Reducing Tardiness Under Global Scheduling by Splitting Jobs

Jeremy P. Erickson and James H. Anderson
The University of North Carolina at Chapel Hill

Abstract

Under current analysis, tardiness bounds applicable to global earliest-deadline-first scheduling and related policies depend on per-task worst-case execution times. By splitting job budgets to create subjobs with shorter periods and worst-case execution times, such bounds can be reduced to near zero for implicit-deadline sporadic task systems. However, doing so will result in more preemptions and could create problems for synchronization protocols. This paper analyzes this tradeoff between theory and practice by presenting an overhead-aware schedulability study pertaining to job splitting. In this study, real overhead data from a scheduler implementation in LITMUS$^{\mathrm{RT}}$ *was factored into schedulability analysis. This study shows that despite practical issues affecting job splitting, it can still yield substantial reductions in tardiness bounds.*

1 Introduction

A *tardiness bound* for a real-time task is an upper bound on the extent to which one of its jobs may complete beyond its deadline. A number of optimal scheduling algorithms exist that can ensure zero tardiness for implicit-deadline sporadic task systems in theory (e.g., [2, 12, 13]). However, such algorithms either cause jobs to experience frequent preemptions and migrations or are difficult to implement in practice. In contrast, a wide variety of global algorithms exist that are reasonable to implement and provide bounded tardiness while allowing full platform utilization [11]. Such algorithms are useful in supporting applications such as multimedia systems where limited tardiness is acceptable.

In prior work on tardiness bounds, the *global earliest deadline first (G-EDF)* scheduler and the improved *global fair lateness (G-FL)* scheduler have received considerable attention [6, 8, 9, 10]. G-FL is considered a *G-EDF-like (GEL)* scheduler because, like G-EDF, each job's priority is defined by a fixed point in time after its release. For each of the just-cited papers, the presented tardiness bounds approach zero as task worst-case execution times approach zero, even if task utilizations remain constant. Therefore, in a theoretical sense, these tardiness bounds could be made arbitrarily close to zero by splitting jobs so that they have arbitrarily small worst-case execution times. However, job splitting introduces some of the same overhead-related concerns that plague many optimal algorithms.

Focus of this paper. In this paper, we examine the practical benefits of applying job splitting to reduce tardiness bounds under GEL schedulers. We specifically focus on G-FL, as it has the lowest bounds among such schedulers. We seek to determine the extent to which job splitting can lower tardiness when relevant system overheads are considered. In addition to increasing overheads, job splitting can adversely affect locking protocols, due to problems associated with splitting critical sections. We address this issue as well.

Contributions. Much of this paper is devoted to explaining how to account for overheads in G-FL when job splitting is applied. We propose overhead accounting methods that augment prior work of Brandenburg, who presented methods applicable to G-EDF based on an implementation of G-EDF in LITMUS$^{\mathrm{RT}}$ [3]. We determined needed accounting modifications for our purposes by also implementing G-FL with job splitting in LITMUS$^{\mathrm{RT}}$ and by noting differences between the G-EDF and G-FL implementations. In describing these modifications, we initially assume that no locking protocol is utilized. In the latter part of the paper, we explain how such protocols affect both the implementation of job splitting and the earlier-presented locking-oblivious accounting methods. With the resulting analytical framework in place, we then present the results of an overhead-aware schedulability study that we conducted to assess the impacts of job splitting under G-FL. In this study, we utilized a new heuristic algorithm developed by us that determines how splitting should be done. This algorithm was found to often enable dramatic tardiness-bound reductions, even in cases where a locking protocol is used. Reductions in the range 25% to 80% were quite common.

The analytical results mentioned above are presented in Secs. 4-7, after first discussing needed background and related work in Secs. 2-3. Our overhead-aware schedulability study is presented in Sec. 8.

2 Task Model

We consider a system τ of n *implicit deadline* sporadic tasks $\tau_i = (T_i, C_i)$ running on $m \geq 2$ processors, where T_i is the minimum separation time between subsequent releases of jobs of τ_i, and $C_i \leq T_i$ is the worst-case execution time of any job of τ_i. We assume that the kernel enforces execution budgets, so that each job cannot run for more than C_i time units. The relative deadline of each job is assumed to be equal to its period. We use $U_i = C_i/T_i$ to denote the *utilization* of τ_i. All quantities are real-valued. We assume that

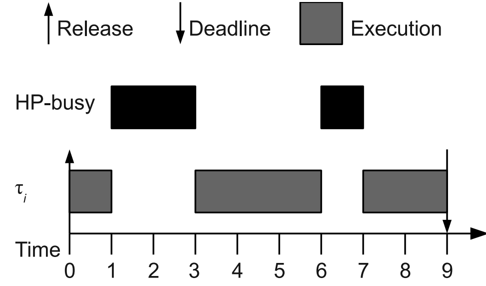
$$\sum_{\tau_i \in \tau} U_i \leq m, \tag{1}$$

which was demonstrated in [11] to be a necessary condition for the existence of tardiness bounds. We assume that $n > m$. If this is not the case, then each task can be assigned its own processor, and no job of each τ_i will complete more than C_i time units after its release.

