus \{\tau_i, \tau_j\}, \psi_q, R_j, 1)) & \text{if cluster } v \text{ is } \tau_j's \text{ cluster.}
\end{cases}$$

For a given resource, summing this expression over all clusters and adding $L_{j,q}$ yields the maximum time that $\tau_{j,\ell}$ can execute non-preemptively. Note that $\ell$ is arbitrary, so this bound can be considered a bound for $\tau_j$, not just for $\tau_{j,\ell}$. Selecting the maximum such bound for all resources and tasks in $\text{lower}(\tau_i)$, ignoring combinations of $\tau_j$ and $\psi_q$ such that $\tau_j$ does not use $\psi_q$, yields the maximum pi-blocking that $\tau_{i,k}$ can incur.

**Accounting Method.** In order to account for blocking, each task’s WCET should be increased by the sum of the maximum s-blocking it may incur and the maximum pi-blocking it can incur. Then, analysis methods that assume the absence of a locking protocol can be used.

We next discuss how to account for overheads related to this locking protocol.

### 2.2.3 Mutex Queue Spinlock Protocol Overheads

We now consider the overheads resulting from the mutex queue spinlock protocol. Brandenburg’s analysis assumes the presence of a release master.

When $\tau_{i,k}$ attempts to access $\psi_q$, execution proceeds as follows, and as depicted in Figure 2.3.

1. $\tau_{i,k}$ begins to execute non-preemptively by issuing a system call. The related overhead is denoted $sci$ in Figure 2.3.

2. $\tau_{i,k}$ uses the locking protocol and attempts to acquire $\psi_q$. If $\psi_q$ is held by another job, then $\tau_{i,k}$ must spin. The time spent spinning is accounted for using the techniques presented in Section 2.2.2, so we do not consider that time in overhead accounting. However, the cumulative
Figure 2.3: Illustration of overheads that occur in a spinlock protocol.

time spent running other parts of the locking protocol is denoted \textit{in} in Figure 2.3. Note that \textit{in} appears both before and after the spinning in Figure 2.3. The overhead term $\Delta^{in}$ would refer to the \textit{sum} of the two occurrences.

3. $\tau_{i,k}$ actually executes its critical section. Brandenburg’s analysis assumes that no overheads affect critical sections, which is why a release master is assumed.

4. $\tau_{i,k}$ uses the locking protocol to release $\psi_q$. The related overhead is denoted \textit{out} in Figure 2.3.

5. $\tau_{i,k}$ then begins to execute preemptively by issuing a system call, with the overhead denoted \textit{sco} in Figure 2.3.

Accounting for overheads within a spinlock protocol is a three-step process. The first step is to apply the overheads due to the spinlock protocol, without accounting for other overheads. Note that the overheads \textit{in} and \textit{out} may delay other jobs waiting to acquire $\psi_q$, causing them to spin for longer times. Therefore, we must add $\Delta^{in} + \Delta^{out}$ to each request length $L_{i,q}$, so that the blocking analysis from Section 2.2.2 correctly bounds the spinning of other tasks. Additionally, during the first step we also account for the extra execution that each $\tau_{i,k}$ experiences due to locking overheads. Each request produces $(\Delta^{sci} + \Delta^{sco} + \Delta^{in} + \Delta^{out})$ units of overhead, and the total number of requests can be computed by summing $N_{i,q}$ over all resources. We adjust $C_i$ accordingly.

During the second step, we apply the lock accounting from Section 2.2.2 to the modified task system, inflating the worst-case execution times of all tasks that use resources.

During the final step, we apply the locking-agnostic overhead accounting from Section 2.2.1 to obtain the final values of $C_i$ and $T_i$ for each task.
Recall that, as discussed in Section 2.2.2, the lock accounting requires a response-time bound $R_i$ for each $\tau_i$, and that the accounting may need to be performed iteratively if it increases $R_i$ for some $\tau_i$. When performing overhead-aware analysis, the second and final steps may need to be repeated iteratively if together they cause some $R_i$ to increase.

### 2.3 SRT Models

Although this dissertation primarily considers the “bounded lateness” definition of SRT, much past research has been performed describing other definitions. Except where noted, these papers consider uniprocessors 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 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. (Normal 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 $k = h + 1$, 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.)

