CHAPTER 1

Introduction

The goal of this dissertation is to improve the state of the art for soft real-time (SRT) multiprocessor scheduling by improving response times without significantly increasing runtime overheads, and by providing the ability to recover from overload situations when more work arrives than is expected. This work is motivated by next-generation unmanned aerial vehicles (UAVs), which will require SRT scheduling for advanced functionality such as mission planning. While prior work can guarantee bounds on response times, this dissertation provides methods to improve upon those bounds without incurring additional overheads, and provides methods to provide more fine-grained control of such bounds. Furthermore, in order to reduce the size, weight, and power requirements for these UAVs, it will be necessary to run hard real-time (HRT) safety-critical software on the same multicore system as SRT mission-critical software. When provisioning the SRT portion of the system, assumptions about execution time could be made that are insufficiently pessimistic. As a result, the system may become overloaded. This dissertation also provides a method to manage such a situation.

We begin this chapter by providing a general introduction to real-time systems. We then provide a description of the basic scheduling framework used by the UAVs we consider. We then state the thesis of this dissertation, describe its contributions, and provide an outline for the remainder of the dissertation.

1.1 Real-Time Systems

Almost any computer system must produce valid outputs in order to be considered a “correct” system. In a real-time system, the result must also 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 “HRT” 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, a task that adjusts flight surfaces on an aircraft has such a requirement, as a missed deadline could result in a 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 circumstance. This usually requires over-provisioning the system.

A system is defined to be “SRT” if it has less stringent requirements. In such a system, each job typically still has a deadline, but the system may be deemed correct even if some jobs miss their deadlines. For example, one type of SRT constraint would be the requirement that some fraction of all deadlines in the system be met. This type of correctness is 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 can allow a system to be more fully provisioned. After describing a task model next, we will describe the particular SRT criterion that we use.

1.1.1 Task Model and SRT Criterion

In this dissertation, we consider scheduling under the sporadic task model. In order to describe this model, we depict an example task running by itself in Figure 1.2. (The key for all figures in this chapter is given in Figure 1.1.) A task represents one process that is composed of a (potentially infinite) series of discrete jobs. When a new job is available for execution, we say that job is released by the task. However, because the task is a single process, if the job’s predecessor has not yet completed, then the new job must wait to actually begin execution. (This requirement is relaxed in Chapter 4 of this dissertation.)

The system is composed of a set \( \tau = \{\tau_1, \tau_2, \ldots, \tau_n\} \) of \( n \) tasks.

The worst-case execution time (WCET) of each task \( \tau_i \), denoted \( C_i \), is the maximum execution time for any of its jobs. In Figure 1.2, \( C_1 = 2 \) ms, so no job runs for over 2 ms. However, some jobs run for only 1 ms, as allowed by the model.

The minimum separation time of each task \( \tau_i \), denoted \( T_i \), is the minimum amount of time between two job releases. In Figure 1.2, \( T_1 = 3 \) ms, so job releases occur at least three units apart.
However, after the job at time 9 is released, no new job is released until time 14, as allowed by the model.

The absolute deadline of a job is the point in time by which that job should finish. In Figure 1.2, the absolute deadline of the first job is at time 2. As discussed above, the precise interpretation of “should finish” depends on whether the system is HRT or SRT. The relative deadline of each task \( \tau_i \), denoted \( D_i \), is the time between the release time and absolute deadline of each job of that task. In Figure 1.2, \( D_1 = 2 \) ms, so for example, the job released at time 9 has its absolute deadline at time 11.

For simplicity, within this chapter we will consider implicit-deadline task systems in which for each task \( \tau_i \), \( D_i = T_i \). However, all original work in the remainder of this dissertation is also applicable to arbitrary-deadline task sets that may violate this assumption. For the purpose of examples, we will often use the notation \( \tau_i = (C_i, T_i) \) for tasks within implicit-deadline task systems.

A final parameter of each task \( \tau_i \) 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.