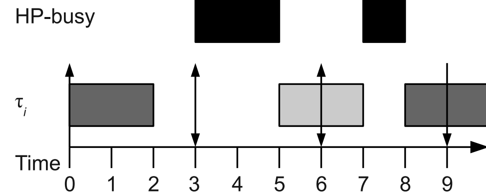
The focus of this work is the splitting of jobs into smaller subjobs with smaller periods and worst-case execution times. An example job of a task τ_i before splitting is depicted in Fig. 1(a), and the same job after a possible splitting is shown in Fig. 1(b). To distinguish between a task before splitting (as in Fig. 1(a)) and the same task after splitting (as in Fig. 1(b)), we define τ_i^{base} as the *base task* before splitting and τ_i^{split} as the *split task* after splitting. To disambiguate between base and split tasks, we also use superscripts on parameters: C^{base}, C^{split}, U^{base}, etc. A job of a base task is called a *base job*, while a split task is instead composed of *subjobs* of base jobs. We define the *split factor* of τ_i^{base}, denoted s_i, to be the number of subjobs per base job. In Fig. 1, $s_i = 3$. If a base job is denoted J_i, its subjobs will be denoted $\langle J_{i,0}, J_{i,1}, \ldots, J_{i,s_i-1} \rangle$. $J_{i,0}$ is its *initial subjob* — e.g., the first subjob in Fig. 1(b) — and J_{i,s_i-1} is its *final subjob* — e.g., the third subjob in Fig. 1(b). The longest time that any job of τ_i^{base} waits for or holds a single outermost lock is denoted b_i. Split tasks use a variant of the sporadic task model that is described in Sec. 7, but the sporadic task model is assumed prior to Sec. 7.

If a job has absolute deadline d and completes execution at time t, then its *lateness* is $t - d$, and its *tardiness* is $\max\{0, t - d\}$. If such a job was released at time r, then its *response time* is $t - r$. We bound these quantities on a per-task basis, i.e., for each τ_i, we consider upper bounds on these quantities that apply to all jobs of τ_i. The *max-lateness* bound for τ is the largest lateness bound for any $\tau_i \in \tau$. Similarly, the *max-tardiness* bound for τ is the largest tardiness bound for any $\tau_i \in \tau$.

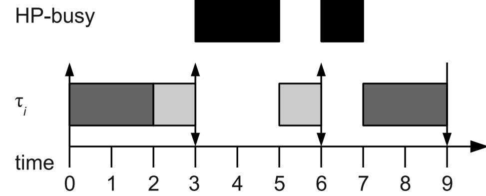
Let J_i be a job of task τ_i released at time r_i. A scheduler is *G-EDF-like (GEL)* if the priority of J_i is



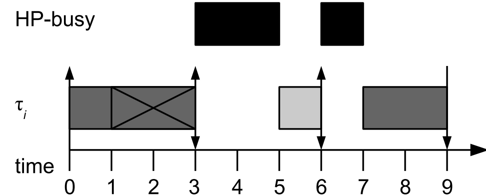
(a) No job splitting.



(b) Naïve job splitting.



(c) Job splitting with early releasing.



(d) Job splitting with early releasing and critical section support. Sections marked by "X" indicate that τ_i is waiting for or holding a lock.

Figure 1: Depiction of job splitting schemes for a single base job J_i of $\tau_i = (9, 6)$ with $s_i = 3$. During "HP-busy" instants, all CPUs are occupied by work with higher priority than τ_i. Alternating shades of gray represent alternating subjobs of J_i. $m = 2$ is assumed. For simplicity, the exact identity of higher-priority work is not depicted in this figure. Inset (c) is considered later in Sec. 4 and inset (d) in Sec. 7.

$r_i + Y_i$, where Y_i is constant across all jobs of τ_i. Y_i is referred to as the *relative priority point* (of the task) and $r_i + Y_i$ as the *absolute priority point* (of the job). G-EDF is the GEL scheduler with $Y_i = T_i$, and G-FL is the GEL scheduler with $Y_i = T_i - \frac{m-1}{m} C_i$.

When a non-final subjob completes, the resulting change in priority point is a *priority point move (PPM)*. In Fig. 1(b), PPMs occur at times 2 and 7.

3 Prior Work

In this section, we discuss prior work that we utilize. In Sec. 3.1, we discuss work relating to tardiness bounds, and in Sec. 3.2, we discuss work relating to overhead analysis.

3.1 Tardiness Bounds

In this subsection, we will briefly review relevant prior work for computing tardiness bounds. The purpose of this review is to show that prior tardiness bounds each approach zero as the maximum C_i in the system approaches zero. We will use the notation described in Sec. 2 rather than the original notation in the relevant papers. Tardiness bounds for G-EDF were originally considered in [6]. That work defines a value

$$x = \frac{\sum_{m-1 \text{ largest}} C_i - \min_{\tau_i \in \tau} C_i}{m - \sum_{m-2 \text{ largest}} U_i}$$

such that no task τ_i will have tardiness greater than $x + C_i$. An improved, but more complex, bound was introduced in [9]. While these works focused on G-EDF itself, [8] proposed G-FL as a new scheduler with analysis similar to G-EDF in [9]. G-FL usually provides a smaller maximum lateness bound for the task system. These improvements are based on analysis following the same basic proof structure as [6], and they maintain the property that all tardiness bounds approach zero as the maximum C_i in the system approaches zero.

3.2 Overhead 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. In [3], Brandenburg provides accounting methods to do so for several different schedulers, including G-EDF. His basic methodology is to transform the real task system τ into a *safe approximation* τ' such that a task of τ_i can have tardiness exceeding δ on a real system only if the corresponding τ'_i can have tardiness exceeding δ on an overhead-free system. Therefore, tardiness bounds produced for τ' while ignoring overheads also apply to τ running on the real system.

Due to space constraints, we only provide here a summary of the analysis provided in [3]. For complete explanations, please consult [3].

Because G-FL is a GEL scheduler, the analysis of G-EDF provided in [3] applies to G-FL. In Sec. 5, we consider how splitting jobs impacts this analysis. We emphasize that in this section, we are not providing any new analysis. Consider Fig. 2, which depicts a job J_1 of task τ_1 that is preempted by a job J_2 of task τ_2. J_1 experiences the following overheads.

1. From the time when an event triggering a release (e.g., a timer firing) occurs to the time that the corresponding interrupt is received by the kernel (time 0 to time 1), there is *event latency*, denoted Δ^{ev}.

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 Δ^{rel} (time 1 to time 2).

3. If the job is to be executed on a CPU other than the one that ran the scheduler, then an *interprocessor interrupt (IPI)* must be sent. In this case, the job will be delayed by the *IPI latency* of the system, denoted Δ^{ipi} (time 2 to time 3).