\(^1\)We have changed their notation slightly to avoid conflict with other terms.
A weaker form of the \((h, k)\) model, the \emph{window-constrained} task model, was described by West and Poellabauer (2000). In that model, \(h\) jobs must meet their deadlines within periodic \emph{fixed} windows of \(k\) jobs of that task. (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 \emph{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 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). A more precise metric and an optimal scheduling algorithm with regard to that metric were proposed by Aydin et al. (2001).

Locke (1986) assigned to each task a \emph{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 2.4. 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 \(\tau\) is the task considered in Figure 2.4(a). If some \(\tau_{i,k}\) completes before the time marked “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 \(\tau_{i,k}\) if it is possible for \(\tau_{i,k}\) to complete before the critical time. Furthermore, suppose there are two tasks \(\tau_i\) and \(\tau_j\) that each have value functions of this form and that at time \(t\) there are ready jobs \(\tau_{i,k}\) and \(\tau_{j,\ell}\). If it is possible to complete either \(\tau_{i,k}\) or \(\tau_{j,\ell}\) before its respective critical time, but not to complete both before their respective critical times, then \(\tau_{i,k}\) should generally be selected if \(\tau_i\) has a higher constant value than \(\tau_j\), and \(\tau_{j,\ell}\) should be selected if
the reverse is true. (This is not strictly true because the choice of $\tau_{i,k}$ or $\tau_{j,\ell}$ 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 2.4(a) is labelled as the “critical time.” Although the step function discussed above is the simplest value function, Locke proposed others. Figure 2.4(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 2.4(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 2.4(b). Finally, as depicted in Figure 2.4(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.

Because value functions are used primarily for overload management, scheduling with value functions is discussed in the next section.

Buttazzo and Stankovic (1995) used a closely related notion of value. They divided tasks into “hard” and “critical” categories. Tasks of both types are expected to meet their deadlines when the...
system is not overloaded, and critical tasks are only allowed to miss deadlines when the system is overloaded even considering only critical tasks. Furthermore, a task can be accepted or rejected. Once a critical task has been accepted it is always guaranteed to complete by its deadline, even during severe overload. In addition, each task can have a deadline tolerance, which is an allowable amount of tardiness. Any job that will not complete within its task’s deadline tolerance should be rejected and not run at all. The scheduling algorithm they proposed is discussed in Section 2.5.

2.4 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.

2.4.1 Locke’s Best-Effort Heuristic

Locke (1986) considered a system that resembles the sporadic task model, scheduled globally on a multiprocessor. 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 times than generally expected.

As discussed in Section 2.3, the system should try to achieve the maximum cumulative value even if such 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 2.5, as scheduled on a uniprocessor. Suppose that \( \tau_1 \) releases \( \tau_{1,0} \) and \( \tau_2 \) releases \( \tau_{2,0} \) at time 0 and that no other job is released before time 15. Further suppose that \( \tau_{1,0} \) is known to require 14 ms of execution, while the job of \( \tau_{2,0} \) is known to require 7 ms of execution. This scenario is depicted in Figure 2.6(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 \( \tau_{1,0} \) as in Figure 2.6(b), in order to achieve a cumulative value of three, rather than selecting \( \tau_{2,0} \) as in Figure 2.6(c), which would only achieve a cumulative value of one.

Suppose, however, that \( \tau_3 \) actually releases \( \tau_{3,0} \) at time 8, and that \( \tau_{3,0} \) is known to require 6 ms of execution. Because \( \tau_{3,0} \) can complete with a value of 4, which is greater than the value that can be achieved by either \( \tau_{1,0} \) or \( \tau_{2,0} \), the system should execute \( \tau_{3,0} \) to maximize the cumulative value. If the system initially chose to execute \( \tau_{1,0} \), as depicted in Figure 2.6(d), then because \( \tau_{1,0} \) does not actually complete, the cumulative value achieved is only four. However, if the system initially chose to execute \( \tau_{2,0} \), as depicted in Figure 2.6(e), then because \( \tau_{1,0} \) completes, the cumulative value achieved is five. Therefore, the optimal choice at time 0 depends on whether \( \tau_{3,0} \) is released at time 8,
Figure 2.6: Several possible schedules for the task system with value functions depicted in Figure 2.5. 
\( \tau_{1,0} \) is released at time 0 and is known to have an execution time of 14. \( \tau_{2,0} \) is also released at time 0 and is known to have an execution time of 7. In (d) and (e), \( \tau_{3,0} \) is released at time 8 and is known to have an execution time of 6.
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 latter.