Suppose a job is released at time \( r \), has an absolute deadline at time \( d \), and completes at time \( t \). Then, its response time is \( t - r \), its lateness is \( t - d \), and its tardiness is \( \max\{0, t - d\} \), as depicted in Figure 1.3. 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.

Figure 1.2: Example of a sporadic task.
With these definitions in place, we now specify the particular SRT criterion we use: 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 time (with response-time bounds) are equivalent 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 time. Much past work, e.g., (Devi and Anderson, 2008; Leontyev and Anderson, 2010), has used the bounded tardiness criterion for SRT. These criteria are useful because each guarantees that each task receives sufficient processor share in the long term. Furthermore, lateness bounds can indicate that jobs must finish before their deadlines, whereas tardiness bounds cannot. For the remainder of this dissertation, when “SRT” is used without qualification, the bounded lateness criterion is in use.

The practicality of the bounded lateness model does depend on having reasonably small lateness bounds, and smaller lateness bounds generally provide a practical improvement. Therefore, one area of focus of this dissertation will be choosing appropriate scheduling algorithms to minimize lateness bounds.

1.1.2 Scheduling Algorithms

We will now discuss several common classes of scheduling algorithms. We first define relevant terms. A task system is feasible if some scheduling algorithm can schedule it correctly. When considering HRT, a system is said to be scheduled “correctly” if no job misses its deadline. When considering SRT, a system is said to be scheduled “correctly” if it has bounded lateness. A scheduler is said to be optimal if it correctly schedules any feasible task system.
A common 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. EDF is an optimal uniprocessor algorithm for both HRT and SRT systems. In particular, EDF can schedule any implicit-deadline task system with $\sum_{i \in \tau} U_i \leq 1$. An example EDF schedule is depicted in Figure 1.4.

When considering multiprocessor systems, we denote the number of processors as $m$. 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 1.5 depicts a correct schedule for this task system on only two processors when all jobs are released as early as possible. Notice that in this schedule, jobs of $\tau_3$ 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 is depicted in Figure 1.6. Unfortunately, as can be seen in the figure, all jobs of $\tau_3$ miss their deadlines. This demonstrates that G-EDF is not optimal in a HRT sense. However, notice that no job of $\tau_3$
misses its deadline by more than 1 ms. In fact, Devi and Anderson (2008) demonstrated that G-EDF is in fact optimal in a SRT sense.

Schedulers that are optimal in a HRT sense for implicit-deadline sporadic task systems do exist, e.g., (Anderson and Srinivasan, 2004; Baruah et al., 1996; Megel et al., 2010; Regnier et al., 2011). However, all such schedulers either are difficult to implement in practice or cause jobs to frequently be preempted by other jobs or migrated between CPUs. Even the schedule in Figure 1.5, which is for a very simple task system, requires each of \( \tau_3 \)'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 1.5, each of \( \tau_3 \)'s jobs initially has a higher priority than the corresponding job of \( \tau_2 \), but only for 1 ms. This type of priority change, which can cause problems for locking protocols Brandenburg (2011), does not occur under G-EDF. Therefore, G-EDF remains a good choice for SRT systems, and we use G-EDF as the basis for the work in this dissertation.

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 \textit{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, the work in this dissertation is directly applicable.

\section*{1.2 Mixed Criticality and MC\textsuperscript{2}}

Sometimes different applications that will be run on the same physical machine have different requirements for timing correctness. For example, as discussed in the beginning of Section 1.1, some applications have HRT constraints (requiring all deadlines to be met), while others have SRT
constraints (where bounded lateness is acceptable). This sort of mixture of requirements will become increasingly relevant for future generations of UAVs, as more tasks that have traditionally been performed by humans are instead performed by software. For example, safety-critical software performing functions such as flight control continues to have 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. On a multicore system, this could be orders of magnitude more pessimistic than actual behavior (Mollison et al., 2010), resulting in a system that is unnecessarily underutilized.
Figure 1.7: Possible schedules for a uniprocessor mixed-criticality system with two criticality levels, A (high) and B (low), both with HRT requirements. Level-A $\tau_1$ has a minimum separation time of 4 ms, a level-A PWCET of 3 ms, and a level-B PWCET of 2 ms. Level-B $\tau_2$ has a minimum separation time of 8 ms, a level-A PWCET of 4 ms, and a level-B PWCET of 3 ms. $\tau_1$ is statically prioritized over $\tau_2$.