4. The scheduler within the kernel must run when the IPI arrives (time 3 to time 4), creating *scheduling overhead*, denoted Δ^{sch}.

5. After the scheduling decision is made, a context switch must be performed (4 to time 5). *Context switch overhead* is denoted Δ^{cxs}.

In Fig. 2, J_2 is released at time 6 and preempts J_1. This causes J_1 to experience two additional costs (time 12 to time 15).

1. When J_1 is scheduled again, it will incur another scheduling overhead Δ^{sch} and context switch overhead Δ^{cxs}.

2. Because J_1 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, J_1 will require extra execution time in order to repopulate caches. Although not depicted in Fig. 2, it is also possible that J_1 will be migrated to another processor, which could cause even greater cache effects. The added time to repopulate caches is called *cache-related preemption and migration delay (CPMD)* and is denoted Δ^{cpd}.

Although these additional costs are experienced by J_1, bounding the number of times that J_1 is preempted

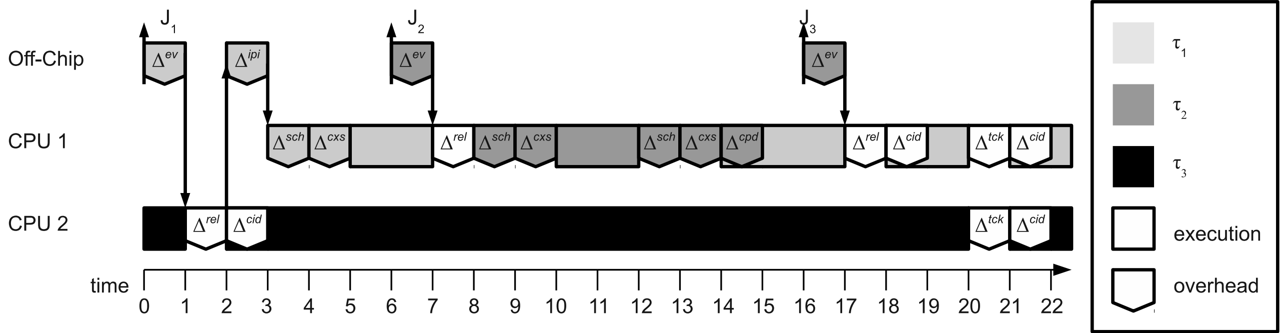


Figure 2: Example depicting the worst-case overheads for a job J_1 of task τ_1. τ_2's first job J_2 has a higher priority than J_1, while its second job J_3 has a lower priority. τ_3 has a single job (released before time 0) with priority higher than any other job in the system. Each overhead is shaded by the task it is charged to, with interrupt accounting colored white because it is handled differently.

usually results in too much pessimism. Fortunately, as shown in [3], we can instead charge these costs to J_2 and maintain a safe approximation.

Another cost that must be accounted for 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 Δ^{tck}, and the maximum cache-related delay from an interrupt is denoted Δ^{cid}. The period of the timer tick is denoted Q.

[3] defines virtual tasks for interrupt sources. For the timer interrupts, such a virtual task is defined by

$$C_0^{tck} = \Delta^{tck} + \Delta^{cid} \tag{2}$$

$$T_0^{tck} = Q. \tag{3}$$

For each task τ_i, the corresponding release interrupts are handled by denoting

$$C_i^{irq} = \Delta^{rel} + \Delta^{cid} \tag{4}$$

$$T_i^{irq} = T_i. \tag{5}$$

U_0^{tck} and U_i^{irq} are defined similarly. From this, a parameter is defined representing how long a CPU can be occupied by interrupts in the short term (see [3]):

$$c^{pre} = \frac{C_0^{tck} + \Delta^{ev} \cdot U_0^{tck} + \sum_{1 \le j \le n} \left(\Delta^{ev} \cdot U_j^{irq} + C_j^{irq} \right)}{1 - U_0^{tck} - \sum_{1 \le j \le n} U_j^{irq}}. \tag{6}$$

With these definitions, each τ_i' in τ' is defined as follows:

$$C_i' = \frac{C_i + 2 \cdot (\Delta^{sch} + \Delta^{cxs}) + \Delta^{cpd}}{1 - U_0^{tck} - \sum_{1 \le j \le n} U_j^{irq}} + 2 \cdot c^{pre} + \Delta^{ipi} \tag{7}$$

$$T_i' = T_i - \Delta^{ev}. \tag{8}$$

If we have one processor dedicated to handling interrupts, then as shown in [3], these formulas can be simplified somewhat.

4 Split G-FL Scheduling Algorithm

In this section, we describe the kernel mechanisms necessary to implement the splitting of jobs under G-FL. Although we will require the system designer to specify the split factor s_i for each job, we do not require the jobs to be split *a priori*. Instead, the kernel will use the budget tracking schemes described in this section to perform PPMs at the appropriate times. In order to aid the reader's understanding, we first describe schemes that ignore overheads and critical sections; we extend these schemes to account for overheads in Sec. 6 and to account for critical sections in Sec. 7. The usage of these budget tracking schemes is depicted in Fig. 1.

The tardiness analysis reviewed in Sec. 3.1 continues to hold if jobs become available for execution before their release times, as long as their priority points are based on release times that follow the minimum separation constraint. The technique of allowing jobs to run before their release times is called *early releasing* [1]. Allowing subjobs to be released early prevents tasks from suspending unnecessarily, as happens in Fig. 1(b) from time 2 to time 3, and may result in shorter response times. An improved schedule that includes early releasing is shown in Fig. 1(c). Because other work may have a higher priority than τ_i for earlier subjobs but not for later ones, such work may execute at a different time based on whether early releasing is used, as seen in insets (b) and (c) of Fig. 1.

In this section, let $C_i^{split} = C_i^{base}/s_i$ and $T_i^{split} = T_i^{base}/s_i$. For example, in Fig. 1(c), $C_i^{split} = 6/3 = 2$

and $T_i^{split} = 9/3 = 3$.