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 2.4(a), a job’s deadline is simply its critical time. However, under any of the other types of value functions depicted in Figure 2.4, a job’s deadline is usually after its critical time. His system 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 \( \tau_{i,k} \) at time \( t \) is computed as follows. Let \( e_{r,i,k}(t) \) be the expected remaining execution time for \( \tau_{i,k} \) at time \( t \), conditioned on how long it has already executed. The expected value \( V(t) \) of \( \tau_{i,k} \) at time \( t \) is defined to be the value that \( \tau_{i,k} \) will accumulate if it completes at time \( t + e_{r,i,k}(t) \). The value density of \( \tau_{i,k} \) at time \( t \) is simply \( V(t)/e_{r,i,k}(t) \). For example, consider the schedule in Figure 2.6(d). At time \( t = 0 \), \( \tau_{1,0} \) 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 \( t = 8 \), \( \tau_{1,0} \) 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. This heuristic is actually illustrated in Figure 2.6(d). As discussed above, at time \( t = 0 \), the value density of \( \tau_{1,0} \) is 3/14. Similarly, the
value density of $\tau_{2,0}$ at $t = 0$ is $1/(7 - 0) = 1/7 < 3/14$, so $\tau_{1,0}$ is selected for execution. At time $t = 8$, as discussed above, $\tau_{1,0}$ now has a value density of $1/2$. However, $\tau_{3,0}$ then has a value density of $4/(14 - 8) = 2/3 > 1/2$, so $\tau_{3,0}$ is selected in place of $\tau_{1,0}$.

Another detail in Locke’s heuristic relates to jobs with value functions of the form depicted in Figure 2.4(d), where completing a job too early can result in smaller achieved value. Because a job’s value is based only on its completion time, it is possible to start running a job long before its critical time, as long as its execution is discontinued when it approaches completion. However, because job execution times are determined stochastically, there is a risk that the job will complete earlier than expected. Locke’s system assumes that a job will not complete if it has executed for less than some user-specified number of standard deviations below its expected execution time. After it has executed for this amount of time, an incomplete job will be suspended until closer to its desired completion time.

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 possible achieved value.

2.4.2 $D^*$

Unless otherwise noted, all papers discussed in the remainder of Section 2.4 consider only step value functions, as depicted in Figure 2.4(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.\(^2\)

\(^2\)Several of the papers cited herein use $k$ for the importance ratio, but we use $q$ (quotient) to avoid conflict with the job index $k$. 
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 τi,k reaches a zero-laxity state. If no other job is in a zero-laxity state when this occurs, then τi,k runs immediately. To handle the case when some τj,ℓ 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 τi,k has a value greater than this sum plus the value of τj,ℓ, then τi,k preempts τj,ℓ, τj,ℓ is abandoned, and the sum is updated. Otherwise, τi,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 prove 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.

2.4.3 D\textsuperscript{over}

Koren and Shasha (1995b) provided a scheduler D\textsuperscript{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\textsuperscript{over}, like the design of D*, is based on EDF and is identical to EDF until an overload occurs.

Even during underload, D\textsuperscript{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 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 $\tau_{j,\ell}$. When waiting $\tau_{i,k}$ reaches a zero-laxity state, its value is compared to $(1 - \sqrt{q}) \cdot (\sum_{\in\Theta} V_{j,\ell})$, where $\Theta$ is the set containing all privileged jobs and the currently running job. If its value is larger than this expression, then $\tau_{i,k}$ preempts the currently running job, and all privileged jobs become waiting jobs. Otherwise, $\tau_{i,k}$ is discarded.

Koren and Shasha (1995b) also demonstrated that $D^{\text{over}}$ achieves the optimal competitive ratio of $\frac{1}{(1+\sqrt{q})^2}$. In other words, $D^{\text{over}}$ 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.

### 2.4.4 MOCA

While $D^*$ and $D^{\text{over}}$ 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 2.7. $\psi$ of the bands are designated with specific value densities, and $\omega$ 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 $\tau_{i,k}$ reaches a zero-laxity state, the system tries to schedule it on a risky processor. It first looks for an idle risky processor, considering first that in the band designated for $\tau_{i,k}$'s value density, followed by those in bands designated for lower value densities in decreasing order, then by those in the central pool. If it finds an idle risky processor, it begins executing $\tau_{i,k}$ there. Otherwise, it considers the same set of risky processors as before and finds the one running the $\tau_{j,\ell}$ with the earliest deadline. If $\tau_{i,k}$ has a later deadline than $\tau_{j,\ell}$, then $\tau_{j,\ell}$ is abandoned, and $\tau_{i,k}$ begins running in its place. Otherwise, $\tau_{i,k}$ is abandoned. This heuristic is 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

\[
\frac{1}{1 + 2m \cdot \left( \max_{1 \leq j \leq \psi} \frac{q^{\frac{j}{\psi}}}{\omega + \frac{q^{\frac{j}{\psi} - 1}}{q^{\frac{1}{\psi} - 1}}} \right)}.
\]

\( \psi \) 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.