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 worst-case execution time (PWCET). For arbitrary level $\ell$, the system is considered to be correct at level-$\ell$ if all tasks with a criticality level at or above level $\ell$ are scheduled correctly, assuming that no job of any task exceeds its level-$\ell$ PWCET. An example is depicted in Figure 1.7 with two criticality levels, A (high) and B (low). Figure 1.7(a) depicts the worst-case behavior assuming that no job of any task exceeds its level-B PWCET, and Figure 1.7(b) depicts the worst-case behavior assuming that no job of any task exceeds its level-A PWCET. Observe that deadlines are only missed in Figure 1.7(b), that only $\tau_2$ (which is a level-B task) has jobs that miss their deadlines, and that this schedule involves jobs exceeding their level-B PWCET. The provided scheduling algorithm correctly schedules this task system.

Motivated by the same UAV system considered in this dissertation, Herman et al. (2012) proposed a specific scheduler, the multi-core mixed-criticality ($MC^2$) scheduler, that 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 1.8. 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 table 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.

In this dissertation, we focus on scheduling at level C.

1.3 Past SRT Work

We now briefly review the past work on SRT scheduling that is directly relevant to this dissertation. Further review and other work on SRT scheduling is discussed in Chapter 2.

As mentioned above, past work on bounded lateness has actually been stated in the form of the equivalent condition of bounded tardiness. The seminal work on bounded tardiness was that by Devi and Anderson (2008), who considered G-EDF scheduling. Devi and Anderson showed that the tardiness of any job of \(\tau_i\) is at most \(x + C_i\), where

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

\(C_{\text{sum}}\) 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 \(U_i\).
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. Window-constrained schedulers have a specific property that guarantees that each job will eventually become and remain the highest priority job in the system. Leontyev and Anderson also considered restricted supply, in which some processors are not fully available to the task system being scheduled. The scheduling of level C in MC^2 can be analyzed by using restricted supply to model execution at levels A and B.

The tardiness bounds provided by Leontyev and Anderson (2010) are significantly more complex than those provided by Devi and Anderson (2008), due to both of the generalizations provided. Therefore, we do not provide the specific expressions here.

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, bounded response time is equivalent to bounded lateness and bounded tardiness. Leontyev et al. provided a method to determine whether a given set of response-time bounds could be met.

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 \( \tau_i \) is \( D_i \) units of time after its release. Under a GEL scheduler, jobs are prioritized based on priority points (PPs) that may differ from absolute deadlines. In an analogous manner to G-EDF and 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 \( Y_i \) units of time 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 1.9. Figure 1.9(a) depicts G-EDF itself, where \( Y_i = D_i \) for all \( i \), and Figure 1.9(b) depicts a different GEL scheduler, the global fair lateness (G-FL) proposed in Chapter 3 of this dissertation.
Figure 1.9: Comparison of two GEL schedules of the same task system, with $\tau_1 = \tau_2 = (2, 4)$ and $\tau_3 = (8, 8)$.

1.4 Thesis Statement

The original bounds provided by Devi and Anderson (2008) were tighter for G-EDF than those provided by Leontyev and Anderson (2010), due to the increased generality considered by Leontyev and Anderson. However, further improvements are possible, and the model considered by Devi and Anderson requires implicit deadlines (each $D_i = T_i$) and does not immediately generalize to other GEL schedulers.

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.