For the purposes of budget tracking, we define several functions with respect to time below. These functions are only defined for time t such that τ_i^{base} has a job that is ready for execution (released and predecessor has completed) but has not completed. We let $J_i(t)$ denote this job. For example, in Fig. 1(c), $J_i(t)$ denotes the same job of τ_i for any $t \in [0, 9)$. If the next job of τ_i is not released until time 10, then each function below will be undefined for $t \in [9, 10)$. Several of these functions are explicitly labelled as "ideal" functions that ignore critical sections — deviation from "ideal" behavior will be described in Sec. 7.

- The *current execution* $e_i(t)$ is the amount of execution that $J_i(t)$ has already completed before time t. In Fig. 1(c), $e_i(2.5) = 2.5$ and $e_i(5.5) = 3.5$.

- The *current release* $r_i(t)$ is the release time of $J_i(t)$. In Fig. 1(c), $r_i(t) = 0$ for all $t \in [0, 9)$.

- The *current offset* $o_i(t)$ is a parameter that will be used to properly handle overheads. In this section, we assume $o_i(t) = 0$.

- The *ideal subjob* $j_i(t)$ is the subjob of $J_i(t)$ that should be executing at time t, ignoring the effect of critical sections. It is defined as follows:

$$j_i(t) = \left\lfloor \frac{s_i \cdot (e_i(t) - o_i(t))}{C_i^{base}} \right\rfloor. \tag{9}$$

  In Fig. 1(c), $j_i(2.5) = 1$. (Recall that subjobs are zero-indexed.)

- The *ideal next PPM* $v_i(t)$ is the time for the next PPM after time t, ignoring the effect of critical sections and assuming that $J_i(t)$ is scheduled continously from time t until $v_i(t)$. It is defined as follows:

$$v_i(t) = t + (j_i(t) + 1)C_i^{base}/s_i + o_i(t) - e_i(t). \tag{10}$$

  In Fig. 1(c), $v_i(2.5) = 4$. Observe that the actual PPM does not occur until time 6 because τ_i is preempted.

- The *ideal subjob release* $\rho_i(t)$ is the release time for the current ideal subjob. It is defined as follows:

$$\rho_i(t) = r_i(t) + T_i^{split} j_i(t). \tag{11}$$

  In Fig. 1(c), $j_i(2.5) = 3$. Notice that, due to early releasing, the ideal subjob release may be after that subjob actually commences execution.

- The *ideal priority point* $y_i(t)$ is the priority point that should be active for $J_i(t)$ at time t, ignoring the effect of critical sections. It is defined as follows:

$$y_i(t) = \rho_i(t) + T_i^{split} - \frac{m-1}{m} C_i^{split}. \tag{12}$$

  In Fig. 1(c), $y_i(2.5) = 5$. (This occurs before the deadline of the ideal subjob because we are using G-FL rather than G-EDF.)

- The *current priority point* $\gamma_i(t)$ is the priority point that the scheduler actually uses for $J_i(t)$ at time t. This value is maintained by the budget tracking algorithm we describe in this section, rather than being merely a definition like the other functions in this section. Because there are no critical sections in Fig. 1(c) (as we are assuming in this section), $\gamma_i(t)$ equals $y_i(t)$ for all t. Therefore, $\gamma_i(2.5)$ is 5.

With these definitions in place, we define budget tracking rules in order to maintain the invariant $\gamma_i(t) = y_i(t)$.

- **R1.** When any job of τ_i^{base} is released at time t, $\gamma_i(t)$ is assigned to $y_i(t)$.

  In Fig. 1(c), applying this rule at time 0, we have $\gamma_i(0) = 2$.

- **R2.** Whenever a non-final subjob of τ_i^{base} is scheduled at time t to run on a CPU, a *PPM timer* is assigned to force a reschedule on that CPU at time $v_i(t)$. Whenever τ_i^{base} is preempted, the PPM timer is cancelled.

  In the schedule depicted in Fig. 1(c), the PPM timer is set at time 0 to fire at time 2. At time 2, it fires and forces a reschedule. Because τ_i^{base} is again selected for execution, the PPM timer is again set at time 2, this time to fire at time 4. At time 3, however, τ_i^{base} is preempted, so the timer is cancelled. At time 5, the timer is again set, this time to fire at time 6. Although it fires at time 6, only the final subjob remains, so the timer is not set again.

- **R3.** Whenever the scheduler is called on a CPU that was running τ_i^{base} at time t, if $y_i(t) > \gamma_i(t)$, then $\gamma_i(t)$ is assigned $y_i(t)$.

  In Fig. 1(c), the scheduler is called and was running τ_i^{base} at times 2, 3, 6, and 9. A PPM occurs at times 2 and 6, establishing $y_i(t) > \gamma_i(t)$ and causing $\gamma_i(t)$ to be updated, but at the other times, it is not the case that $y(t) > \gamma_i(t)$, so $\gamma_i(t)$ is not reassigned.

5 Overhead Analysis

We now consider how to extend the analysis provided in Sec. 3.2 for the case of jobs that are split. G-FL with job splitting is simply G-FL with the budget tracking rules described in Sec. 4. In addition, G-FL is simply G-EDF with different priority points. Therefore, the G-EDF overheads described in [3] will serve as a starting point for analyzing the overheads of G-FL with job splitting. In this section we continue to assume the absence of critical sections; critical sections will be handled in Sec. 7. An illustration of overheads due to job splitting is given in Fig. 3.

Due to space constraints, we merely summarize our needed modifications to Brandenburg's analysis in this section. A full description is presented in an online appendix [7]. Properly accounting for overheads will require us to inflate the *actual budget* used by the kernel differently than the *analytical budget* used for schedulability tests, and to define $o_i(t)$ for each task. Here we summarize the computation of the analytical budget, and in Sec. 6 we summarize how to modify the actual budget and to define $o_i(t)$.