### 2.4.5 Schedulers Accounting for Dependencies

Some work has been performed on scheduling with value functions in the presence of dependencies such as shared resources (Cho, 2006; Clark, 1990; Garyali, 2010; Li, 2004; Li et al., 2006). Such dependencies are not considered in our work on overload management, so we do not review most of these works in detail. However, Garyali (2010) proposed two multiprocessor heuristics that may be useful even without such dependencies, so we discuss his work as a representative example.

Both of Garyali’s heuristics assume that each task has a parameter specifying the expected execution time of each of its jobs and another parameter specifying the exact period between releases. In order to handle dependencies and shared resources, both heuristics maintain a directed acyclic graph (DAG) representing dependencies among jobs.

An example DAG is depicted in Figure 2.8. Resource \( i \) is identified as \( R_i \). In this example, \( \tau_{1,0} \) is currently holding the lock for \( R_1 \), and \( \tau_{3,2}, \tau_{5,0}, \) and \( \tau_{6,1} \) are waiting to acquire that lock. \( \tau_{2,3} \) is not holding any lock that has been requested by any other job. \( \tau_{4,1} \) holds the lock for \( R_2 \). \( \tau_{7,2} \) is waiting to acquire the lock for \( R_2 \), but holds the lock for \( R_3 \). Finally, \( \tau_{8,2} \) is waiting to acquire the lock for \( R_3 \).

Whenever a job requests, acquires, or releases a resource, the DAG is updated. If a cycle is created, causing the “acyclic” property of the graph to be violated, then a deadlock has occurred.
Such a situation implies that the jobs involved in the cycle cannot make progress. Both heuristics proposed by Garyali choose to abort one of the involved jobs in order to resolve the deadlock.

If a job has no incoming edges in the DAG, then it already holds any resources it requires to run. Such a job is called a zero-in-degree job. In Figure 2.8, $\tau_{1,0}$, $\tau_{2,3}$, and $\tau_{4,1}$ are zero-in-degree jobs. Both of Garyali’s heuristics consider only zero-in-degree jobs for scheduling, as other jobs cannot perform useful work until they acquire their requested locks.

Furthermore, both heuristics use a notion of global value density (GVD), similar to the notion of “value density” used by Locke (1986). Recall that the “value density” of $\tau_{i,k}$ at time $t$ is the value $\tau_{i,k}$ is expected to accrue if it runs continuously until its expected remaining execution is exhausted, divided by its expected remaining execution $e_{i,k}^r(t)$. Garyali’s heuristics also take into account the value achieved by completing jobs waiting on $\tau_{i,k}$—the descendants of $\tau_{i,k}$ within the DAG. Let $V_{j,\ell}$ be the value achieved if $V_{j,\ell}$ completes before its deadline, and $\Theta$ be the set containing $\tau_{i,k}$ and all of $\tau_{i,k}$’s descendants. The GVD of $\tau_{i,k}$ at $t$ is $\frac{\sum_{\tau_{j,\ell}\in\Theta} V_{j,\ell}}{\sum_{\tau_{j,\ell}\in\Theta} e_{j,\ell}^r(t)}$.

The first heuristic Garyali proposed is the non-greedy global utility accrual (NG-GUA) algorithm. NG-GUA attempts to create a G-EDF schedule. The deadline used for scheduling $\tau_{i,k}$ is actually the earliest deadline of $\tau_{i,k}$ itself or any descendent in the DAG, thus implementing the priority inheritance protocol (PIP) (Sha et al., 1990). Each time a scheduling decision must be made, NG-GUA simulates a G-EDF schedule of all zero-in-degree jobs. If no deadline is missed in this schedule, then NG-GUA simply makes scheduling decisions according to G-EDF. On the other hand,

---

3 Garyali (2010) referred to jobs as “phases.”

4 Garyali (2010) used the term “utility” for the same concept as “value” in (Locke, 1986).
if some deadline is missed in that schedule, NG-GUA removes jobs from consideration in order of increasing GVD until the simulated schedule no longer misses deadlines. Jobs are prioritized from the resulting schedule.