As discussed in the last section, restricted supply analysis can be used to account for level-A and level-B work when considering level-C behavior in MC^2. In order to do so, accurate WCETs must be used in the analysis. Mollison et al. (2010) used the level-C PWCET for each level-A or -B task, in order to maximize the actual utilization of the system. However, because the PWCETs at
levels A and B are more pessimistic than those at level C, it is possible that level-A or -B tasks may sometimes exceed their level-C PWCETs. This overload compromises guarantees at level C.

In order to address these limitations, we will support the following thesis:

*G-EDF can be modified to support smaller lateness bounds than previous work allows, with more flexibility to specify desired lateness criteria. Furthermore, such modifications do not violate the assumptions required for multiprocessor locking protocols, and the modified scheduler is easier to implement and/or has lower overheads than known HRT-optimal schedulers. In addition, recovery from overloads caused by tasks in MC² overrunning their level-C PWCETs can be facilitated by modifying the scheduler to delay job releases dynamically.*

### 1.5 Contributions

We now describe our contributions in support of this thesis.

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

Yet smaller lateness bounds are possible by further modifying the scheduler. In this dissertation, we discuss two techniques to do so: allowing multiple jobs of the same task to run simultaneously, and splitting jobs into smaller subjobs.

In the context of MC², we generalize the restricted supply analysis from Leontyev and Anderson (2010) by accounting for level-C PWCET overruns at levels A, B, and C. We also provide a method to recover at runtime from such an overload.

We now discuss each contribution in more detail.

#### 1.5.1 Analysis of GEL Schedulers

In Chapter 3, we discuss improved analysis of GEL schedulers and propose methods to choose GEL schedulers to obtain the best lateness bounds. The basic strategy for our analysis of lateness bounds is essentially that from (Devi and Anderson, 2008), but we make several improvements.
As discussed above, Devi and Anderson define the tardiness bound for \( \tau_i \) as \( x + C_i \), with a single value of \( x \) for the entire task system. One fundamental change we make is to define a separate \( x_i \) for each \( \tau_i \). We also allow for relative PPs that differ from minimum separation times, which allows us to consider both arbitrary deadlines and arbitrary GEL schedulers.

The tardiness bound \( x + C_i \) from Devi and Anderson is equivalent to the response-time bound \( D_i + x + C_i \). In our analysis, \( Y_i \) replaces \( D_i \), so we derive response-time bounds of the form \( Y_i + x_i + C_i \).

Stated as lateness bounds, these are of the form \( Y_i + x_i + C_i - D_i \).

We define a term
\[
S_i(Y_i) = C_i \cdot \max \left\{ 0, 1 - \frac{Y_i}{T_i} \right\}.
\]
that accounts for the difference between \( Y_i \) and \( T_i \), and we use it to provide the following bound on \( x_i \).

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

(1.1)

Notice that \( x_i \) effectively appears on both sides of (1.1), so (1.1) cannot be used directly to compute \( x_i \). However, we show how to define a linear program in order to determine the smallest values of \( x_i \) that satisfy (1.1) for all \( i \). Furthermore, if each \( Y_i \) is treated as a variable rather than as a constant, we 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.

We also propose G-FL, the same scheduler that was depicted in Figure 1.9(b). Under G-FL, for each \( \tau_i \), for each \( \tau_i \)
\[
Y_i = D_i - \frac{m-1}{m} \cdot C_i.
\]

As can be seen in Figure 1.9, G-FL can provide better lateness than G-EDF. We also show that it provably provides the smallest possible maximum lateness bound, given our analysis.

### 1.5.2 Removing The Intra-Task Precedence Constraint

In Chapter 4, we propose 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. We refer to this as the \textit{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 us to derive 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, we must require 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.

1.5.3 Job Splitting

In Chapter 5, we propose another modification to the scheduler to improve lateness bounds. The
current lateness bounds 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 worst-case
execution time of 2 ms and a period of 4 ms could have its jobs split in half, resulting in a task with
a worst-case execution time 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 1.10. Figure 1.10(a) depicts an
example schedule in the absence of splitting. Notice that $\tau_{3,0}$ completes 4 ms late. Figure 1.10(b)
depicts the schedule where jobs of $\tau_3$ are split into two subjobs. $\tau_{i,j,k}$ is used to denote subjob $k$ of
$\tau_{i,j}$. Notice that $\tau_{3,0}$ 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 1.10(c), where $\tau_{3,0,0}$ runs for 8 ms instead of 7 ms, and
$\tau_{3,0,1}$ 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 potential
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.