Whenever a PPM is necessary by Rule R3 above, the scheduler can simulate a job completion for the old subjob (with the old PP), followed by an immediate arrival for the new subjob (with the new PP). This situation occurs in Fig. 1(c) at time 2, where the next subjob happens to be selected for execution, and at time 6, where the next subjob is not selected. Because the PPM timer handler is executed on the CPU that is already running the subjob to be ended, the handler can be accounted for as a normal job completion. The release of the new subjob will trigger the same code path as the case when a job completes after its successor has been released, so the time required to run the scheduler will not change.

Having considered the direct overheads produced by PPMs, we now consider other relevant overheads that happen while running the system. As a simple overhead accounting method, we can simply analyze split tasks rather than base tasks, treating subjobs as jobs. This method is actually more pessimistic than necessary. Unlike the release timer, it is not possible for the PPM timer to affect the behavior of other tasks in the system. In Fig. 1(c), a release interrupt will only occur at time 0, not at times 2 and 6. Similarly, in Fig. 3, a release interrupt only occurs between time 7 and time 9. Therefore, when computing T_i^{irq} in (5), we use base tasks rather than split tasks, so that splitting does not increase the overheads caused by interrupts.

In addition, each non-initial subjob becomes available immediately when its predecessor completes. Consider the behavior in Fig. 1(c) at time 2. Because the scheduler in LITMUS$^{\text{RT}}$ does not release the global scheduler lock between processing a job completion and the next arrival, if the new subjob has sufficient priority to execute it will run on the same CPU as its predecessor. There are two improvements that are made possible by this observation. First, only the initial subjob of each base job can experience event latency or require an IPI. Second, only the first subjob of each base job can preempt another job. These observations allow us to avoid charging every subjob for the related overheads, as described in [7], where we present detailed overhead accounting expressions that extend Brandenburg's to handle job splitting.

6 Budget Accounting

As described in Sec. 4 above, our splitting mechanism works by enforcing budgets for subjobs. In Sec. 5 above we considered how to account for overheads. As we mentioned in that section, care must be taken when performing budget enforcement while accounting for overheads. In order to do so, we must distinguish between the actual budget used by the kernel and the analytical budget used in schedulability tests, because some overheads are charged to different tasks than those their execution affects. For example, the Δ^{cpd} overhead in Fig. 3 from time 14 to time 15 is incurred by τ_1, but is charged to τ_2. How to properly enforce budgets while accounting for overheads has not been previously described to our knowledge.

Our goal with budget enforcement is to simulate the execution of shorter subjobs in place of longer base jobs. As described above, if the worst-case execution time of a base job is known, it can be split into subjobs in a straightforward manner. As can be seen in Fig. 3, most of the overheads considered above — Δ^{ev}, Δ^{rel}, Δ^{ipi}, Δ^{sch}, Δ^{cxs}, and the execution of interrupt handlers — are not part of a job itself, so they should contribute only to the analytical job budget. However, the cache-related delays Δ^{cpd} and Δ^{cid} do affect the runtime of jobs. For the purpose of the analytical budget, Δ^{cpd} is usually charged to the preempting job (except for a single charge at the beginning of each non-initial subjob), while Δ^{cid} is charged to the interrupt handler. However, in a real system these overheads would actually be incurred by the job that is preempted. Therefore, instead of assigning these overheads to the actual job budget ahead of time (as with the analytical budget), we wait until runtime to add each overhead to the budget for the job of τ_i^{base} *preempted* at time t using the $o_i(t)$ term, as described in [7]. For the special case of the Δ^{cpd} term charged to a non-initial subjob, we do add Δ^{cpd} to the actual budget of all subjobs except initial subjobs. (The precise mechanism to do so is described in [7].)

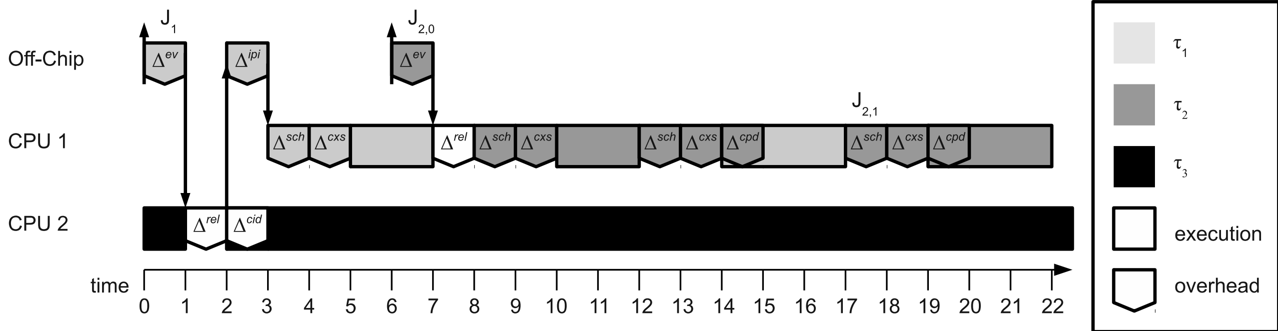


Figure 3: The same situation as Fig. 2, except that J_2 and J_3 (both from τ_2) have now been replaced with $J_{2,0}$ and $J_{2,1}$ to illustrate job splitting overheads. The execution time of J_1 has been shortened and the timer tick removed to simplify the figure.

Because G-FL is analyzed based on the analytical budgets, each relative priority point must be determined from the analytical budget, even though the kernel also needs to know the actual budget.

7 Handling Critical Sections

One of the advantages of G-FL promoted in [8] is that it is a *job-level static priority (JLSP)* algorithm, which is important for synchronization mechanisms such as those discussed in [3]. However, when splitting is introduced, the algorithm is no longer truly JLSP. If a subjob ends while waiting for or holding a lock, then the priority of the underlying job is changed, potentially violating the assumptions of synchronization algorithms. Fortunately, we can simply extend subjob budgets for as long as a resource request is active.