The second heuristic Garyali proposed is the greedy global utility accrual (G-GUA) algorithm. G-GUA is "greedy" in the sense that it tries to optimize accrued value even in the absence of expected deadline misses. G-GUA works by partitioning jobs (not tasks) onto CPUs and simulating an EDF schedule (with the PIP) on each CPU. Jobs are considered in decreasing GVD order. If a job cannot be scheduled on any CPU without causing an expected deadline miss, the partitioning heuristic continues to the next job. Jobs are prioritized based on the resulting schedule. Every time a new scheduling decision must be made, the partitioning heuristic is re-executed, so it is possible that a job will be migrated to a different CPU, thus G-GUA is truly a global algorithm.

Garyali demonstrated the effectiveness of his heuristics using a real in-kernel implementation. He showed that both heuristics significantly outperform traditional approaches like G-EDF when overloads are present, and that G-GUA can often accrue more utility at the expense of unnecessary deadline misses in the underload case.

### 2.5 Rate-Based Earliest Deadline Scheduling

Buttazzo and Stankovic (1995) proposed the robust earliest-deadline (RED) scheduler that uses a similar model to value functions. Each task has an associated deadline, value, and deadline tolerance. If a job completes within its deadline plus its deadline tolerance, then it achieves its value; otherwise, it achieves no value. 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<sup>ever</sup>, 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 \( \tau_{i,k} \) is released, RED uses the list to determine whether adding \( \tau_{i,k} \) will cause a deadline miss. If it will not, \( \tau_{i,k} \) is immediately accepted. Otherwise, RED will attempt to find one or more hard jobs that can be dropped. Buttazzo and Stankovic (1995) considered two different techniques to select such jobs, one that identifies the single job of lowest value that can be dropped (if such a job exists), and one that may drop multiple jobs if \( \tau_{i,k} \) is critical. If either strategy succeeds in finding a set of hard jobs with total value less than that of \( \tau_{i,k} \), and which if dropped will ensure no deadline misses for other accepted jobs, then all jobs in that set are rejected. Otherwise, \( \tau_{i,k} \) is rejected. When RED rejects a job, it adds that job to a list of rejected jobs. If another job completes early, then RED will examine that list to determine whether it can then schedule a job it previously rejected.

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 hard 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 hard 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.

### 2.6 Overload Management by Changing Execution Rates

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 determined 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, earliest eligible virtual deadline first (EEVDF) (Stoica et al., 1996), uses a notion of virtual time similar to that used in Chapter 6 of this dissertation. Therefore, 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 2.9. The actual progression of time is graphed on the bottom axis. From time 0 to time 2, only \( \tau_1 \) is present in the system. Therefore, it receives all of the CPU time. At time 2, \( \tau_2 \) enters the system. Because \( \tau_1 \) has a weight of 4 and \( \tau_2 \) has a weight of 2, \( \tau_1 \) receives twice as much processor time as \( \tau_2 \). Until \( \tau_3 \) arrives at time 8, \( \tau_1 \) receives 2/3 of the processor time and \( \tau_2 \) 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 \( \tau_i \), the speed of virtual time at 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

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

For example, between time 0 and time 2 in Figure 2.9, only \( \tau_1 \) 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. 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
Figure 2.9: EEVDF schedule of a task system. $\tau_1$ has a weight of 4 and always issues requests of size 2 ms. $\tau_2$ has a weight of 2 and always issues requests of size 2 ms, although its second request issued at actual time 8 completes early. $\tau_3$ has a weight of 2 and always issues requests of size 1 ms.

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 $\tau_2$ does in Figure 2.9 for the request issued at actual time 8.