Our 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.

In order to determine the impacts of job splitting on lateness bounds, it is necessary to use realistic measures of overhead. Therefore, we implemented G-FL with job splitting in LITMUSRT, a real-time extension to the Linux kernel developed at UNC, and measured relevant overheads. We used these overheads in lateness-bound computations and showed that significant reductions in lateness bounds are possible, even accounting for overheads and even in the presence of critical sections.

1.5.4 Handling Overload in MC^2

In Chapter 6, we consider the problem of overload within MC^2. In order to address scheduling in MC^2, we add restricted supply to our analysis of GEL schedulers. Our basic strategy for handling restricted supply is like that of Leontyev and Anderson (2010), but because we do not use the full generality of window-constrained scheduling, our bounds are tighter. Furthermore, we improve analysis under the case when most processors are not fully available, but have minimal supply restriction. This is the common case under MC^2, because most processors have both level-A and -B tasks, but the level-A and -B PWCETs for those tasks are highly pessimistic, resulting in a great deal of slack for level C.

Because level-C PWCETs are not as pessimistic as level-A or -B PWCETs, it is possible that jobs at any level may overrun their level-C PWCETs. (MC^2 can optionally enforce job budgets to ensure that jobs do not overrun their PWCETs at their own criticality levels, but even if this feature is enabled, level-A and -B jobs can still overrun their level-C PWCETs.) The effects of overload

\[\text{http://www.litmus-rt.org/}\]
are depicted in Figure 1.11, which depicts an MC² system that has only level-A and -C tasks. For this example, level-A tasks are depicted using the notation \((C_C^i, C_A^i, T_i)\), where \(C_C^i\) is its level-C PWCET and \(C_A^i\) is its level-A PWCET, while level-C tasks are depicted using the notation \((C_C^i, T_i, Y_i)\). Figure 1.11(a) depicts a schedule in the absence of overload, while Figure 1.11(b) depicts the results of some level-A jobs running for their full level-A PWCETs. As a result of the overload, all future job release times are impacted.

To analyze this situation, we generalize both the restricted supply model and the task model. We then describe a technique that can be used to recover from such an overload situation. Our technique is depicted in Figure 1.11(c). We use a notion of virtual time, as originally introduced by Zhang (1990) and used in uniprocessor real-time scheduling by Stoica et al. (1996). Essentially, we maintain 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 OS can choose to use a slower speed, as occurs from actual time 19 to actual time 29 in Figure 1.11(c). Our technique does not prescribe a particular choice of \(s(t)\), but we provide experimental results that provide guidance.

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. We derive dissipation bounds, or upper bounds on the dissipation time.

1.6 Organization

In Chapter 2, we discuss relevant background work in SRT scheduling and overload management. Then, in Chapter 3, we discuss our analysis of GEL schedulers, G-FL, and our linear programming techniques to compute and optimize lateness bounds. In Chapter 4, we discuss the impact of removing intra-task precedence constraints, and in Chapter 5 we discuss the impact of job splitting. Afterward, in Chapter 6, we discuss analysis that includes restricted supply, overload recovery, and dissipation bounds. Finally, in Chapter 7, we offer concluding thoughts and discuss future work.
Example MC\textsuperscript{2} schedule in the absence of overload, illustrating bounded response times.

The same schedule in the presence of overload caused by level-A jobs started at time 20 running for their full level-A PWCETs. Notice that response times of level-C jobs settle into a pattern that is degraded compared to (a). For example, consider \( \tau_{2,6} \), 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.

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

Figure 1.11: Example MC\textsuperscript{2} task system, illustrating overload and recovery.