In order to support the necessary budget extensions, we use a more complicated set of rules than those described in Sec. 4. The behavior of our algorithm in the presence of critical sections is depicted in Fig. 1(d). These rules allow the budget for a subjob to be extended when its PPM is delayed. Furthermore, because this delay does not change the expressions for $j_i(t)$, $v_i(t)$, $\rho_i(t)$, or $y_i(t)$, the next subjob implicitly has its budget shortened. Essentially, we are only allowing each PPM to "lag" behind the ideal PPM by at most b_i units of the corresponding base job's execution. It is even possible for a subjob to be implicitly skipped by this mechanism if $s_i > C_i^{split}$.

- **R1.** When any job of τ_i^{base} is released at time t, $\gamma_i(t)$ is assigned to $y_i(t)$.

  This rule is identical to Rule R1 from Sec. 4.

- **R2.** Whenever a non-final subjob of τ_i^{base} is scheduled at time t to run on a CPU, a *PPM timer* is assigned to force a reschedule on that CPU at time $v_i(t)$. Whenever τ_i^{base} is preempted, or τ_i^{base} requests a resource, the PPM timer is cancelled.

  In the schedule in Fig. 1(d), the PPM timer is set at time 0 to fire at time 2, but is cancelled at time 1 when τ_i^{base} requests a resource. It will be set again at time 5 to fire at time 6, when it will actually fire. Because only a final subjob remains after time 6, however, the timer will not be set again.

- **R3.** Whenever a critical section ends, if $y_i(t) > \gamma_i(t)$, then a reschedule is forced.

  Observe in Fig. 1(d) that for $t \in [2, 3])$, the current subjob (according to priority) is an earlier subjob than $j_i(t)$. Thus, when the critical section ends, a PPM should occur. Triggering a reschedule will cause the needed PPM.

- **R4.** Whenever the scheduler is called on a CPU that was running τ_i^{base} at time t, if $y_i(t) > \gamma_i(t)$, then $\gamma_i(t)$ is assigned $y_i(t)$.

  This rule is identical to Rule R3 in Sec. 4 and functions the same way. However, the scheduler could have been invoked either due to Rule R2 or Rule R3. In Fig. 1(d) it is invoked due to Rule R3 at time 3 and due to Rule R2 at time 6.

We let C_i^{split} denote the ideal worst-case execution time of a subjob, ignoring critical sections. When we account for critical sections, a single subjob of a job from τ_i can run for as long as $C_i^{split} + b_i$. Nonetheless, τ_i's processor share over the long term is not affected, because the total execution of all subjobs must be the execution of the base job. In [7], we describe the necessary modifications in G-FL analysis to support the b_i term with no utilization loss.

8 Experiments

In order to test benefits splitting has on tardiness bounds on a real system, we used measurement data provided by Brandenburg at [4] for G-EDF on a four-chip 24-core system. On that system, pairs of cores share an L2 cache and six cores on a chip share an L3 cache. [3] reports that the best scheduler for bounded tardiness is *clustered earliest-deadline-first*, where CPUs are grouped by either L2 cache (*C-EDF-L2*) or L3 cache (*C-EDF-L3*). If a *release master (RM)* is used, then the first CPU is dedicated to handling interrupts. In such cases, we add "-RM" to the name of the scheduler, and the first cluster has one less CPU than the other clusters. Whether an RM is used or not, G-EDF is used on each cluster.

Similarly, we define the *clustered fair lateness (C-FL)* scheduler, where CPUs grouped by L2 cache (*C-FL-L2*) or L3 cache (*C-FL-L3*), with or without an RM. We assigned tasks to clusters using a worst-fit decreasing heuristic: we ordered tasks by decreasing utilization, and we placed each task in order on the CPU with the most remaining capacity.

Heuristic for Determining s_i. In order to use splitting to reduce tardiness bounds, it is necessary to determine appropriate s_i values for the tasks. To do so, we used a simple heuristic algorithm. A simple description follows.

- A task τ_i is *split-beneficial* if adding one to s_i results in a smaller maximum lateness bound for the entire task system.
- A task τ_i is *saturated* if adding one to s_i results in a system with unbounded tardiness.
- When trying to find a split-beneficial τ_i within a cluster, we first order tasks based on their contribution to the lateness bound. Because this ordering depends on the full algorithm for computing lateness bounds, we provide the details in [7]. We then loop through the tasks and stop upon finding a split-beneficial task τ_i. If we find such a split-beneficial task, we permanently increase its s_i by one. During the loop, we mark saturated tasks, and we skip tasks known to be saturated.
- To find a good splitting, we repeatedly try to find a split-beneficial task in the cluster with the maximum lateness bound. (The particular cluster that has the maximum lateness bound can change each time we find a split-beneficial task.) If there is no split-beneficial task in that cluster, then we attempt to find split-beneficial tasks in the remaining clusters in case doing so reduces system-wide locking overheads. When we do not find any split-beneficial task, we terminate the algorithm.

Task Set Generation. To determine the benefits of splitting, we generated implicit-deadline task sets based on the experimental design in [3]. We generated task utilizations using either a uniform, a bimodal, or an exponential distribution. For task sets with uniformly distributed utilizations, we used either a *light* distribution with values randomly chosen from $[0.001, 0.1]$, a *medium* distribution with values randomly chosen from $[0.1, 0.4]$, or a *heavy* distribution with values randomly chosen from $[0.5, 0.9]$. For tasks sets with bimodally distributed utilizations, values were chosen uniformly from either $[0.001, 0.5]$ or $[0.5, 0.9]$, with respective probabilities of 8/9 and 1/9 for *light* distributions, 6/9 and 3/9 for *medium* distributions, and 4/9 and 5/9 for *heavy* distributions. For task sets with exponentially distributed utilizations, we used exponential distributions with a mean of 0.10 for *light* distributions, 0.25 for *medium* distributions, and 0.50 for *heavy* distributions. Utilizations were drawn until one was generated between 0 and 1. We generated integral task periods using a uniform distribution from $[3ms, 33ms]$ for *short* periods, $[10ms, 100ms]$ for *moderate* periods, or $[50ms, 250ms]$ for *long* periods.