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 $r + \frac{a}{W}$. Alternatively, the task may exit the system at the time its next request would otherwise be eligible, as $\tau_1$ does in Figure 2.9 at actual time 16.$^5$

For example, the first request of $\tau_1$ in Figure 2.9 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 $\tau_1$ 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 delay between eligible virtual times is 0.5 ms in both cases, but the delay between eligible actual times is 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 $\tau_2$ 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

---

$^5$Stoica et al. (1996) provide more complex rules that allow a task to leave at other times, but we do not consider those here.
\[ d = r + \frac{\tau}{W} \]. For example, the first request of \( \tau_1 \) in Figure 2.9 has a virtual eligible time of 0 and a request size of 2 ms, so its 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 \( \tau_2 \) that completes at actual time 12, 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 actual time 2 in Figure 2.9, \( \tau_1 \) has a request with a virtual deadline of 1 and \( \tau_2 \) has a request with a virtual deadline of 1.5. Because \( \tau_1 \) has an earlier virtual deadline, its request runs for the requested 2 ms. The next request of \( \tau_1 \) does not have an eligible time until virtual time 1, or actual time 5, so \( \tau_2 \)’s request runs from actual time 4 to actual time 5. In Figure 2.9, deadline ties are broken by task index, so when \( \tau_1 \)’s third request becomes eligible at actual time 5, it preempts the executing request of \( \tau_2 \).

Observe in Figure 2.9 that \( \tau_3 \) and \( \tau_2 \) receive the same processor share, even though their request sizes differ. The request size of \( \tau_2 \) is always 2 ms (even though the full size may not be used) and the request size of \( \tau_3 \) is always 1. However, from actual time 8 onward (when \( \tau_3 \) enters the system), \( \tau_3 \) releases jobs twice as frequently as \( \tau_2 \), except for the shift in release time for \( \tau_2 \) caused by the early completion. This occurs because both tasks have the same weight.

### 2.7 Overload Management within Mixed-Criticality Scheduling

Recall that, as discussed in Section 1.2, each task in a mixed-criticality system has multiple PWCETs, and guarantees at level \( \ell \) are conditioned on all jobs running for at most their respective level-\( \ell \) PWCETs. However, it is possible that some task’s level-\( \ell \) PWCET 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 PWCET overrun by simply dropping all jobs of level-\( \ell \) tasks from that point forward. However, this is usually an unacceptable response. In Chapter 6 we discuss how to handle level-C PWCET overruns in MC\(^2\), and here we discuss prior work addressing this problem within other mixed-criticality schedulers. In Section 2.7.1, we consider methods that reduce the number of low-criticality jobs that
are dropped, and in Section 2.7.2, we discuss methods that scale minimum separation times as an alternative to dropping jobs.

### 2.7.1 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, when some job scheduled using this technique overruns its level-\(\ell\) PWCET, all jobs at levels \(\ell\) 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 \(\tau_{i,k}\) of \(\tau_i\) overruns its PWCET.

1. If some \(\tau_j\) has a lower criticality *but a higher priority* than \(\tau_i\), it is not necessary to drop jobs from \(\tau_i\). This follows from a property of the analysis.

2. It is possible to set an *allowance* for each such \(\tau_i\) and criticality level \(\ell\) below that of \(\tau_i\), so that if \(\tau_{i,k}\) exceeds its PWCET by less than that allowance, it is not necessary to drop jobs at level \(\ell\). 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 \(\tau_i\) 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. proposed applies to fixed-priority systems. In order to restore the system to level \(\ell\), the system keeps track of a series of times \(f_{i}^{\ell}\), ordered by decreasing task priority. \(f_{0}^{\ell}\) is the last completion time of a job that overran its level-\(\ell\) PWCET. For \(i > 0\), \(f_{i}^{\ell}\) is the earliest time not earlier than \(f_{i-1}^{\ell}\) such that there is no active job of \(\tau_i\). Once \(f_{n}^{\ell}\) has been detected, where \(n\) is the number of tasks, all tasks with criticalities at least \(\ell\) 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 \(f_{n}^{\ell}\) to occur after an overload finishes.
The second mechanism Santy et al. proposed applies to any system where job priorities are fixed. The mechanism to return the system to level $\ell$ works by tracking the actual schedule relative to a *reference schedule* in which all jobs run for their level-$\ell$ PWCETs. 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 $\ell$ 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.

### 2.7.2 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 PWCETs*, 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 PWCETs. 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 PWCETs, 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.

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 PWCETs, and for low-criticality tasks, they used utilizations based on low-criticality PWCETs 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 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. Furthermore, Jan et al. provided a mechanism where the system can specify a prioritization for which tasks to scale back first.

2.8 Summary

In this chapter, we reviewed prior work on SRT scheduling and overload. We discussed both prior SRT work using the bounded tardiness model that is closely related to the bounded lateness model used in this dissertation, and prior SRT work using other models of SRT. We then focused in more detail on prior work dealing with overload management, including that focusing on MC systems. In the following chapters, we will discuss our approach to SRT scheduling and overload management within MC², building on some of this prior work.