When testing the behavior with locking, critical sections were chosen uniformly from either $[1\mu s, 15\mu s]$ for *short* critical sections, $[1\mu s, 100\mu s]$ for *medium* critical sections, or $[5\mu s, 1280\mu s]$ for *long* critical sections. We denote the number of resources as n_r and performed tests with $n_r = 6$ and $n_r = 12$. We denote the probability that any given task accesses a given resource as p_{acc} and performed tests with $p_{acc} = 0.1$ and $p_{acc} = 0.25$. For a task using a given resource, we generated the number of accesses uniformly from the set $\{1, 2, 3, 4, 5\}$. These parameter choices are a subset of those used in [3] because, unlike [3], we chose to perform experiments on a larger variety of working set sizes to facilitate better comparisons to experiments without locking. An implementation study in [5] demonstrated that for typical soft real-time applications, the vast majority of critical sections are less than $10\mu s$. Therefore, the short critical section distribution is likely to be the most common in practice.

For each tested set of distribution parameters, we generated 100 task sets for each utilization cap of the form $\frac{24i}{20}$ where i is an integer in $[1, 20]$. Tasks were generated until one was created that would cause the system to exceed the utilization cap, which was discarded. We tested each task set with each cluster size, with and without an RM, and for tests involving locking with the clustered OMLP and mutex queue spinlock locking protocols (see [3]). We ignored on each scheduler task sets that were either not schedulable under C-FL (without splitting) or that resulted in zero tardiness, because our goal was to show improvements upon previously available schedulers. For each task set that

was schedulable, we applied task splitting using the algorithm described above and compared the maximum tardiness bound before and after splitting. (Because $s_i = 1$ is allowed by our algorithm, every considered task set is scheduable under C-FL with splitting by definition.)

Results. Examples of results without locking are depicted in Figs. 4 and 5, which have the same key. (Additional results can be found in [7]. In total, our experiments resulted in several hundred graphs.) Observe that improvements over 25% are common, and can be nearly 100% in some cases. Because task systems with higher working set sizes are more likely to be unschedulable even without splitting, higher working set sizes often represent significantly smaller groups of tasks and are skewed towards task sets with smaller utilization. This can cause a downward trend in the tardiness bounds with increased working set sizes for C-FL, but our purpose is to compare the effect of splitting when bounded tardiness is already achievable by C-FL.

Fig. 6 has the same key as Figs. 4 and 5, but depicts the difference in bounds with respect to system utilization cap rather than working set size. Observe that the bounds with splitting (dashed lines) tend to grow more slowly than the bounds without (solid lines) splitting until they grow drastically before all tested task sets were unschedulable. This phenomenon occurs because the overheads from splitting use some of the system's remaining utilization, and when very little utilization is available the tasks cannot be split as finely. An interesting result we observed in our study was that C-EDF-L3 and C-EDF-L3-RM generally had much larger s_i values (often on the order of 30) than C-EDF-L2 and C-EDF-L2-RM (where s_i values were usually on the order of 2 or 3), but the magnitude of the improvements was similar.

Figs. 7, 8, and 9 have the same key as each other and depict the behavior of the system in the presence of locks. The "Semaphore" tests use the clustered OMLP, and the "Spinlock" tests use mutex queue locks. The phenomenon of reducing bounds with increasing working set sizes is even more pronounced in these results, and the low schedulability of task sets with long critical sections produces strange trends in Fig. 9. Note that C-EDF-L2-RM with semaphores was not able to schedule most task systems we tested.

9 Conclusions

Tardiness bounds established previously for GEL schedulers can be lowered in theory by splitting jobs. However, such splitting can increase overheads and create problems for locking protocols. In this paper, we

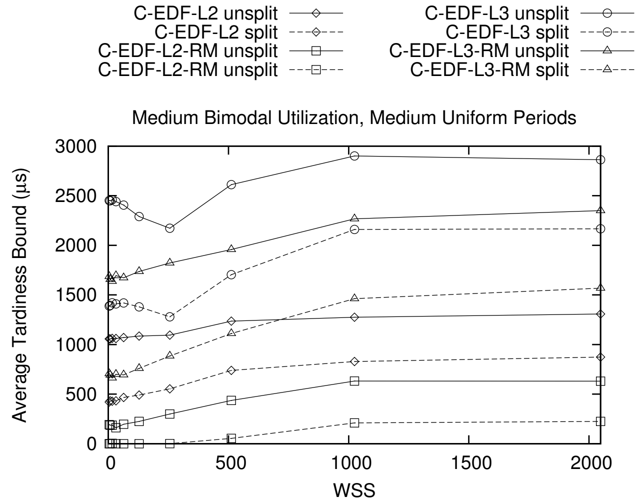


Figure 4: Light Uniform Utlization, Short Uniform Periods

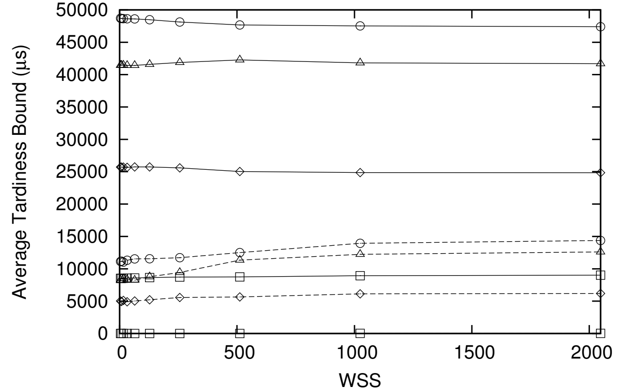


Figure 5: Medium Bimodal Utilization, Medium Uniform Periods

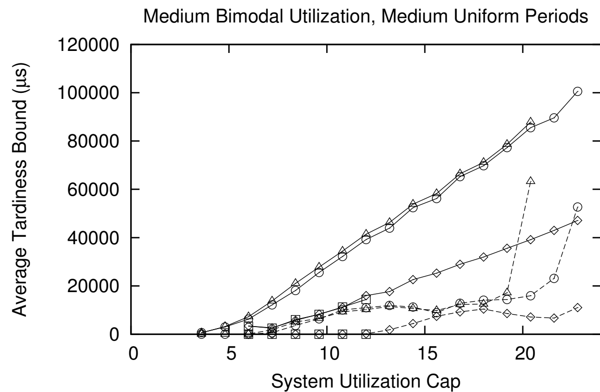


Figure 6: Medium Bimodal Utilization, Medium Uniform Periods, WSS = 128KB. Graph with respect to utilization instead of WSS.

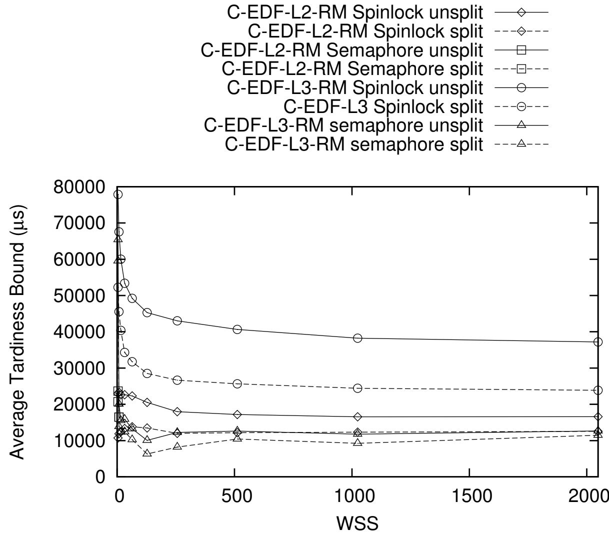


Figure 7: Medium Bimodal Utilization, Medium Uniform Periods, Short Critical Sections, $n_r = 6$, $p_{acc} = 0.25$

showed how to incorporate splitting-related costs into overhead analysis and how to address locking-related concerns. We then applied these results in a schedulablity study in which real measured overheads were considered. This study suggests that job splitting can viably lower tardiness bounds in practice.

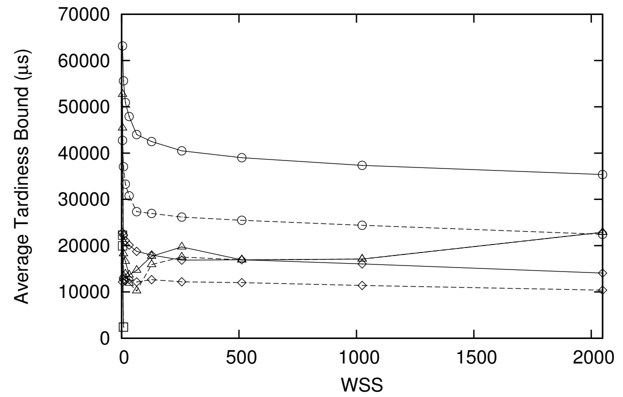


Figure 8: Medium Bimodal Utilization, Medium Uniform Periods, Medium Critical Sections, $n_r = 6$, $p_{acc} = 0.25$

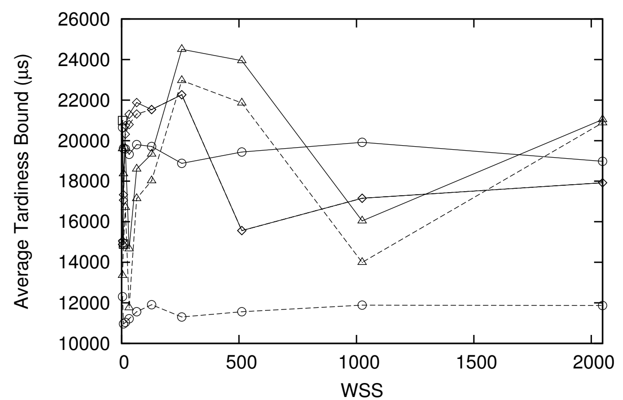


Figure 9: Medium Bimodal Utilization, Medium Uniform Periods, Long Critical Sections, $n_r = 6$, $p_{acc} = 0.25$

References

[1] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *ECRTS*, pp. 35–43, 2000.

[2] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[3] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[4] B. Brandenburg. Scheduling and locking in multiprocessor real-time operating systems dissertation companion page. `http://cs.unc.edu/~bbb/diss`, August 2011.

[5] B. Brandenburg and J. Anderson. Feather-trace: A lightweight event tracing toolkit. In *OSPERT*, pp. 61–70, 2007.

[6] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Sys.*, 38(2):133–189, 2008.

[7] J. Erickson and J. Anderson. Appendix to Reducing tardiness under global scheduling by splitting jobs. `http://cs.unc.edu/~anderson/papers.html`, October 2012.

[8] J. Erickson and J. Anderson. Fair lateness scheduling: Reducing maximum lateness in g-edf-like scheduling. In *ECRTS*, pp. 3–12, 2012.

[9] J. Erickson, U. Devi, and S. Baruah. Improved tardiness bounds for global EDF. In *ECRTS*, pp. 14–23, 2010.

[10] J. Erickson, N. Guan, and S. Baruah. Tardiness bounds for global EDF with deadlines different from periods. In *OPODIS*, pp. 286–301, 2010. Revised version at `http://cs.unc.edu/~jerickso/opodis2010-tardiness.pdf`.

[11] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Sys.*, 44(1):26–71, 2010.

[12] T. Megel, R. Sirdey, and V. David. Minimizing task preemptions and migrations in multiprocessor optimal real-time schedules. In *RTSS*, pp. 37–46, 2010.

[13] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *RTSS*, pp. 104–115, 2011.